



A Siemens Business

---

# Questa® Verification Run Manager User Guide

Software Version 10.7c

Document Revision 3.5

---

© 1995-2018 Mentor Graphics Corporation  
All rights reserved.

This document contains information that is proprietary to Mentor Graphics Corporation. The original recipient of this document may duplicate this document in whole or in part for internal business purposes only, provided that this entire notice appears in all copies. In duplicating any part of this document, the recipient agrees to make every reasonable effort to prevent the unauthorized use and distribution of the proprietary information.

Note - Viewing PDF files within a web browser causes some links not to function (see [MG595892](#)).  
Use HTML for full navigation.

This document is for information and instruction purposes. Mentor Graphics reserves the right to make changes in specifications and other information contained in this publication without prior notice, and the reader should, in all cases, consult Mentor Graphics to determine whether any changes have been made.

The terms and conditions governing the sale and licensing of Mentor Graphics products are set forth in written agreements between Mentor Graphics and its customers. No representation or other affirmation of fact contained in this publication shall be deemed to be a warranty or give rise to any liability of Mentor Graphics whatsoever.

**MENTOR GRAPHICS MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.**

MENTOR GRAPHICS SHALL NOT BE LIABLE FOR ANY INCIDENTAL, INDIRECT, SPECIAL, OR CONSEQUENTIAL DAMAGES WHATSOEVER (INCLUDING BUT NOT LIMITED TO LOST PROFITS) ARISING OUT OF OR RELATED TO THIS PUBLICATION OR THE INFORMATION CONTAINED IN IT, EVEN IF MENTOR GRAPHICS HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

**U.S. GOVERNMENT LICENSE RIGHTS:** The software and documentation were developed entirely at private expense and are commercial computer software and commercial computer software documentation within the meaning of the applicable acquisition regulations. Accordingly, pursuant to FAR 48 CFR 12.212 and DFARS 48 CFR 227.7202, use, duplication and disclosure by or for the U.S. Government or a U.S. Government subcontractor is subject solely to the terms and conditions set forth in the license agreement provided with the software, except for provisions which are contrary to applicable mandatory federal laws.

**TRADEMARKS:** The trademarks, logos and service marks ("Marks") used herein are the property of Mentor Graphics Corporation or other parties. No one is permitted to use these Marks without the prior written consent of Mentor Graphics or the owner of the Mark, as applicable. The use herein of a third-party Mark is not an attempt to indicate Mentor Graphics as a source of a product, but is intended to indicate a product from, or associated with, a particular third party. A current list of Mentor Graphics' trademarks may be viewed at: [mentor.com/trademarks](http://mentor.com/trademarks).

The registered trademark Linux® is used pursuant to a sublicense from LMI, the exclusive licensee of Linus Torvalds, owner of the mark on a world-wide basis.

**End-User License Agreement:** You can print a copy of the End-User License Agreement from: [mentor.com/eula](http://mentor.com/eula).

Mentor Graphics Corporation  
8005 S.W. Boeckman Road, Wilsonville, Oregon 97070-7777  
Telephone: 503.685.7000  
Toll-Free Telephone: 800.592.2210  
Website: [mentor.com](http://mentor.com)  
Support Center: [support.mentor.com](http://support.mentor.com)

Send Feedback on Documentation: [support.mentor.com/doc\\_feedback\\_form](http://support.mentor.com/doc_feedback_form)

# Revision History

---

Revision	Changes	Status/ Date
3.5	<p>Modifications to improve the readability and comprehension of the content. Approved by Farshad Dailami.</p> <p>All technical enhancements, changes, and fixes listed in the Release Notes are reflected in this document.</p> <p>Approved by Tim Peeke.</p>	Released August 2018
3.4	<p>Modifications to improve the readability and comprehension of the content. Approved by Farshad Dailami.</p> <p>All technical enhancements, changes, and fixes listed in the Release Notes are reflected in this document.</p> <p>Approved by Tim Peeke.</p>	Released June 2018
3.3	<p>Modifications to improve the readability and comprehension of the content. Approved by Farshad Dailami.</p> <p>All technical enhancements, changes, and fixes listed in the Release Notes are reflected in this document.</p> <p>Approved by Tim Peeke.</p>	Released March 2018
3.2	<p>Modifications to improve the readability and comprehension of the content. Approved by Farshad Dailami.</p> <p>All technical enhancements, changes, and fixes listed in the Release Notes are reflected in this document.</p> <p>Approved by Tim Peeke.</p>	Released December 2017

**Author:** In-house procedures and working practices require multiple authors for documents. All associated authors for each topic within this document are tracked within the Mentor Graphics Technical Publication's source. For specific topic authors, contact Mentor Graphics Technical Publication department.

**Revision History:** Released documents maintain a revision history of up to four revisions. For earlier revision history, refer to earlier releases of documentation which are available on Support Center (<http://support.mentor.com>).



# Table of Contents

---

## Revision History

### Chapter 1

Verification Run Manager Overview .....	23
Use Models .....	24

### Chapter 2

Example Tutorial .....	27
Getting Started.....	28
RMDB Example .....	28
XML Syntax .....	30
Creating the Basic Structure for an RMDB .....	33
Creating an r mdb Element .....	33
Creating Runnable Elements .....	34
Adding Parameters and Parameter References.....	36
Adding preScript and postScript Actions to a Group.....	37
Adding execScript Actions for the Tests .....	39
Directed Tests .....	40
Pulling it All Together .....	42
Run the Example Tutorial in GUI Mode .....	42
Run in Batch Mode and Debug Regression Tests .....	52
Executing VRM .....	52
Finding the Runtime Files.....	52
Failure to Recognize an Action as a Simulation.....	53
A More Complete Example .....	53

### Chapter 3

Basic Operations .....	57
Modes of Operation .....	58
Execution Mode .....	58
Control/Abort Mode .....	58
Status Mode .....	59
Notification Mode.....	59
RMDB Database File.....	60
Comments in the RMDB File .....	61
Embedding Options in the RMDB File .....	62
Dry Run of the RMDB Database .....	62
Locating the RMDB Database .....	63
Validation of the RMDB Database.....	63
Split RMDB Database Among Several Files .....	64
Set the Search Path for Shell Scripts .....	65
Locating Script and Log Files.....	66

Status Reports . . . . .	68
Action Script Status Reports . . . . .	68
Exit Code Status . . . . .	70
Regression Run Status Report . . . . .	71
Creating the Text-based vrun Status Report . . . . .	71
Creating the HTML-based vrun Status Report . . . . .	72
Creating the TCL-Format vrun Status Report . . . . .	72
Advanced Tasks . . . . .	73
Regression Run Text Status Report Example . . . . .	76
Regression Run HTML Status Report Reference . . . . .	78
Regression Run Status Report Reference . . . . .	82
Managing the VRMDATA Directory . . . . .	87
Manage a Questa work Directory and HDL Sources . . . . .	88
Implement a Macro by Calling VRM Recursively . . . . .	90
Limit Concurrently Running Processes . . . . .	92
Override Repeat Counts on Runnable Nodes . . . . .	92
Integration with Compute Grid Systems . . . . .	94
Launch an Action Onto a Grid System . . . . .	94
Interaction with a Job Once Launched . . . . .	96
Finding the JobId . . . . .	96
Reporting on Grid-job Status . . . . .	96
Killing a Queued Job . . . . .	97
Specify Selected Runnables . . . . .	98
Showing which Runnables are Selected . . . . .	100
Selecting Instances of Repeating Runnables . . . . .	100
Wildcards and Repeating Runnables . . . . .	101
Selecting Runnables Based on Results of Previous Run . . . . .	102
Context Chains . . . . .	104
Rooted Context Chains . . . . .	106
Globbing in Context Chains . . . . .	107
Using Regular Expressions in Context Chains . . . . .	107
Adhoc Mode . . . . .	108
Batch Mode Usability Functions . . . . .	110
View RMDB Contents . . . . .	110
Build a Runlist of Runnables to Execute . . . . .	112
Executing the Selected Runnables . . . . .	113
Selecting Values for Embedded Options . . . . .	115
String Parameters . . . . .	117
Numeric Parameters . . . . .	117
Enumerated Parameters . . . . .	118
Interaction with Parameter Prompts . . . . .	119
Use Model Examples . . . . .	121
Example of Front-end Runnable Selection . . . . .	121
Combined Selection and Prompting Example . . . . .	122
VRM Email Functionality . . . . .	123
VRM Email Use Model . . . . .	123
VRM Email Settings and Parameters . . . . .	124
Manually Generated VRM Email . . . . .	126
Composing Email Settings from Various Sources . . . . .	127

## Table of Contents

---

Regression Run Modification . . . . .	130
Parameter and Attribute Modification . . . . .	130
Modifying an Active Regression from the Command Line . . . . .	131
Override Parameter Values from Command Line . . . . .	132
Limiting Parameter Overrides to a Specific Context . . . . .	133
Context Partial-Match Algorithm . . . . .	136
Access RMDB API via the vrun Command Line . . . . .	138
<b>Chapter 4</b>	
<b>Graphical User Interface (GUI) . . . . .</b>	<b>139</b>
GUI Environment . . . . .	139
VRM Cockpit Window . . . . .	142
Adding RMDB Files to the Project File . . . . .	144
Adding New Configurations to the Project File . . . . .	144
Edit VRM Configurations . . . . .	147
Quick Access to the Edit VRM Configuration Dialog Box . . . . .	147
Edit VRM Configuration Dialog Box Tabs . . . . .	148
Importing Existing Regression History . . . . .	155
Initiating a Regression Run . . . . .	155
Attach to Running Regression and/or Viewing the Completed Results . . . . .	156
Viewing Regression Results Over Time . . . . .	157
VRM Results Window . . . . .	159
Control Other Functions from the VRM Results Window . . . . .	161
VRM Results Window Filters . . . . .	162
Configuring Filters for the VRM Results Window . . . . .	162
Applying Filters in the VRM Results Window . . . . .	163
Filter Expressions Dialog Box . . . . .	165
Create Filter Expression Dialog Box . . . . .	166
VRM Results History Window . . . . .	168
VRM Project File and Configurations . . . . .	169
Invoke Batch-mode Regressions During a GUI Session . . . . .	169
Initial Start-up and Project Management . . . . .	171
Start-up and the toprunnables Attribute . . . . .	172
Relationship Between Configurations and VRM Data Directories . . . . .	173
Menus . . . . .	175
Menu Item Descriptions . . . . .	176
Project Control . . . . .	176
Project Creation and Maintenance . . . . .	178
Tree Control . . . . .	179
Regression Launch . . . . .	179
View VRM Cockpit Contents . . . . .	181
Viewing Regression Contents . . . . .	182
Job Control . . . . .	184
Display Refresh . . . . .	185
GUI Use Model . . . . .	186
Getting Started with the GUI . . . . .	187
Launching a New Regression from the VRM GUI Command Line . . . . .	187
Building Up a New Project File from Nothing . . . . .	187

Generate a Project File by Invoking an Existing Batch-mode vrun in GUI Mode . . . . .	190
Generate a Configuration by Invoking vrun at Transcript Prompt. . . . .	190
Configuring the GUI to Open a Text File in an External Editor . . . . .	190
Start a New Regression Run and/or View Results of Previous Regression Run . . . . .	192
Launch a New Regression Run from a Configuration Row. . . . .	192
View Results of a Previously Launched Regression Run . . . . .	193
Launch a New Regression Run from a History Row . . . . .	194
Launch a New Regression from VRM Results Window of a Previous Regression Run. .	195
Launch a New Regression from the Transcript Prompt. . . . .	195
Import the Results of Manually Launched Regression Runs into a Project. . . . .	196
Create a New Configuration that Points to an Existing VRM Data Directory. . . . .	198
Change Value of the VRM Data Directory After Regression Data is Accumulated . .	198
Launch Multiple Regression Runs in a Single VRM Data Directory . . . . .	200
Delete File and/or Status Event Log from the VRM Data Directory. . . . .	201
Analyze Results and Rerun Tests Based on the Analysis. . . . .	203
Execute the Initial Regression Run . . . . .	203
Analyze the Results and Rerun the Tests. . . . .	203
Counter Control in the VRM GUI . . . . .	204
Changing the Counter Configuration . . . . .	204
Configuration File for vrmcounter Commands . . . . .	204
Effect of Projects and Configurations on Batch-mode VRM Regressions . . . . .	207
Batch-mode Regression . . . . .	208
Interaction with the Legacy Mode -config Command-line Option . . . . .	208
Create RMDB File from Existing UCDB File. . . . .	209
Action Scripts . . . . .	209
Compilation and Post-processing . . . . .	211
<b>Chapter 5</b>	
<b>VRM Regression Author . . . . .</b>	<b>213</b>
VRM Regression Author Introduction . . . . .	213
Invocation Workflow - Server Only . . . . .	213
Invocation Workflow - Combined Server and Browser . . . . .	214
vrun -author . . . . .	216
VRM Regression Author GUI . . . . .	217
<b>Chapter 6</b>	
<b>RMDB Database Topology . . . . .</b>	<b>219</b>
RMDB Database Requirements. . . . .	219
Database Content and Naming Conventions . . . . .	220
Database Structure . . . . .	222
Runnables and the Regression Suite Hierarchy . . . . .	223
Execution Method Elements. . . . .	224
User-defined (usertcl) TCL Code Fragments . . . . .	224
Execution Model . . . . .	225
Actions and Commands. . . . .	227
Repeating a Task or a Group . . . . .	229
Controlling the Repeat Index (foreach Functionality) . . . . .	230
Dynamically Repeating Runnables . . . . .	233

## Table of Contents

---

Use Model . . . . .	234
Activating the Feature . . . . .	234
Controlling the Feature . . . . .	236
Lexical Requirements for foreach Attribute Values and Tokens . . . . .	237
GetNextIteration Procedure . . . . .	238
Concurrency and Multiple Repeating Runnable Instances . . . . .	239
Repeating Runnable Example . . . . .	240
Selective Membership in a Group . . . . .	243
Basic Regression Execution Algorithm . . . . .	245
Run (and Rerun) Algorithm . . . . .	247
Notification of Regression Start and/or Completion . . . . .	247
Failed Test Rerun . . . . .	247
Job Progress Monitoring . . . . .	247
Test-centric Reporting and Control . . . . .	249
Test-centric Limitations . . . . .	249
Test-centric Reporting . . . . .	250
Defining Test Names in the RMDB File . . . . .	250
Enabling a Test-centric Regression . . . . .	252
Controlling Action Scripts by Test Name . . . . .	252
<b>Chapter 7</b> <b>Execution Graph and Advanced Topology . . . . .</b>	<b>255</b>
Execution Graph Expansion . . . . .	256
Execution Graph Example . . . . .	258
Restrictions on Group Membership . . . . .	259
Handling Errors from Action Scripts . . . . .	260
Passing Exit Codes From Action Scripts to vrun . . . . .	260
<b>Chapter 8</b> <b>Parameters and Parameter Expansion . . . . .</b>	<b>263</b>
Predefined Parameter Usage . . . . .	266
Including TCL Code in a Parameter Definition . . . . .	267
Referring to Source Files Via a Parameter . . . . .	269
Nested Parameter Expansion and TCL Parameters . . . . .	270
Inheritance, Overrides, and Search Order . . . . .	271
Parameter Inheritance Example . . . . .	278
Runnable Types and Inheritance . . . . .	280
Search Order When Expanding a Parameter Reference . . . . .	281
Inheritance Rules . . . . .	281
Parameter Expansion Rules . . . . .	281
Parameterizing the Regression Suite for Debug . . . . .	283
Exporting Parameters to Action Scripts . . . . .	284
<b>Chapter 9</b> <b>Working Directory and Script Execution . . . . .</b>	<b>287</b>
Working Directory Overview . . . . .	288
Local File Generation . . . . .	289
Code Examples . . . . .	294

Disposition of Test Collateral . . . . .	295
File Safety . . . . .	295
Script Execution and Timeouts . . . . .	296
Action and Wrapper Script Generation . . . . .	297
vsim Mode Wrapper . . . . .	298
exec Mode Wrapper . . . . .	300
Using Existing Scripts with VRM . . . . .	302
Passing Values Via Position-dependent Parameters . . . . .	302
Passing Values Via Environment Variables . . . . .	303
Copying the Script File to the Working Directory . . . . .	303
Running Scripts in Existing Directory Structure . . . . .	304
Script Return Values . . . . .	306
Modifying vsim Mode User Scripts to Return Status . . . . .	307
Modifying exec Mode User Scripts to Return Status . . . . .	308
Execution Mode Simulations Hang . . . . .	308
Pipe-based Command Launching . . . . .	309
Sending User-defined Messages from an Action Script . . . . .	310
Spoofing Default Behavior of Action Scripts . . . . .	311
Timeouts . . . . .	311
Execution Methods . . . . .	315
Selecting an Execution Method . . . . .	316
Methods and Timeouts . . . . .	317
Conditional Execution Methods . . . . .	318
Methods and Base Inheritance . . . . .	321
Execution Method Examples . . . . .	327
Recommended Execution Method Configurations . . . . .	327
Timing the Execution of Each Action . . . . .	330
Queuing Actions to Run on a Server Grid . . . . .	330
Execution Queues and Job Aggregation . . . . .	333
Named Execution Queues . . . . .	333
Aggregating Multiple Actions in a Single Job . . . . .	336
Multi-Action Clubbing Fixed-list Mode . . . . .	336
Creating Grid Array Jobs . . . . .	337
Common Rules for Clubbed Jobs and Array Jobs . . . . .	338
Controlling the Grid Submission Command . . . . .	339
Generating and Launching Multi-Action Script Files . . . . .	341
Multi-Action Grid Jobs and Concurrency . . . . .	343
Multi-Action Grid Jobs and Local Rerun . . . . .	343
Multi-Action Grid Jobs and maxrunning Execution Limits . . . . .	343
Multi-Action Grid Jobs and Job Control . . . . .	343
Limiting the Execution Time of a Given Named Queue . . . . .	344
Failure Rerun Functionality . . . . .	346
Local (Immediate) Rerun . . . . .	347
Action Flow . . . . .	347
Default Behavior . . . . .	348
OkToRerun Procedure . . . . .	349
Possible Use Models . . . . .	350
Global (Command-line Based) Method . . . . .	351
Logging Failed Actions . . . . .	351

## Table of Contents

---

Replaying Failed Actions . . . . .	351
Global (Parameter-based) Method . . . . .	353
Semi-automatic (Two-command) Mode . . . . .	354
Automating Failure Rerun With a Macro Runnable . . . . .	355
Fully Automatic (One-command) Mode . . . . .	357
Reuse of Random Seeds in Automated Rerun . . . . .	358
<b>Chapter 10</b>	
<b>Post-execution Analysis . . . . .</b>	<b>361</b>
Pass/Fail Analysis . . . . .	364
Failure Propagation . . . . .	365
Determining Pass/Fail for a Given Action . . . . .	367
ActionCompleted Concept . . . . .	369
OkToMerge Concept . . . . .	369
MergeOneUcdb Concept . . . . .	370
StopRunning Concept . . . . .	370
StopPropagating Concept . . . . .	371
Auto-Delete Functionality . . . . .	372
Execution Graph . . . . .	372
Coverage Merge Options . . . . .	376
Assumptions . . . . .	376
Automated Merging Example . . . . .	376
Post-execution Processing . . . . .	378
Coverage Merging Use Models . . . . .	379
Do-It-Yourself (DIY) Merging . . . . .	379
List-based Merging . . . . .	384
Incremental Merging . . . . .	386
Queued Merging . . . . .	388
Queued Merging Example . . . . .	390
Auto-Merge Tips and Traps . . . . .	392
Auto-Merging and Delete-on-Pass . . . . .	392
Sorting Coverage Information Based on Test Status . . . . .	392
Automatic Deletion of Merge/Triage Files . . . . .	394
Auto-merge and Auto-triage Use of -inputs Argument . . . . .	395
Testplan Auto-import . . . . .	395
Enabling and/or Disabling Coverage Merge Flows . . . . .	396
Automated Results Analysis . . . . .	398
UCDB File “clubbing” in Automated Merge and Automated Triage . . . . .	400
Automated Trend Analysis . . . . .	402
Multi-level Auto-merge Example . . . . .	403
Auto-merge Example Design . . . . .	403
Auto-merge Example RMDB Database . . . . .	403
First-level Merge in Detail . . . . .	405
Avoiding Problems with Inheritance . . . . .	406
Second-level Merge in Detail . . . . .	408
Complete Multi-level Auto-merge Example . . . . .	408
User-definable Procedures and Loading of Arbitrary TCL Code . . . . .	412
Modifying VRM Behavior with User-definable Procedures . . . . .	413

Loading User-supplied TCL Code Using usertcl into VRM .....	415
Collate Pass/Fail UCDB Files for Merge and/or Triage .....	422
Basic Regression Suite Example .....	423
Fully Automated Merge and Triage .....	424
Using the Event Log to Gather UCDB Files .....	426
Override OkToMerge and Rely on Auto-merge.....	427
Maintain Separate Lists of UCDBs Internal to VRM.....	428
Enabling Auto-Delete .....	430
Protecting Select Files from Deletion.....	430
Disabling Auto-Delete from the Command Line .....	432
Listing Actions from the Command Line .....	433
Deletion Process .....	433
Auto-Clean .....	434
Auto-Delete and Auto-Triage .....	434
<b>Chapter 11</b>	
<b>Reference .....</b>	<b>437</b>
vrun Command .....	438
vrmcounter.....	459
Job Control Options Quick Reference.....	459
Message Verbosity and Debugging .....	460
Definitions of Terms .....	461
XML Terms .....	464
Typographical Conventions.....	465
<b>Chapter 12</b>	
<b>Communication Channels, Event Log, and Status Report .....</b>	<b>467</b>
Action Script to VRM .....	467
Unattended Batch Mode .....	468
Communication Channels .....	469
Basic Communication Model .....	469
Format of Messages from Action Script to vrun Core .....	470
Error Response .....	471
Event Logging .....	471
Attach to a Running vrun Process via Event Log .....	473
Send Status Message from Wrapper/User Script.....	475
<b>Chapter 13</b>	
<b>Controlling Grid Jobs.....</b>	<b>477</b>
Specifying the Type of Grid System .....	478
Specifying the Job Control Commands .....	480
Pro-active Grid Status Checks .....	481
Locally Launched Jobs .....	482
Predefined User-definable Procedures .....	483
Controlling Jobs Remotely .....	484
Command Line Control from a Second vrun Process .....	485
Adjust Timeout Values to Account for Time Suspended.....	487
Running with JobSpy .....	487

---

## Table of Contents

---

<b>Chapter 14</b>	
<b>Environment Variables and Built-in Parameters .....</b>	<b>489</b>
Environment Variables .....	489
Predefined Parameters .....	492
Parameters Recognized/Defined by VRM.....	493
Parameterized Element Attributes .....	495
Relative File Rules.....	496
<b>Chapter 15</b>	
<b>Catalog of User-Definable and Utility Procedures .....</b>	<b>499</b>
Catalog Format .....	499
User Definable Procedures .....	501
Common Arguments Passed to All User Definable Procedures.....	501
Global Status Change Procedures.....	504
RegressionStarting .....	504
RegressionCompleted.....	504
ProcessIdleTasks .....	505
Action Status Change Procedures.....	506
ActionEligible .....	506
ActionLaunched .....	506
ActionStarted .....	507
ActionCompleted .....	507
Post-execution Processing Procedures .....	509
AnalyzePassFail .....	509
GetUcdbCoverage .....	511
IsMergedUcdb .....	511
MergeRerun .....	512
OkToRerun .....	512
OkToMerge .....	513
MergeOneUcdb .....	513
OkToTrend .....	514
OkToTriage .....	515
TriageOneUcdb .....	516
OkToDelete .....	517
Regression Control Procedures.....	520
SelectRunnables .....	520
StopPropagating .....	521
StopRunning .....	522
GetNextIteration .....	523
Grid Management Procedures .....	525
<gridtype>SubmitOptions .....	526
<gridtype>GetJobId .....	527
<gridtype>GetJobStatus .....	528
<gridtype>GetStderr .....	529
<gridtype>KillJob .....	530
<gridtype>CacheStatus .....	530
<gridtype>GetCachedStatus.....	531
<gridtype>GetLastStatus .....	532

Logging Procedures . . . . .	534
ProclaimPassFail . . . . .	534
ProclaimUserMessage . . . . .	535
Utility Procedure Catalog . . . . .	537
System-level Utilities . . . . .	538
testMode . . . . .	538
ActionToTestname . . . . .	539
SeedRandomGenerator . . . . .	539
GetRandomValues . . . . .	539
ExpandTestlist . . . . .	540
GetTestlistTokens . . . . .	541
GetMostRecentFile . . . . .	542
GetMostRecentFileContents . . . . .	543
GetStatusCounts . . . . .	543
RightNow . . . . .	545
GetNextToken . . . . .	545
GetStderrForAction . . . . .	546
TranslateExtStatus . . . . .	546
TranslateRunStatus . . . . .	547
Path-conversion Procedures . . . . .	548
PathRelativeToDatadir . . . . .	548
PathRelativeToVrun . . . . .	548
PathRelativeToRmdb . . . . .	549
PathRelativeToQuesta . . . . .	549
ActionToLogFile . . . . .	550
ActionToLogFile . . . . .	550
File and Directory Manipulation Procedures . . . . .	552
VerifySafeTarget . . . . .	552
MakeDirectory . . . . .	553
WriteToFile . . . . .	553
AppendToFile . . . . .	554
CopyFile . . . . .	554
LinkToFile . . . . .	555
GetFileLines . . . . .	555
GetFileContents . . . . .	555
DoNotDelete . . . . .	556
Parameter Expansion Procedure . . . . .	558
OverrideRmdbParameter . . . . .	558
ExpandRmdbParameters . . . . .	558
UCDB Access and Analysis Procedures . . . . .	560
GetUcdbTestAttribute . . . . .	560
GetUcdbSummaryAttribute . . . . .	560
GetUcdbSeed . . . . .	561
GetUcdbStatus . . . . .	561
ConvertUcdbStatus . . . . .	562
IsCoverstore . . . . .	563
IsTplanUcdb . . . . .	563
IsSweepUcdb . . . . .	564
FindMaxTeststatus . . . . .	564

## Table of Contents

---

TranslateUcdbStatus . . . . .	564
CheckUcdbLock . . . . .	565
ValidateUcdb . . . . .	565
Event Log Access Procedures . . . . .	567
FetchEventRecords . . . . .	567
FetchEventActions . . . . .	568
FetchEventUcdbs . . . . .	568
Email-related Procedures . . . . .	569
AutoEmail . . . . .	569
AutoEmailSignature . . . . .	569
GenerateStatusSummary . . . . .	569
GetDefaultEmailParameters . . . . .	570
GetEmailParameters . . . . .	570
SendEmailMessage . . . . .	571
<b>Appendix A</b>	
<b>VRM Database (RMDB) API . . . . .</b>	<b>573</b>
Types of Elements . . . . .	573
Data and Meta-data . . . . .	575
Definitions of Variables Used in Commands . . . . .	576
Command Shortcuts . . . . .	577
Attributes Per Element Type . . . . .	578
Element Attribute Default Values . . . . .	582
Parameter and Script Inheritance . . . . .	583
Document Type Definition and Validation . . . . .	583
Including Nested XML Files . . . . .	583
Global API . . . . .	584
Read access API . . . . .	587
Write Access API . . . . .	590
Cached Access API . . . . .	597
Opening and Closing Cached RMDB Files . . . . .	597
Additional Commands . . . . .	598
Enhanced Commands . . . . .	598
<b>Appendix B</b>	
<b>Replacing SCRATCH with VRMDATA . . . . .</b>	<b>601</b>
Terminology and Typography . . . . .	601
Command-line Option and Default Value . . . . .	602
Pre-defined Parameters . . . . .	603
User-definable Procedures and TCL Utility Procedures . . . . .	603
<b>Appendix C</b>	
<b>RMDB Reference . . . . .</b>	<b>605</b>
Element Definitions . . . . .	606
rmdb . . . . .	607
runnable . . . . .	608
parameters and parameter . . . . .	614
members and member . . . . .	616

---

## Table of Contents

localfile .....	616
preScript .....	618
execScript .....	619
postScript .....	621
mergeScript .....	623
tplanScript .....	624
trendScript .....	626
triageScript .....	627
method .....	629
command .....	632
userctl .....	633
<b>Appendix D</b>	
<b>RMDB Versions .....</b>	<b>635</b>
Version Descriptions .....	636
RMDB 1.0 .....	636
RMDB 1.1 .....	636
<b>End-User License Agreement</b>	

# List of Figures

---

Figure 1-1. Verification Run Manager Flow Diagram .....	24
Figure 2-1. default.rmdb File .....	28
Figure 2-2. top.sv File .....	30
Figure 2-3. Questa Verification Run Manager GUI - Tutorial .....	43
Figure 2-4. Default Configuration .....	44
Figure 2-5. Edit VRM Configuration nightly Selected .....	45
Figure 2-6. Create a VRM Data Directory .....	45
Figure 2-7. VRM Cockpit Run .....	46
Figure 2-8. Regression Run in Progress .....	47
Figure 2-9. Regression Run Completed .....	48
Figure 2-10. VRM Cockpit .....	49
Figure 2-11. Action Log File .....	50
Figure 2-12. Open Files in VRM Data Directory Window .....	50
Figure 2-13. execScript.do File .....	51
Figure 3-1. RMDB File Example .....	60
Figure 3-2. XInclude Allows Database Contents Divided Among Multiple Files .....	64
Figure 3-3. Regression Run HTML Status Report .....	79
Figure 3-4. VRM Data Directory Tree Structure .....	87
Figure 3-5. Predefined DATADIR and RMDBDIR Directories in a Database .....	89
Figure 3-6. Enum and Foreach Examples .....	119
Figure 4-1. Questa Verification Run Manager GUI .....	140
Figure 4-2. Default Configuration Created Dialog Box .....	141
Figure 4-3. Project File Loaded into the VRM Cockpit Window .....	143
Figure 4-4. Edit VRM Configuration Dialog Box — General Tab .....	149
Figure 4-5. Edit VRM Configuration Dialog Box — Runnable Tab .....	150
Figure 4-6. Edit VRM Configuration Dialog Box — Parameters Tab .....	151
Figure 4-7. Edit VRM Configuration Dialog Box — Runtime Tab .....	152
Figure 4-8. Edit VRM Configuration Dialog Box — Features Tab .....	153
Figure 4-9. Edit VRM Configuration Debug Tab .....	154
Figure 4-10. VRM Results Window .....	157
Figure 4-11. VRM Results History Window .....	158
Figure 4-12. Filter Expressions Dialog Box .....	165
Figure 4-13. Create Filter Expression Dialog Box .....	166
Figure 4-14. Open VRM Project .....	169
Figure 4-15. Run Manager Prompt .....	170
Figure 4-16. Open File in VRM Data Directory .....	182
Figure 4-17. Verification Management Browser .....	183
Figure 4-18. Job Control Options .....	184
Figure 4-19. Add Item RMDB File .....	188
Figure 4-20. Create VRM Configuration Name .....	189

Figure 4-21. Edit Regression Options Window . . . . .	193
Figure 4-22. Transcript Window . . . . .	196
Figure 4-23. Import VRM Data Directory into VRM Cockpit . . . . .	197
Figure 4-24. Select VRM Data Directory Window . . . . .	197
Figure 4-25. Imported Results . . . . .	198
Figure 4-26. Verify File/Directory Deletion . . . . .	201
Figure 4-27. VRM Data (doesn't exist) . . . . .	202
Figure 6-1. VRM Execution Model . . . . .	226
Figure 7-1. Grouping of Tasks and Groups Diagram . . . . .	258
Figure 8-1. Parameter Inheritance Diagram . . . . .	279
Figure 10-1. Post-execution Analysis Flow . . . . .	362
Figure 10-2. Execution Graph . . . . .	372
Figure 10-3. Execution Graph Including deleteScript . . . . .	373
Figure 10-4. Execution Graph After mergeScript is Added . . . . .	374
Figure 10-5. Automated Merging Example . . . . .	377
Figure 10-6. DIY Merge Invoking vcover merge After Each Simulation . . . . .	381
Figure 10-7. DIY Merge Adding postScript to Top-level Group Runnable . . . . .	382
Figure 10-8. DIY Merge User-defined TCL Override Procedure . . . . .	383
Figure 10-9. List-based Merge postScript Action at end of Simulation . . . . .	385
Figure 10-10. Incremental Merge with mergefile Parameter . . . . .	387
Figure 10-11. Results Analysis (Triage) Flow . . . . .	398
Figure 10-12. Topology of the Example RMDB Database File . . . . .	404
Figure 10-13. Multi-level Auto-merge . . . . .	409
Figure 10-14. Cancel Regression Run Code Fragment . . . . .	414
Figure 10-15. Delete-on-pass Implementation . . . . .	414
Figure 10-16. Basic Regression Suite . . . . .	423
Figure 12-1. Batch Regression Run . . . . .	468
Figure 12-2. Event Log File Sample . . . . .	472
Figure 12-3. Incomplete Event Log File . . . . .	474
Figure 13-1. LSF Grid Management System Regression Sample . . . . .	478
Figure 14-1. Hard-code Questa or 3rd Party Tool Release Path in Script . . . . .	491

# List of Tables

---

Table 2-1. Special Character Substitutions in an XML File .....	31
Table 3-1. Modes of Operation .....	58
Table 3-2. vrun -exitcodes Bit-Wise Definition .....	70
Table 3-3. Status Report Output Columns .....	82
Table 3-4. Possible Values for the Various Status Columns .....	84
Table 3-5. Filter Operators .....	86
Table 3-6. Status Keywords .....	103
Table 3-7. accept Attribute Values .....	116
Table 3-8. Numeric Parameter Limit Strings .....	118
Table 3-9. VRM Email Settings .....	124
Table 3-10. VRM Email Settings — section Keywords .....	125
Table 4-1. Create Filter Expression Dialog Box - Field Box Options .....	167
Table 4-2. Generate RMDB File Menu Picks .....	209
Table 6-1. XML Internal Entities for Meta-characters .....	220
Table 8-1. RMDB Database Data Item Types .....	278
Table 9-1. Conditional Attributes for the method Element .....	319
Table 9-2. Status Test Keyword Types .....	354
Table 9-3. Composite Keywords Recognized .....	354
Table 10-1. Possible UCDB Return Values .....	368
Table 10-2. Coverage Merge Flow Summary .....	396
Table 10-3. Automated Results Analysis (Triage) Modes .....	399
Table 10-4. Merges for Avoiding Inheritance Problems in Example .....	407
Table 10-5. Methods of Collating Passed/Failed UCDB Files .....	422
Table 10-6. DoNotDelete Commands .....	431
Table 11-1. vrun Job Control Options Quick Reference .....	459
Table 11-2. Jobs Launched Locally vs. Jobs Launched on Grid .....	460
Table 12-1. Event Field Strings .....	470
Table 13-1. Post-launch Control Operations .....	477
Table 13-2. Signals Sent Via the Kill Command .....	482
Table 13-3. vrun Messages .....	484
Table 13-4. Regression Run Control Command Options .....	485
Table 14-1. VRM Environment Variables .....	489
Table 14-2. Predefined Parameters .....	492
Table 14-3. Special Named Parameters .....	493
Table 14-4. Parameterized Element Attributes .....	495
Table 14-5. File/Pathname Specification Options .....	496
Table 15-1. Common Arguments Passed to All User-definable Procedures .....	501
Table 15-2. Common Arguments Passed to Action- and Runnable-related Procedures ..	502
Table 15-3. Input Arguments Passed Conditionally .....	502
Table 15-4. Data Elements Passed to the Post-execution Procedures .....	502

Table 15-5. passfail Keywords . . . . .	508
Table 15-6. Input Arguments for GetUcdbCoverage Procedure . . . . .	511
Table 15-7. Input Arguments for IsMergedUcdb Procedure . . . . .	511
Table 15-8. Arguments Passed in as Data Elements set by OkToMerge . . . . .	514
Table 15-9. Arguments Passed in as Data Elements set by OkToTrend . . . . .	515
Table 15-10. Arguments Passed as Data Elements set by OkToTriage . . . . .	516
Table 15-11. Keywords for passfail Element . . . . .	517
Table 15-12. Keywords for ucdbstat Element . . . . .	518
Table 15-13. Element Members for SelectRunnables . . . . .	520
Table 15-14. SelectRunnables Input Arguments . . . . .	521
Table 15-15. StopPropagating Input Arguments . . . . .	522
Table 15-16. StopRunning Input Arguments . . . . .	523
Table 15-17. Postfix String and Behavior . . . . .	525
Table 15-18. LsfSubmitOptions Input Arguments . . . . .	526
Table 15-19. GetJobStatus Input Arguments . . . . .	528
Table 15-20. ProclaimPassFail Arguments . . . . .	534
Table 15-21. ProclaimUserMessage Arguments . . . . .	536
Table 15-22. GetNextToken Arguments . . . . .	545
Table 15-23. GetStderrForAction Argument . . . . .	546
Table 15-24. TranslateExtStatus Argument . . . . .	547
Table 15-25. TranslateRunStatus Argument . . . . .	547
Table 15-26. PathRelativeToDatadir Input Arguments . . . . .	548
Table 15-27. PathRelativeToVrun Input Arguments . . . . .	548
Table 15-28. PathRelativeToRmdb Input Arguments . . . . .	549
Table 15-29. PathRelativeToQuesta Input Arguments . . . . .	549
Table 15-30. ActionToFile Input Arguments . . . . .	550
Table 15-31. ActionToFiles Input Arguments . . . . .	550
Table 15-32. VerifySafeTarget Input Arguments . . . . .	552
Table 15-33. MakeDirectory Input Arguments . . . . .	553
Table 15-34. WriteToFile Input Arguments . . . . .	553
Table 15-35. AppendToFile Input Arguments . . . . .	554
Table 15-36. CopyFile Input Arguments . . . . .	554
Table 15-37. LinkToFile Input Arguments . . . . .	555
Table 15-38. GetFileLines Input Arguments . . . . .	555
Table 15-39. GetFileContents Input Arguments . . . . .	556
Table 15-40. DoNotDelete Input Arguments . . . . .	556
Table 15-41. OverrideRmdbParameter Input Arguments . . . . .	558
Table 15-42. ExpandRmdbParameters Input Arguments . . . . .	558
Table 15-43. GetUcdbTestAttribute Input Arguments . . . . .	560
Table 15-44. GetUcdbSummaryAttribute Input Arguments . . . . .	561
Table 15-45. GetUcdbSeed Input Arguments . . . . .	561
Table 15-46. GetUcdbStatus Input Arguments . . . . .	562
Table 15-47. Strings Returned for GetUcdbStatus . . . . .	562
Table 15-48. ConvertUcdbStatus Input Arguments . . . . .	563
Table 15-49. IsCoverstore Input Arguments . . . . .	563

## List of Tables

---

Table 15-50. IsTplanUcdb Input Arguments .....	563
Table 15-51. IsSweepUcdb Input Arguments .....	564
Table 15-52. FindMaxTeststatus Input Arguments .....	564
Table 15-53. TranslateUcdbContext Input Arguments .....	564
Table 15-54. CheckUcdbContext Lock Input Arguments .....	565
Table 15-55. ValidateUcdbContext Input Arguments .....	565
Table 15-56. FetchEventRecords Options .....	567
Table 15-57. FetchEventRecords -return Selections .....	567
Table A-1. Variables Used in Commands .....	576
Table A-2. Keys for rmdb method Command .....	577
Table A-3. Document Attributes .....	578
Table A-4. runnable Attributes .....	579
Table A-5. Parameter Attributes .....	579
Table A-6. action Attributes .....	580
Table A-7. localfile Attributes .....	580
Table A-8. method Attributes .....	581
Table A-9. usertcl Attributes .....	581
Table A-10. Enumerated Type Attributes .....	582
Table A-11. Options for dbName Argument .....	597
Table C-1. rmdb Element — Allowed Child Elements .....	607
Table C-2. rmdb Element — Supported Attributes .....	607
Table C-3. runnable Element — Allowed Child Elements .....	609
Table C-4. runnable Element — Supported Attributes .....	610
Table C-5. parameters Element — Allowed Child Elements .....	614
Table C-6. parameter Elements — Supported Attributes .....	614
Table C-7. members Element — Allowed Child Elements .....	616
Table C-8. localfile Element — Allowed Child Elements .....	616
Table C-9. localfile Element — Supported Attributes .....	617
Table C-10. preScript Element — Allowed Child Elements .....	618
Table C-11. preScript Element — Supported Attributes .....	618
Table C-12. execScript Element — Allowed Child Elements .....	619
Table C-13. execScript Element — Supported Attributes .....	620
Table C-14. postScript Element — Allowed Child Elements .....	621
Table C-15. postScript Element — Supported Attributes .....	621
Table C-16. mergeScript Element — Allowed Child Elements .....	623
Table C-17. mergeScript Element — Supported Attributes .....	623
Table C-18. tplanScript Element — Allowed Child Elements .....	625
Table C-19. tplanScript Element — Supported Attributes .....	625
Table C-20. trendScript Element — Allowed Child Elements .....	626
Table C-21. trendScript Element — Supported Attributes .....	626
Table C-22. triageScript Element — Allowed Child Elements .....	627
Table C-23. triageScript Element — Supported Attributes .....	628
Table C-24. method Element — Allowed Child Elements .....	629
Table C-25. method Element — Supported Attributes .....	629
Table C-26. usertcl Element — Supported Attributes .....	633



# Chapter 1

## Verification Run Manager Overview

---

Verification Run Manager (VRM) provides a mechanism for the user to define an arbitrary number of regression tasks, execute the entire set of tasks, or any subset of tasks with a single command.

VRM features enable the user to do the following:

- Organize, optimize, and manage regressions.
- Automate the regression process.
- Manage the simulation regression results efficiently, which enhances debug productivity.
- Transform and sort regression results in a way that aids in fast bug identification.
- Manage different kinds of compute job. For example, 0-In Formal runs, Questa SIM simulation runs, or runs of other executables.

VRM can be invoked from the command line or by an automated script as part of a scheduled regression run. Once invoked, VRM consults a database and initiates one or more “jobs” (also called “Actions”). These Actions can be handled as follows:

- Execute as background tasks on the local machine.
- Exported to a specific server.
- Queued for execution on a grid system.

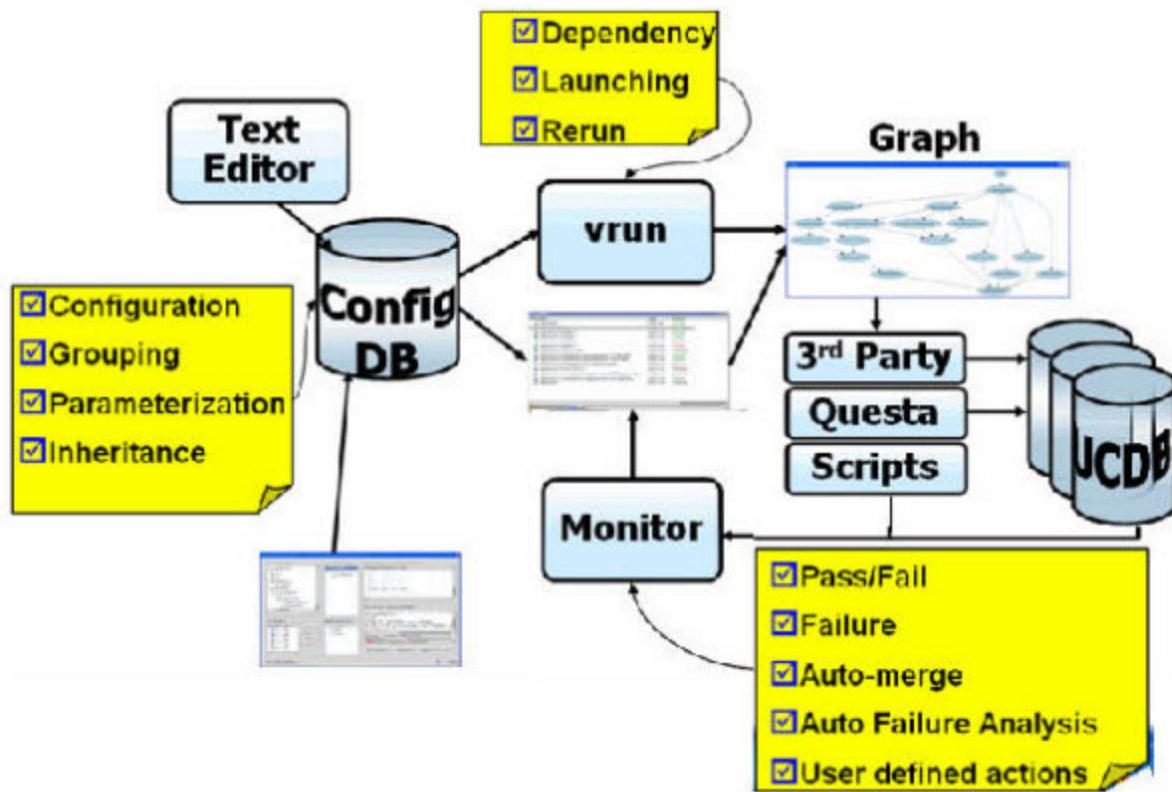
The resulting jobs are monitored in order to satisfy any inter-dependence requirements. When a job is completed, notification can be passed to one or more downstream activities (that is, triage, trending, loading results into a running test browser, and so on).

VRM contains many built-in short-cut Questa® hooks to address the conflict between generality and ease-of-use for Questa users. For example, Unified Coverage DataBase (UCDB) files can be automatically collected and merged without the user needing to explicitly specify complex merge scripts, based on the presence of a specially recognized parameter.

There are conventions for Questa users that make it possible for Questa to perform automated post-execution tasks. (See “[Post-execution Analysis](#)” on page 361”).

[Figure 1-1](#) illustrates the Verification Run Manager flow.

Figure 1-1. Verification Run Manager Flow Diagram



Use Models ..... 24

## Use Models

Following are some use models for VRM:

- Nightly regressions

A project-wide Run Manager DataBase (RMDB) file defines a top-level test suite, possibly consisting of smaller subgroups of tests. The run is kicked off in batch mode, unattended. Jobs are farmed out to a grid manager.

- Desktop run of user's tests

A user-maintained RMDB file consists of individual tests grouped into one or more suites. The user kicks off test runs manually. The tests run in batch mode (foreground or background). The user generally waits for the results.

- Rerun of failing tests after the source code fix

A group of tests are selected from a list of failed tests, often from a GUI browser. Test definitions are drawn from the project-wide RMDB file. A temporary *suite* is generated to define the list of tests to run. The tests run interactively. The user waits for the results.

- Automated rerun in debug mode

The failing tests from a nightly run can be automatically rerun in debug mode in order to gather additional debug information (for example, *WLF* file with logging of many signals enabled) that can help trace the failures.

- Smoke test prior to check-in

A suite of tests or short test suites are preselected (from the project-wide RMDB file) for maximum short/broad coverage and added to a *short-broad* suite. These same tests generally fall into larger suites within the project-wide RMDB file. The short-broad run is kicked off by the user manually. Each test runs in the background (or farmed out to a grid manager). The user is notified when all the tests are done.

- Search for high-coverage random seeds

A random test can be run a fixed number of times (possibly overnight). At the end of the run, coverage from each test can be ranked and the most promising seeds added to the database for regular execution.



# Chapter 2

## Example Tutorial

---

This tutorial walks through the process of setting up a VRM Run Manager DataBase (RMDB) manually.

- The RMDB defines a trivial regression test. The test suite includes three directed tests, the stimulus of which is enabled by conditional compilation directives in the single design source file.
- The design source has a random-stimulus mode (enabled if no directed test is selected). It selects three promising seeds to try in random-stimulus mode as part of the nightly regression.
- The regression suite consists of six simulations. Once the six simulations are run, it then generates a single merged coverage Unified Coverage DataBase (UCDB) file.

Refer to “[Definitions of Terms](#)” on page 461 for a description of the terms used in this tutorial.

<b>Getting Started.....</b>	<b>28</b>
<b>Run the Example Tutorial in GUI Mode.....</b>	<b>42</b>
<b>Run in Batch Mode and Debug Regression Tests.....</b>	<b>52</b>
<b>A More Complete Example.....</b>	<b>53</b>

# Getting Started

---

You can find the source files for this example tutorial can be found in the *<install\_dir>/examples/vrm/simple* directory.

The example directory contains an RMDB database (see [Example 2-1](#)) and a single design source file (see [Example 2-2](#)):

```
simple/
    default.rmdb <!-- default RMDB file name is default.rmdb -->
    README
    src/
        top.sv
```

This example tutorial teaches you how to write a Run Manager Database file (*.rmbd* file), which is implemented using XML. XML is designed to transport and store data.

<b>RMDB Example .....</b>	<b>28</b>
<b>XML Syntax .....</b>	<b>30</b>
<b>Creating the Basic Structure for an RMDB .....</b>	<b>33</b>

## RMDB Example

The RMDB for the tutorial is named *default.rmbd*.

[Example 2-1](#) is the database *default.rmbd* file for this simple example tutorial. Observe that the *default.rmbd* file has the line numbers turned on. This line number file is used to explain how to write a Run Manager Database file (see “[Creating the Basic Structure for an RMDB](#)” on page 33).

In addition, basic XML Syntax is explained (see “[XML Syntax](#)” on page 30).

**Figure 2-1. default.rmbd File**

```
1  <?xml version="1.0" ?><rmbd version="1.1">
2      <runnable name="nightly" type="group">
3          <parameters>
4              <parameter name="mergefile">merge.ucdb</parameter>
5          </parameters>
6          <members>
7              <member>directed</member>
8              <member>random</member>
9          </members>
10         <postScript launch="vsim">
11             <command>vcover attr -name ORIGFILENAME -name TESTSTATUS
12                 (%mergefile%)</command>
13             </postScript>
14         </runnable>
15         <!-- ===== -->
16         <!-- DIRECTED TESTS -->
```

```
16  <!-- ===== -->
17  <runnable name="directed" type="group">
18      <parameters>
19          <parameter name="ucdbfile">../(%INSTANCE%).ucdb</parameter>
20      </parameters>
21      <members>
22          <member>dirtest1</member>
23          <member>dirtest2</member>
24          <member>dirtest3</member>
25      </members>
26      <execScript>
27          <command>vlib work</command>
28          <command>vlog -sv +define+dirtest=\"(%INSTANCE%) \\" (%RMDBDIR%)/src/
top.sv</command>
29          <command>vsim -c top</command>
30          <command>run -all</command>
31          <command>coverage attribute -name TESTNAME -value (%INSTANCE%) </
command>
32          <command>coverage save (%ucdbfile%)</command>
33      </execScript>
34  </runnable>
35  <runnable name="dirtest1" type="task">
36  <runnable name="dirtest2" type="task">
37  <runnable name="dirtest3" type="task">
38  <!-- ===== -->
39  <!-- RANDOM TESTS -->
40  <!-- ===== -->
41  <runnable name="random" type="group">
42      <parameters>
43          <parameter name="ucdbfile">../test(%seed%).ucdb</parameter>
44      </parameters>
45  >      <members>
46          <member>randtest1</member>
47          <member>randtest2</member>
48          <member>randtest3</member>
49      </members>
50      <preScript>
51          <command>file delete -force work</command>
52          <command>vlib work</command>
53          <command>vlog -sv (%RMDBDIR%)/src/top.sv</command>      </preScript>
54      <execScript>
55          <command>vsim -lib ../work -sv_seed (%seed%) +SEED=(%seed%) -c
top</command>      <command>run -all</command>
56          <command>coverage attribute -name TESTNAME-value randtest(%seed%) </
command>
57          <command>coverage save (%ucdbfile%)</command>
58      </execScript>
59  </runnable>
60  <runnable name="randtest1" type="task">
61      <parameters>
62          <parameter name="seed">123</parameter>
63      </parameters>
64  </runnable>
65  <runnable name="randtest2" type="task">
66      <parameters>
67          <parameter name="seed">456</parameter>
68      </parameters>
69  </runnable>
```

```
70      <runnable name="randtest3" type="task">
71          <parameters>
72              <parameter name="seed">789</parameter>
73          </parameters>
74      </runnable>
75  </rmdb>
```

Example 2-2 is the design source file for this simple example.

**Figure 2-2. top.sv File**

```
module top;
`ifdef dirstest
initial begin
    $display("Hello from %s", `dirstest);
end
`else
integer seed;
initial begin
    $value$plusargs("SEED=%d", seed);
    $display("Hello from random test, seed is %d", seed);
end
`endif
endmodule
```

## XML Syntax

An XML file is similar in structure to an HTML file. Data or text items are assigned a significance by marking them with tags (short identifiers surrounded in angle brackets). For example,

```
<s>This is a sentence</s>

<s>This is
another sentence</s>
```

Another example is to mark an ill-advised shell command *rm -r* as a command element in an XML file, you type something as follows:

```
<command>rm -r *</command>
```

The tag starting with a slash (/) is called a closing tag and marks the end of the command text. All elements MUST have a closing tag. The opening and closing tags, together with the data they surround, make up what is called an *element* in XML. Elements can be nested to any depth. All XML elements MUST be properly nested within each other. For example,

```
<p><s>This sentence is
in the paragraph</s></p>
```

In this example, properly nested means that since the *<s>* element is opened inside the *<p>* element, it must be closed inside the *<p>* element.

XML tags are case sensitive. The tag `<Name>` is different from the tag `<name>`. Opening and closing tags must be written in the same case. For example,

```
<parameter>This is wrong</Parameter>  
<parameter>This  
is correct</parameter>
```

You can add attributes to opening tags using either of the constructs `<name>="<value>"` or `<name>='<value>'`, where the value is a quoted string, for example:

```
<command ill-advised="yes">rm  
-r *</command>  
  
<command ill-advised='yes'>rm  
-r *</command>
```

In XML, the attribute values MUST always be quoted (as shown in the example above). Either single or double quotes can be used. Attribute names and tag names are limited to the known set understood by the application. Attributes generally hold meta-data (that is, information about the data item rather than the data item itself). However, in some cases the distinction is not obvious.

Whitespace within an element whose contents (data) are used as a string in VRM (such as within `command` or `parameter` elements) can be significant. Whitespace outside such elements is generally insignificant. For example, HTML truncates multiple whitespace characters to one single whitespace and XML does not truncate white spaces.

Write comments in XML as follows:

```
<!-- This is a comment -->
```

Notice the special treatment of the greater than (`>`), less than (`<`), and ampersand (`&`) characters. These characters have specific meaning in an XML file. User data that makes use of these characters (for example, a redirection option on a shell command) must use the entity forms of these characters as shown in [Table 2-1](#):

**Table 2-1. Special Character Substitutions in an XML File**

For	Instead of	Use
less than	<	&lt;
greater than	>	&gt;
ampersand	&	&amp;

Following are the XML naming rules:

- Names can contain letters, numbers, and other characters.
- Names cannot start with a number or punctuation character.
- Name cannot start with the letters `xml`, or `XML`, or `Xml`, and so on.

- Names cannot contain spaces.

# Creating the Basic Structure for an RMDB

The Verification Run Manager DataBase (RMDB) file is the VRM database. The RMDB file contains data for one or more regression test suites arranged in a hierarchical tree topology, which allows the user to group tests and other tasks in various ways.

The *simple/default.rmdb* file is used to explain how to write a RMDB database.

<b>Creating an rmdb Element .....</b>	<b>33</b>
<b>Creating Runnable Elements .....</b>	<b>34</b>
<b>Adding Parameters and Parameter References .....</b>	<b>36</b>
<b>Adding preScript and postScript Actions to a Group .....</b>	<b>37</b>
<b>Adding execScript Actions for the Tests .....</b>	<b>39</b>
<b>Directed Tests.....</b>	<b>40</b>
<b>Pulling it All Together.....</b>	<b>42</b>

## Creating an rmdb Element

The single top-most element in a RMDB database is the *rmdb* element. Create a file with the following top-level *rmdb* element:

```
<!-- See Figure 2-1 on page 28 -->
1 <?xml version="1.0" ?>
2 <rmdb version="1.1">
...
78 </rmdb>
```

- Line 1 — the XML declaration. It defines the XML version as 1.1. Refer to the Appendix “[RMDB Versions](#)” for information on version differences.
- Line 2 — describes the root element of the file. XML documents must contain a root element. This element is the “parent” of the other elements.

In the example code above, the ellipsis (...) represents further details to be added later; insert the ellipsis as a place-holder if you wish, as data outside of the predefined elements is ignored.

In this simple example tutorial, there are six unique tests. Each needs a *Runnable* element of type *task*. Tasks are leaf-level elements in a runnable. Give each of these elements a *name* attribute that matches the name you intend to refer to these tests when grouping or running them. Runnables (*Runnable* elements of type *task*, *group*, or *base*) all share a common namespace and must have a unique name.

## Creating Runnable Elements

Create task runnables for each of the six tests. The *Runnable* for the first random test is as follows:

```
<!-- See Example 2-1 on page 28 -->
63 <Runnable name="randtest1" type="task">
    ...
```

The type is set to *task* because this is a leaf-level node in the regression suite. It is called “task” rather than “test” to emphasize the fact that these *Runnable* elements do not have to relate directly to simulation tests. There can be other steps in the regression process for which a leaf-level “task” is useful.

Tasks are grouped using a *Runnable* element of type *group*. This *simple* example tutorial contains three groups as follows:

- Directed tests
- Random tests
- Top-level group (“nightly”), which runs everything

A group generally contains a *members* element within which are zero or more *member* elements containing, as their text content, the name of a *Runnable* member of that group. For example, the group containing three random tests is as follows:

```
<!-- See Example 2-1 on page 28 -->
42 <Runnable name="random" type="group">
43     <parameters>
44         <parameter name="ucdbfile">../test(%seed%).ucdb</parameter>
45     </parameters>
46     <members>
47         <member>randtest1</member>
48         <member>randtest2</member>
49         <member>randtest3</member>
50     </members>
    ...
```

The *task* and *group* elements are all children of the *rmdb* document element. Their order in the file is insignificant. XML comments (<! -- comment -->) can be added to the database if you wish. After all the groups and tests are defined, the initial RMDB example looks as follows:

```
<!-- See Example 2-1 on page 28 -->
<?xml version="1.0" ?>
2 <rmdb version="1.1">
<!-- we run this group, containing all tests, every night -->
3   <runnable name="nightly" type="group">
4     <members>
5       <member>directed</member>
6       <member>random</member>
7     </members>
...
...
...
</runnable>
<!-- all our directed tests are in this group -->
16   <runnable name="directed" type="group">
17     <parameters>
18       <parameter name="ucdbfile">.../(%INSTANCE%).ucdb</parameter>
19     </parameters>
20     <members>
21       <member>dirtest1</member>
22       <member>dirtest2</member>
23       <member>dirtest3</member>
24     </members>
...
...
...
</runnable>
<!-- all our random tests are in this group -->
44   <runnable name="random" type="group">
45     <parameters>
46       <parameter name="ucdbfile">.../test(%seed%).ucdb</parameter>
47     </parameters>
48     <members>
49       <member>randtest1</member>
50       <member>randtest2</member>
51       <member>randtest3</member>
52     </members>
...
...
...
```

```
<!-- The following code snippet is not from Example 2-1 on page 28. -->
<!-- It is included here for information only. -->
</runnable>
<!-- define all unique tests -->
<runnable type="task" name="dirtest1">
    ...
</runnable>
<runnable type="task" name="dirtest2">
    ...
</runnable>
<runnable type="task" name="dirtest3">
    ...
</runnable>
<runnable type="task" name="randtest1">
    ...
</runnable>
<runnable type="task" name="randtest2">
    ...
</runnable>
<runnable type="task" name="randtest3">
    ...
</runnable>
</rmdb>
```

It is recommended to define the groups in top-down order, and then define the tests.

## Adding Parameters and Parameter References

Parameters can be added to any *runnable* element. A parameter is a name/value pair expressed as an XML element. The *name* of the parameter is specified as an attribute of the *parameter* element, and the contents of that element define the parameter's value.

Parameters (regardless of how many there are) are grouped inside a single *parameters* element within the *runnable* element. In this example, each of the random tests uses one of the three promising seeds identified. The following code has a “seed” parameter added to each of the three random tests:

```
<!-- See Example 2-1 on page 28 -->
69    <runnable name="randtest1" type="task">
70        <parameters>
71            <parameter name="seed">123</parameter>
72        </parameters>
73    </runnable>
74    <runnable name="randtest2" type="task">
75        <parameters>
76            <parameter name="seed">456</parameter>
77        </parameters>
78    </runnable>
79    <runnable name="randtest3" type="task">
80        <parameters>
81            <parameter name="seed">789</parameter>
82        </parameters>
83    </runnable>
```

Each of these tests uses the same basic simulation command, except that a different seed is passed each time. While this is jumping ahead somewhat, the command for running the simulation can be common for all three tests if there is a way to substitute the value of the “seed” parameter into the command at the proper place. This is where parameter references come in.

A parameter reference consists of the name of a parameter surrounded by parentheses and percent signs, for example (%name%). This combination of characters was selected as it is unlikely to occur in typical commands in any commonly known scripting language. When this character combination is used in a command, VRM retrieves the value of the *parameter* element whose name matches the name specified in the parameter reference and substitutes that value in place of the parameter reference string. Therefore, the following command Line 59 as shown in [Example 2-1](#) on page 28:

```
<command>vsim -lib ..\work -sv_seed (%seed%) +SEED=(%seed%) -c top
</command>
```

when run as part of "randtest1" becomes:

```
<command>vsim -lib ..\work -sv_seed 123 +SEED=123 -c top
</command>
```

Note that the *SEED plusarg* is used in this command because this example design wants to print the seed value to the log file. The *vsim* command does not do anything with this *SEED plusarg* other than to pass it to the example design. A real simulation would probably use the seed value in the *-sv\_seed* option instead.

## Adding preScript and postScript Actions to a Group

To compile once and then run multiple tests, place your compile information in a *preScript* because that is what happens once; and you put the simulation in the *execScript* because that is what happens multiple times. If you need to run multiple tests, each with its own compile step, then you have nothing that you need to do only once at the beginning of the run so there is no need for a *preScript*.

When Groups are nested, there can be multiple placeholders where a *preScript* can go but, typically, you only have one compile step that sets things up for all the tests, regardless of grouping. In this case, you can have several Groups without *preScripts* because those Groups do not have anything they need to do once before the tests (or child Groups) in the Group execute. This is acceptable because if a script is empty, then it is skipped (silently).

This *simple* example tutorial has three random tests and share a common work library, since the design only needs to be compiled once. If defined, the *preScript* of the “random” group runs prior to any of the tests in the group. This is a good place to define the compile command needed to initialize the work library used by the random tests. The *preScript* element defines the *preScript* commands. It is a child of the *runnable* element corresponding to the group for which

it is being defined. It contains zero or more *command* elements, each representing a line of the *preScript* script. A typical *preScript* is as follows:

```
<!-- For reference only, not from Example 2-1 on page 28 -->
<preScript launch="exec">
    <command>#!/bin/csh -f</command>
    <command>rm -rf work</command>
    <command>vlib work</command>
    <command>vlog -sv (%RMDBDIR%)/src/top.sv</command>
</preScript>
```

Note that this script is actually a C-shell script. The *launch* attribute on the *preScript* element is set to “exec” to inform VRM that this script is to be executed by an external shell as opposed to being run under *vsim*.

Following is the *preScript* in the *simple* example tutorial:

```
<!-- See Example 2-1 on page 28 -->
53     <preScript>
54         <command>file delete -force work</command>
55         <command>vlib work</command>
56         <command>vlog -sv (%RMDBDIR%)/src/top.sv</command>
57     </preScript>
```

The commands are straight-forward. Observe in the *vlog* command (Line 56) the predefined parameter *RMDBDIR* being used. The value of the *RMDBDIR* parameter contains the directory from which the RMDB database is read. If you looked at the contents of the *examples/vrm/simple* directory, you see that the *src* subdirectory contains the design source (*top.sv* file) that is a sibling of the *default.rmdb* RMDB file.

By using the *RMDBDIR* parameter to locate the design source file, the user can actually execute VRM from any directory, pointing to the RMDB database file by its full path name, and the design source corresponding to that database is easily located. There are several predefined parameters supplied by VRM (see “[Predefined Parameter Usage](#)” on page 266).

After the random tests are run, then merge their UCDB files. To do this, use the *postScript* of the “random” group, as defined by the *postScript* element, as follows:

```
<!-- See Example 2-1 on page 28 -->
64 <postScript mintimeout="60">
65 <!-- a wildcard is acceptable here if merge order is not an issue -->
66 <command>vcover merge ../random.ucdb test123.ucdb test456.ucdb
test789.ucdb</command>
67 </postScript>
<!-- As line 65 states, a wildcard can be used, so Line 66 can become:
<command>vcover merge ../random.ucdb test*.ucdb</command> -->
```

Note that this *postScript* is given extra time to finish by adding the *timeout* attribute. By default, all scripts (*preScript*, *execScript*, and *postScript*) have a 30 second timeout. This can be altered on the command line or on a per-script basis in the RMDB database.

An important point regarding working directories is that by default, VRM runs every task and every group in a separate directory to avoid file name conflicts. The structure of these directories match the calling sequence of the tasks and groups. For example, the *randtest1* task runs in a subdirectory immediately below that of the “random” group that called it. The “random” group runs in a subdirectory immediately below that of the “nightly” group from which it was invoked.

By convention, the tests and groups in this example write their final UCDB files in the parent of their current directory. As later discussed in this tutorial, *randtest1* and the other random tests write their UCDB files into the directory of their parent group. Since the *postScript* of that group runs in the parent directory, the UCDBs created by its child tests are all collected in the directory from which the *postScript* was executed (making them easy to find). The “random” group’s *postScript* finds its input files in the current directory and, likewise, writes its output one directory up. Note that the user can choose to use other file-naming conventions.

## Adding execScript Actions for the Tests

As mentioned before, each of the tests in the “random” group can use the same basic execution script, except that a different seed value is used each time. In order to save a lot of typing, if a leaf-level task does not define an *execScript* element, the chain of calling “parentage” is scanned until an *execScript* element is found (this only happens in the case of an *execScript*). To take advantage of the search algorithm, add the *execScript* element containing the random test execution commands to the “random” *runnable* element as follows:

```
<!-- See Example 2-1 on page 28 -->
44  <runnable name="random" type="group">
45    <parameters>
46      <parameter name="ucdbfile">../test(%seed%).ucdb</parameter>
47    </parameters>
48    <members>
49      <member>randtest1</member>
50      <member>randtest2</member>
51      <member>randtest3</member>
52    </members>
53    <preScript>
54      <command>file delete -force work</command>
55      <command>vlib work</command>
56      <command>vlog -sv (%RMDBDIR%)/src/top.sv</command>
57    </preScript>
58    <execScript>
59      <command>vsim -lib ../work -sv_seed (%seed%) +SEED=(%seed%) -c
top</command>
60      <command>run -all</command>
61      <command>coverage attribute -name TESTNAME -value
randtest (%seed%)</command>
62      <command>coverage save (%ucdbfile%)</command>
63    </execScript>
```

In this case, the commands are executed from within a *vsim* shell (Line 59). The first two commands run the simulation. A unique name is assigned to the *TESTNAME* test record

attribute in the UCDB to identify the specific test in the final merge file (name is based on the *seed* parameter value). Finally, the *coveragesave* command (Line 62) creates the UCDB.

Notice that the *ucdbfile* parameter value (Line 62) is used to supply the name of the UCDB file. This parameter has a special meaning to VRM. If an *execScript* defines a *ucdbfile* parameter, then VRM assumes that the *execScript* represents a simulation. It opens the UCDB file after the *execScript* has been executed and attempts to discover the pass/fail status of the simulation. This pass/fail information is used to prematurely terminate the regression run if there is widespread failure, and it also reports to the user at the conclusion of the regression run.

Note also that the *ucdbfile* parameter value itself contains the (%*seed*%) parameter reference. Parameter references can be nested to any depth (so long as they do not create circular references). The parameter references are resolved at the time the script file is created so, in this case, the value of (%*seed*%) is different for each test, even though the *execScript* element is found in the “random” group rather than in the individual test definitions. This allows for maximum sharing of common information through deep parameterization.

Assuming all three random tests pass, a merge file containing the results of all three tests is created in the *VRMDATA/nightly* directory (the parent of the directory in which the “random” group executes).

## Directed Tests

Since the single design source file contains the stimulus for all three directed tests, enabled by a conditional compile directive, the directed tests can also share a common *execScript* element. However, because of the conditional compile directive, each test must compile its own version of the source in a separate work library. The *preScript* element is left undefined (these

undefined actions are skipped by VRM) and the compile steps are moved to the *execScript* element as follows:

```

<!-- See Example 2-1 on page 28 -->
16   <runnable name="directed" type="group">
17     <parameters>
18       <parameter name="ucdbfile">../(%INSTANCE%).ucdb</parameter>
19     </parameters>
20     <members>
21       <member>dirtest1</member>
22       <member>dirtest2</member>
23       <member>dirtest3</member>
24     </members>
25     <execScript>
26       <command>vlib work</command>
27       <command>vlog -sv +define+dirtest='(%INSTANCE%)' \
(%RMDBDIR%)/src/top.sv</command>
28       <command>vsim -c top</command>
29       <command>run -all</command>
30       <command>coverage attribute -name TESTNAME -value \
(%INSTANCE%)</command>
31       <command>coverage save (%ucdbfile%)</command>
32     </execScript>
33     <postScript>
34 <!-- a wildcard is acceptable here if merge order is not an issue -->
35       <command>vcover merge ../directed.ucdb dirtest1.ucdb \
dirtest2.ucdb dirtest3.ucdb</command>
36     </postScript>
37   </runnable>
<!-- As line 34 states, a wildcard can be used, so Line 35 can become:
<command>vcover merge ../directed.ucdb dirtest*.ucdb</command> -->
```

Note that on the compile command, the *INSTANCE* parameter is used. This is another predefined parameter whose value is the name of the group or task currently executing. In this case, it is either *dirtest1*, *dirtest2*, or *dirtest3*. The *INSTANCE* parameter is also used to give the test a unique name in the UCDB. The *ucdbfile* parameter is used here in the same way as in the random tests. Finally, a *postScript* element defines the command that merges the coverage from these three tests into a *directed.ucdb* file in the *VRMDATA/nightly* directory.

Since the leaf-level tasks do not, by themselves, provide any additional information besides their name, the *runnable* elements corresponding to those tasks have no contents. For example,

```

<!-- See Example 2-1 on page 28 -->
38   <runnable name="dirtest1" type="task"/>
39   <runnable name="dirtest2" type="task"/>
40   <runnable name="dirtest3" type="task"/>
```

## Pulling it All Together

In order to allow the entire regression suite to be run as a single entity, a top-level “nightly” group is defined. The example “nightly” group is as follows:

```
<!-- See Example 2-1 on page 28 -->
3  <runnable name="nightly" type="group">
4    <members>
5      <member>directed</member>
6      <member>random</member>
7    </members>
8    <postScript launch="vsim">
9      <command>vcover merge merge.ucdb directed.ucdb \
10        random.ucdb</command>
11      <command>vcover attr -name ORIGFILENAME \
12        -name TESTSTATUS merge.ucdb</command>
11    </postScript>
12  </runnable>
```

The only addition besides the list of members is a *postScript* element that merges the two intermediate merge files into a single UCDB. A second command is added to provide some kind of indication that the process actually worked.

After creating the *default.rmdb* file, run the *simple* example tutorial.

## Run the Example Tutorial in GUI Mode

Following are the steps to run this *simple* example tutorial:

1. Copy the contents of the *examples/vrm/simple* directory to an empty directory:

```
cp -r <install_dir>/examples/vrm/simple
```

2. Change directory to *vrm/simple*:

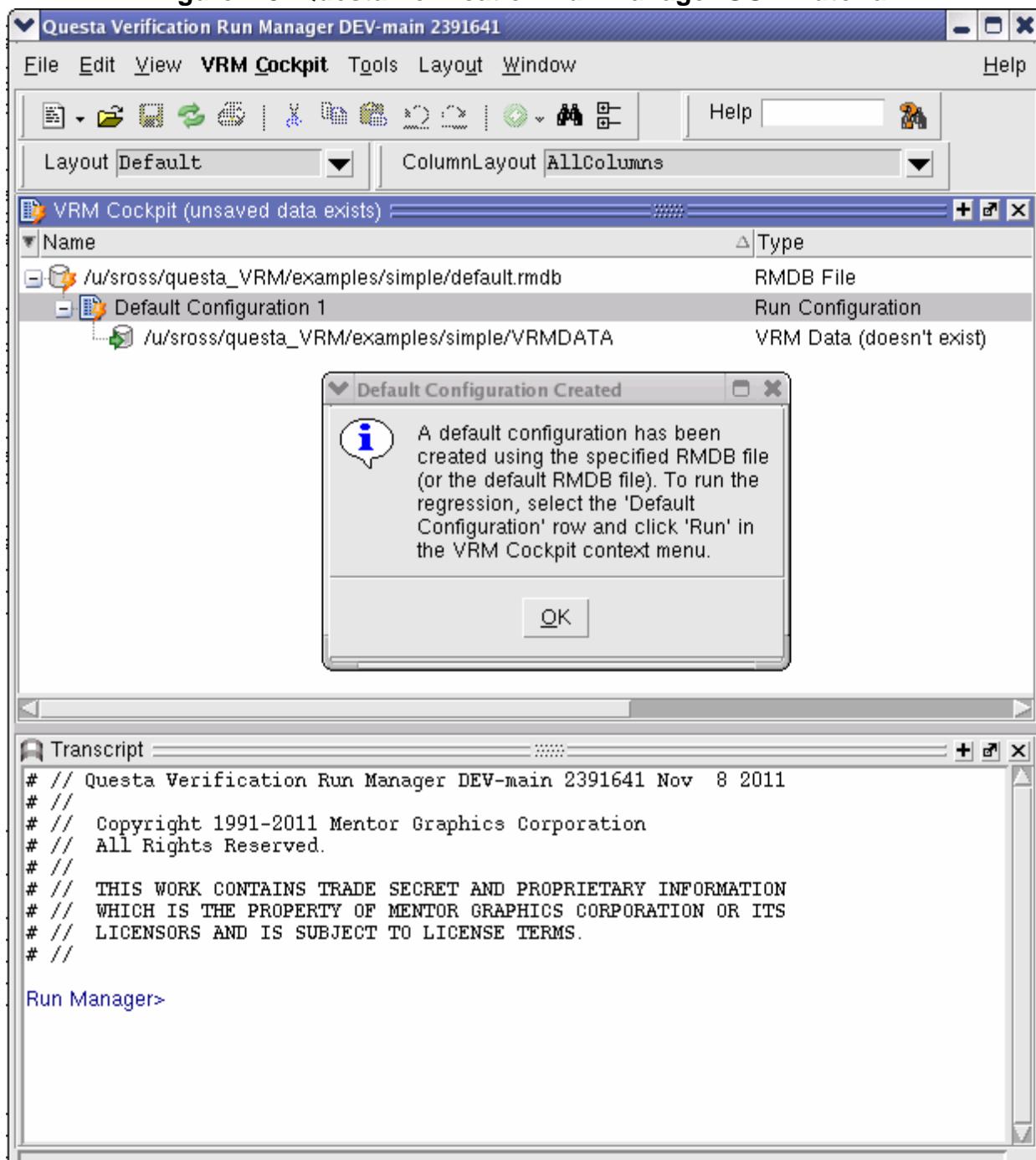
```
cd simple
```

3. Execute the command to run the *default.rmdb* database as follows:

```
vrun -gui
```

This invokes the **Questa Verification Run Manager GUI** as shown in [Figure 2-3](#).

Figure 2-3. Questa Verification Run Manager GUI - Tutorial



The VRM GUI invokes with the **Default Configuration Created**, **VRM Cockpit**, and the **Transcript** windows as shown in Figure 2-3. In the **Default Configuration Created** window, do the following:

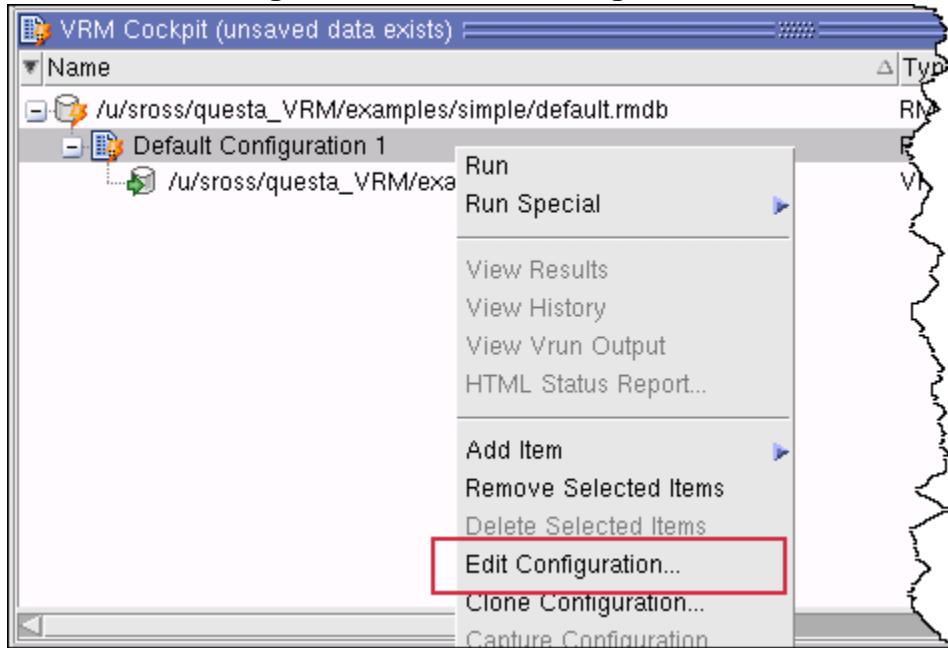
**select -> OK**

In the **VRM Cockpit** window, do the following as shown in [Figure 2-4](#):

**select -> Default Configuration**  
1

**right-click -> select -> Edit Configuration...**

**Figure 2-4. Default Configuration**



This invokes the **Edit VRM Configuration** window.

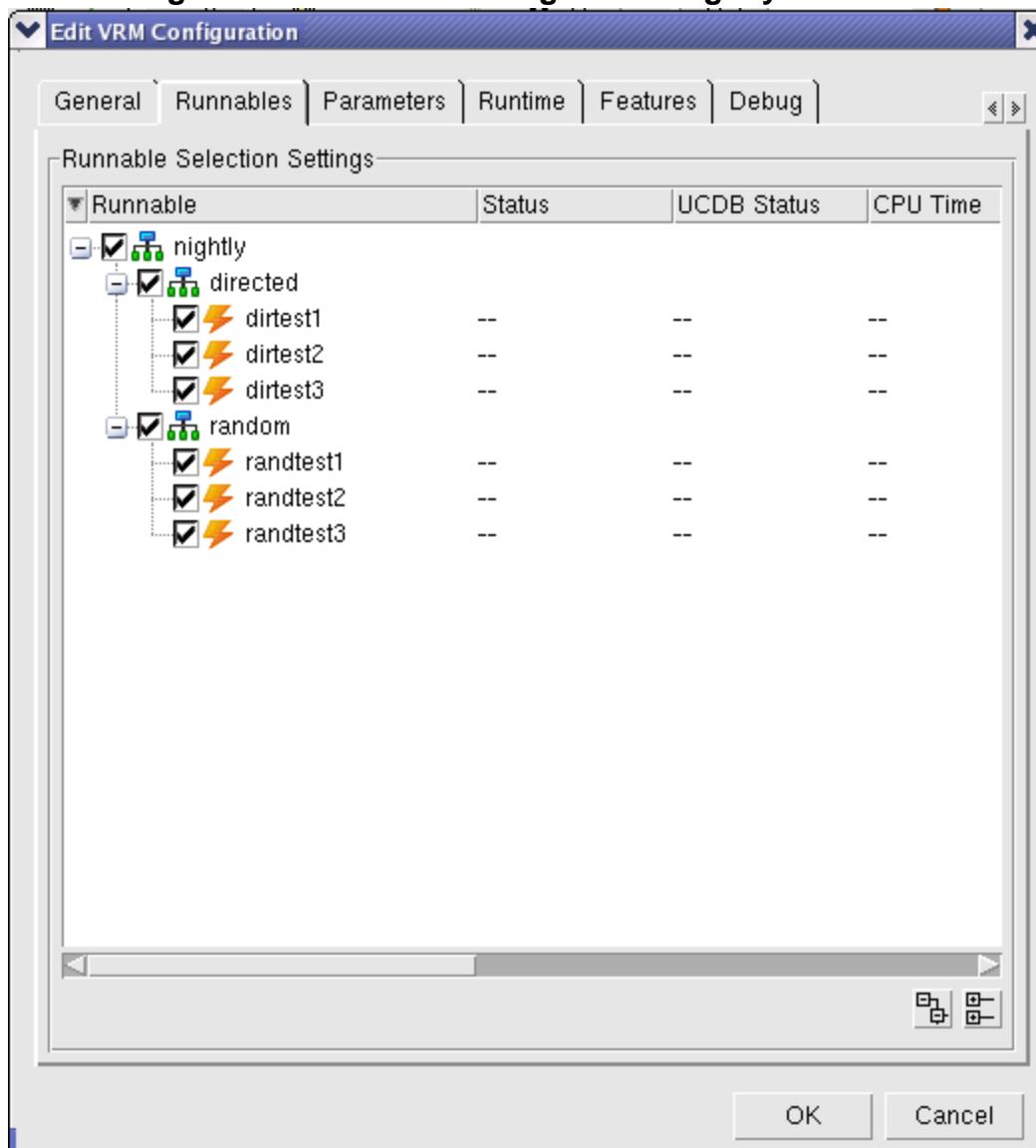
**select -> Runnables Tab**

[Figure 2-5](#) shown the **Edit VRM Configuration Runnable** tab.

**select -> nightly**

The check boxes display the Runnables that will be run.

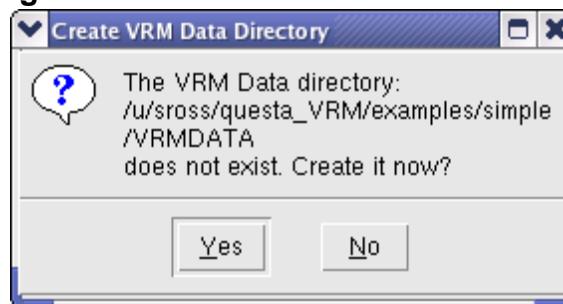
**Figure 2-5. Edit VRM Configuration nightly Selected**



**select -> OK**

This invokes the **Create VRM Data Directory** window as shown in Figure 4-3.

**Figure 2-6. Create a VRM Data Directory**

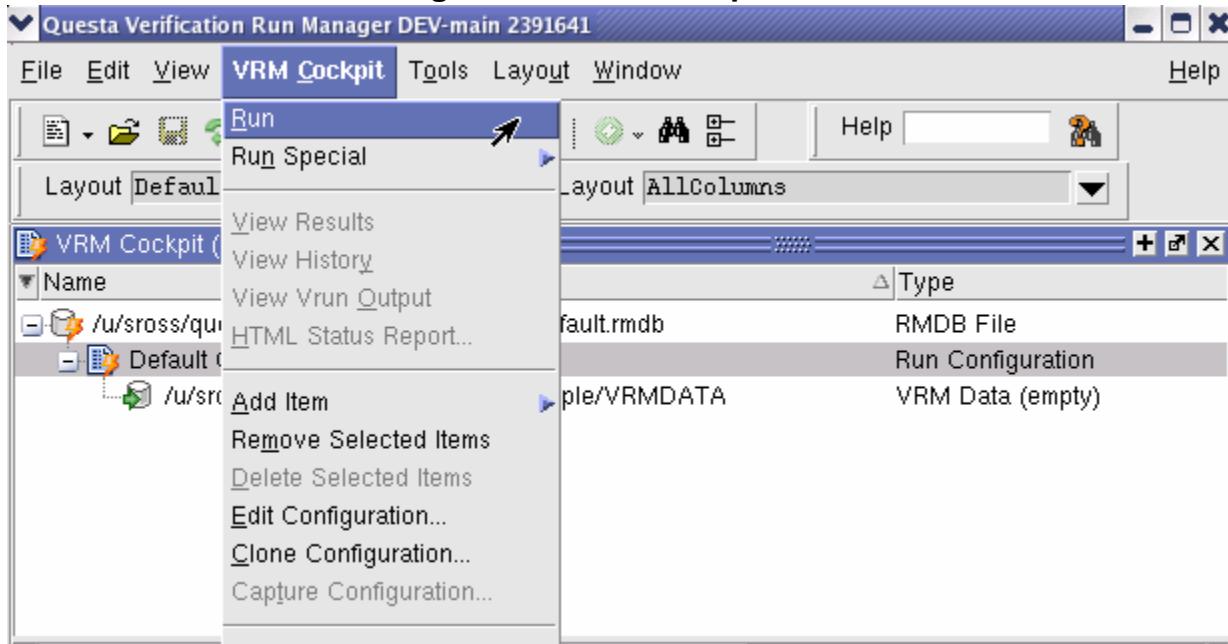


**select -> Yes**

To initiate the regression run, do the following as shown in [Figure 2-7](#).

**select -> VRM Cockpit -> Run**

**Figure 2-7. VRM Cockpit Run**



To initiate the regression run, do the following:

**select -> VRM Cockpit -> Run**

[Figure 2-8](#) shows the **VRM Results** window with the regression run in progress.

Below the Action status table is a row of counters with an animated “running light” on the left-most side of the row (see [Figure 2-8](#)). The running light indicates the state of the overall regression run (such as Running, Queued, or Passed). The counters indicate how many Actions exist in each of the listed categories as follows:

- The **Pending** category includes tests that are not yet eligible to launch (they are waiting for something else to finish. For example,

### Pending:3

indicates that there are three jobs pending.

- The **Queued** category includes tests that are eligible and have been launched, but have not yet reported that they are running (these tests are probably in the grid queue).
- The **Running** category indicates the tests that are running. The running tests are listed in the **Status** column.

- Several other counters (Suspended, Passed, Failed, Timeout, Killed, Skipped, and Dropped) identify other potential states.
- The **Coverage** category provides information about coverage collected for the tests.
- The **Testplan** category provides information about any related testplans for the tests.

**Figure 2-8. Regression Run in Progress**

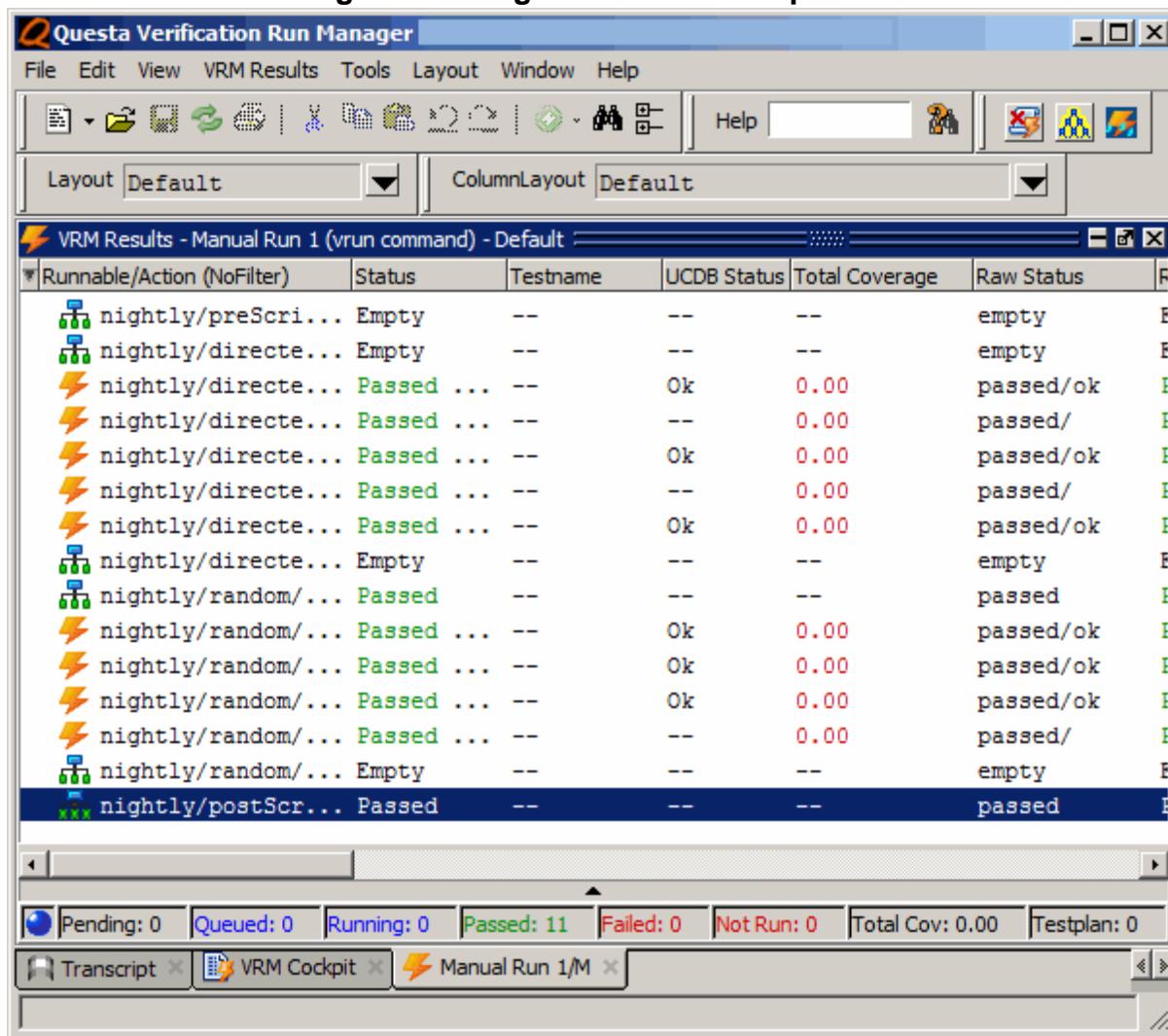
The screenshot shows the Questa Verification Run Manager (VRM) interface. The main window title is "Questa Verification Run Manager". The menu bar includes File, Edit, View, VRM Results, Tools, Layout, Window, and Help. The toolbar contains various icons for file operations like Open, Save, Print, and a search function. Below the toolbar are dropdown menus for "Layout Default" and "ColumnLayout Default". The central area displays a table titled "VRM Results - Manual Run 1 (vrun command) - Default". The table has columns: Runnable/Action (NoFilter), Status, Testname, UCDB Status, Total Coverage, and Raw Status. The data in the table is as follows:

Runnable/Action (NoFilter)	Status	Testname	UCDB Status	Total Coverage	Raw Status
nightly/preScri...	Empty	--	--	--	empty
nightly/directe...	Empty	--	--	--	empty
nightly/directe...	Passed	---	Ok	0.00	passed/ok
nightly/directe...	Running	--	--	--	running
nightly/directe...	Running	--	--	--	running
nightly/directe...	Pending	--	--	--	pending
nightly/random/...	Passed	--	--	--	passed
nightly/random/...	Running	--	--	--	running
nightly/random/...	Running	--	--	--	running
nightly/random/...	Running	--	--	--	running
nightly/random/...	Pending	--	--	--	pending
nightly/postScr...	Pending	--	--	--	pending

Below the table is a summary bar with the following counts: Pending: 3, Queued: 0, Running: 5, Passed: 2, Failed: 0, Not Run: 0, Coverage: 0, and Testplan: 0. At the bottom, there is a navigation bar with tabs: Transcript, VRM Cockpit, and Manual Run 1/M.

Figure 2-9 shows the regression run completed.

**Figure 2-9. Regression Run Completed**



When the regressions finish running, the results can be observed in the **Status** column of the **VRM Results** window as shown in [Figure 2-9](#). Observe that all *nightly/directed/dirtest\** and *nightly/random/randtest\** tests passed.

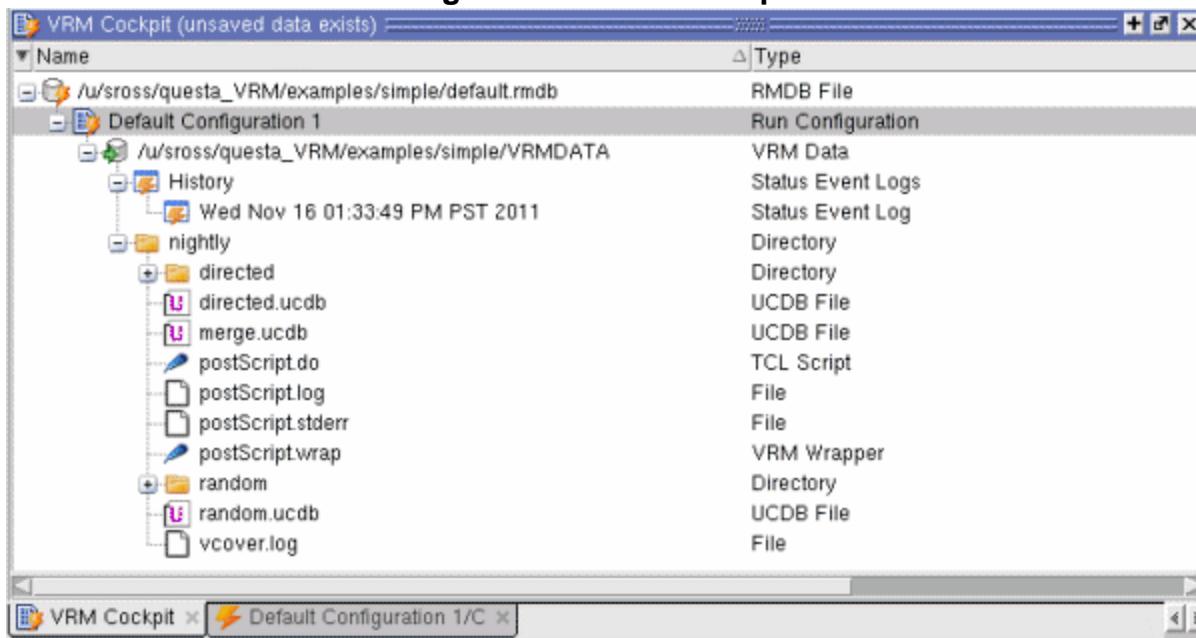
The top-level *simple* directory now contains the automatically generated *transcript* file and *VRMDATA* directory.

The *VRMDATA* directory contains the *History* directory and *nightly* directory as shown in [Figure 2-10](#).

The *logs* directory contains data files pertaining to the status of the regression runs launched in that directory.

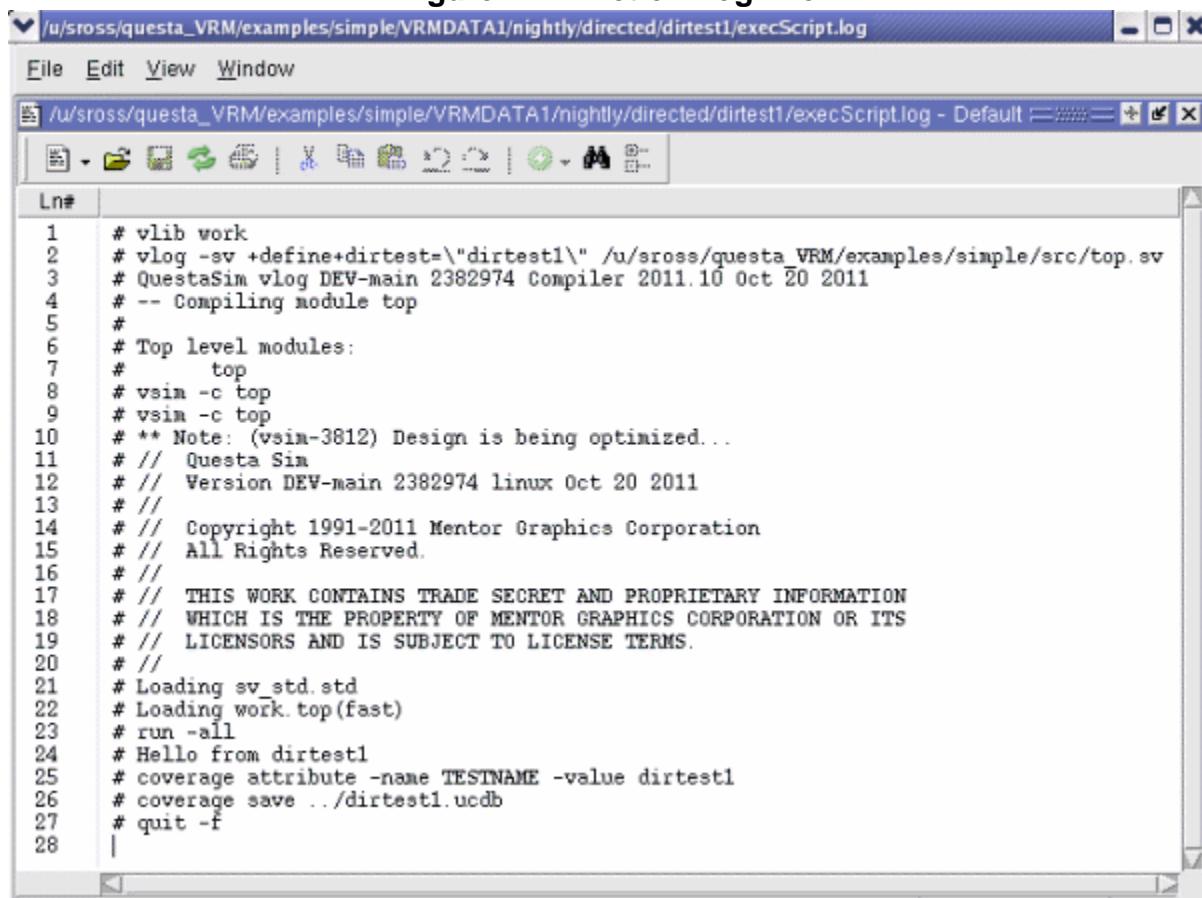
The *nightly* directory contains the *directed* directory and *random* directory, and many files.

Figure 2-10. VRM Cockpit



These automatically generated output files can be viewed using the GUI. In the **VRM Results** window select the Action whose output you wish to observe and right-click on the action and select **View > Action Log File**

This displays the *execScript.log* file as shown in [Figure 2-11](#).

**Figure 2-11. Action Log File**


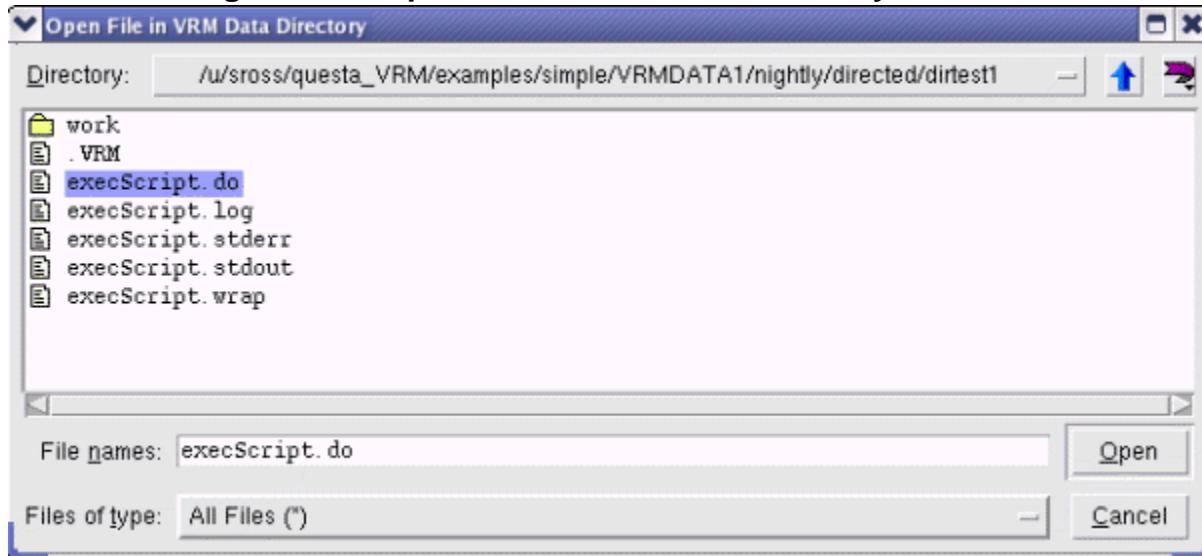
The screenshot shows a window titled 'execScript.log' with the path '/u/sross/questa\_VRM/examples/simple/VRMDATA1/nightly/directed/dirtest1/execScript.log'. The window contains a code editor with the following content:

```

Ln#
1 # vlib work
2 # vlog -sv +define+dirtest1\\" /u/sross/questa_VRM/examples/simple/src/top.sv
3 # QuestaSim vlog DEV-main 2382974 Compiler 2011.10 Oct 20 2011
4 # -- Compiling module top
5 #
6 # Top level modules:
7 #   top
8 # vsim -c top
9 # vsim -c top
10 # ** Note: (vsim-3812) Design is being optimized...
11 # // Questa Sim
12 # // Version DEV-main 2382974 linux Oct 20 2011
13 # //
14 # // Copyright 1991-2011 Mentor Graphics Corporation
15 # // All Rights Reserved.
16 # //
17 # // THIS WORK CONTAINS TRADE SECRET AND PROPRIETARY INFORMATION
18 # // WHICH IS THE PROPERTY OF MENTOR GRAPHICS CORPORATION OR ITS
19 # // LICENSORS AND IS SUBJECT TO LICENSE TERMS.
20 #
21 # Loading sv_std.std
22 # Loading work.top(fast)
23 # run -all
24 # Hello from dirtest1
25 # coverage attribute -name TESTNAME -value dirtest1
26 # coverage save ../dirtest1.ucdb
27 # quit -f
28 |

```

You can view other files by right-clicking an action and selecting View > File in VRM Data, which opens the **Open File in VRM Data Directory** window, as shown in [Figure 2-12](#).

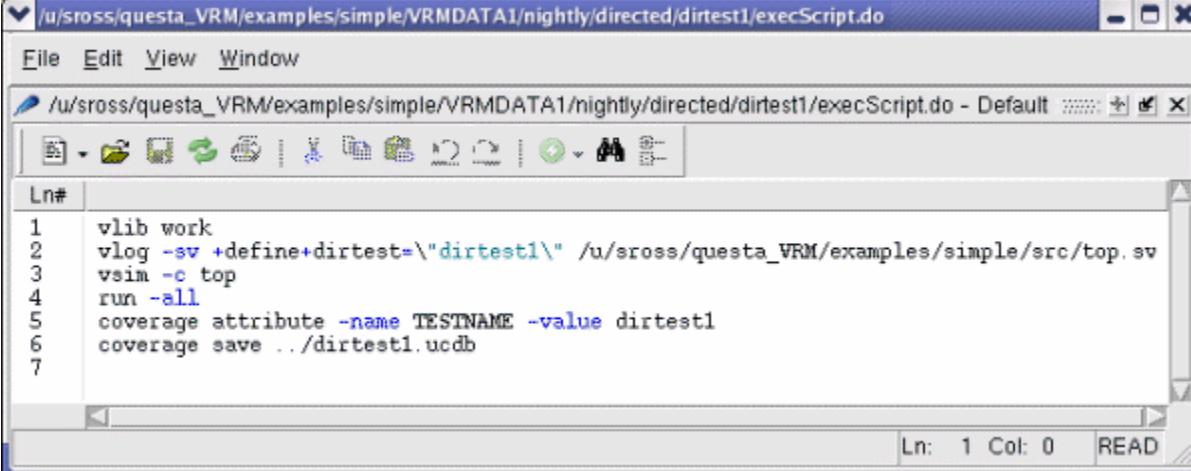
**Figure 2-12. Open Files in VRM Data Directory Window**

**select -> file**

**select -> Open**

The file that you select opens in the GUI window as shown in Figure 2-13. For this example tutorial, the *execScript.do* file was selected.

**Figure 2-13. execScript.do File**



A screenshot of a terminal window titled "/u/sross/questa\_VRM/examples/simple/VRMDATA1/nightly/directed/dirtest1/execScript.do". The window shows the following script:

```
1 vlib work
2 vlog -sv +define+dirtest=\\\"dirtest1\\\" /u/sross/questa_VRM/examples/simple/src/top.sv
3 vsim -c top
4 run -all
5 coverage attribute -name TESTNAME -value dirtest1
6 coverage save ../dirtest1.ucdb
7
```

The window has a toolbar with various icons at the top, and a status bar at the bottom indicating "Ln: 1 Col: 0 READ".

# Run in Batch Mode and Debug Regression Tests

---

After you have created a VRM database you can execute the regressions and view the runtime files to investigate the results of the regression.

See “[Creating the Basic Structure for an RMDB](#)” on page 33.

<b>Executing VRM .....</b>	<b>52</b>
<b>Finding the Runtime Files.....</b>	<b>52</b>
<b>Failure to Recognize an Action as a Simulation .....</b>	<b>53</b>

## Executing VRM

To execute this *simple* example tutorial, make sure the RMDB database is named *default.rmdb* and is stored in the current working directory (default place for VRM to look for its database) and that the *src* directory containing the design source *top.sv* file is also in the current working directory. Make sure Questa version 10.1 or later is in the search path and execute the following command:

**vrun nightly**

Alternately, you can (from any directory you have write permission) choose to execute the following command:

**vrun -rmdb \$MTI\_HOME/examples/vrm/simple/default.rmdb nightly**

This command points VRM directly at the RMDB file provided with the release. Since the compile command uses the *VRUNDIR* parameter to locate the design source, this command works just as well as the first command.

## Finding the Runtime Files

VRM creates a *VRMDATA* directory in the current working directory. This is the root of a tree of working directories created to run the various tasks specified in the regression suite. Beneath this directory is a single *nightly* directory in which the “nightly” group is run. Beneath this are the *directed* and *random* directories and beneath each of these directories are subdirectories named for each leaf-level test. For example, the directory *VRMDATA/nightly/random/randtest1* should contain the following files:

```
-rwxr-xr-x 1 name      group      140 Sep 26 15:43
execScript.do*-rw-rw-r-- 1 name      group     1027 Sep 26 15:43
execScript.log-rw-rw-r-- 1 name      group       0 Sep 26 15:43
execScript.stderr-rw-rw-r-- 1 name      group    1747 Sep 26 15:43
execScript.stdout-rw-r--r-- 1 name      group    1588 Sep 26 15:43
execScript.wrap
```

The *execScript.do* file contains the parameter-expanded version of the commands found in the *execScript* element under the *randtest1* runnable. The *execScript.wrap* file contains the *vsim* wrapper used to launch the script. The *execScript.log* file contains the log output of the *execScript* script.

The *execScript.stdout* and *execScript.stderr* files contain the standard output and standard error output of the wrapper script, respectively. These two files can generally be ignored.

More files can be created depending on the design and the execution method, but these are the basic files generated for each action. The group directories can have a set of *preScript* and/or *postScript* files, depending on which of these actions is defined.

The final merge result is placed in the VRMDATA/*nightly* directory under the name *merge.ucdb*.

## Failure to Recognize an Action as a Simulation

If an *execScript* action represents a simulation but the pass/fail analysis seems to not recognize the action as a simulation or returns a strange error, check the following:

- Make sure the *ucdbfile* parameter points to the UCDB file (relative paths start from the working directory if the leaf-level task is under consideration).
- Make sure the *execScript* saves off a UCDB file (with the *coverage save* command).
- Check that the UCDB has not been corrupted.
- Load the UCDB into the UCDB Browser and check that the *TESTSTATUS* attribute has a valid value.
- If all else fails, notify Mentor Graphics Customer Support.

## A More Complete Example

The following is an example of the RMDB database for a regression suite set up to run some number of known seed simulations and some number of random simulations. Two macros are defined as follows:

- Macro “all” runs four simulations with known seeds and four random simulations.
- Macro “rand” runs ten random simulations.

If this example file was called *default.rmdb*, then the user can execute either of these macros by typing nothing more than *vrun all* or *vrun rand* from within the directory where the RMDB database file is stored.

Note that the Runnables *all* and *rand* exist as stand-alone Task Runnables and are not part of the regression suite. A single RMDB database can contain any number of independent trees of related Runnables as well as stand-alone Tasks that are not part of any such tree (as is the case

## Example Tutorial

### A More Complete Example

---

for this example's two macro Runnables). The user has the flexibility to create a single-task regression that recursively calls the VRM executable to fire off another regression suite in the same database with a predetermined set of command-line options.

```
<rmdb>
    <runnable name="all" type="task">
        <execScript>
            <command>vrun -rmdb (%RMDBFILE%) -vrldata (%DATADIR%)
                -Gallseeds=2,4,6,8 -Grandcount=4 nightly</command>
        </execScript>
    </runnable>
    <runnable name="rand" type="task">
        <execScript>
            <command>vrun -rmdb (%RMDBFILE%) -vrldata (%DATADIR%)
                -Grandcount=10 nightly</command>
        </execScript>
    </runnable>
    <runnable name="nightly" type="group">
        <parameters>
            <parameter name="allseeds"/>
            <parameter name="randcount"/>
            <parameter name="ucdbfile">test (%seed%) .ucdb</parameter>
        </parameters>
        <members>
            <member>goodseeds</member>
            <member>randseeds</member>
        </members>
        <preScript launch="exec">
            <command>rm -rf work</command>
            <command>vlib work</command>
            <command>vlog -sv +cover=bces (%VRUNDIR%)/src/top.sv</command>
        </preScript>
        <execScript>
            <command>vsim -c top -coverage -lib ../work -sv_seed
                (%seed%)</command>
            <command>run -all</command>
            <command>coverage attribute -name TESTNAME -value
                test (%seed%)</command>
            <command>coverage save (%ucdbfile%)</command>
        </execScript>
    </runnable>
    <runnable name="goodseeds" type="task" unless="(%allseeds%) eq {}"
        foreach="(%allseeds%)">
        <parameters>
            <parameter name="seed">(%ITERATION%)</parameter>
        </parameters>
    </runnable>
    <runnable name="randseeds" type="task" unless="(%randcount%) eq {}"
        repeat="(%randcount%)">
        <parameters>
            <parameter name="seed">random</parameter>
        </parameters>
    </runnable>
</rmdb>
```

Note that the macro options are setup to use the same RMDB database as that from which the macro is read and to use the same *VRM Data* directory as supplied to the original *vrun* invocation. This allows the user to control these aspects of the actual regression run even while invoking one of the macro Tasks. Also, empty values are defined for the *allseeds* and *randcount* parameters and these values are tested in the leaf-level *Runnable* elements. This allows the macro to invoke either or both kinds of simulation (known-seed and/or random) as needed. If the key parameter value for either of the leaf-level Tasks is not specified, then that Task is not executed. More complex combinations can easily be created using the same principles.

Note that since this macro functionality uses the same infrastructure as any other Runnable, these macro-like Task Runnables can also be grouped and parameterized, taking full advantage of base and group inheritance. However, since these macro Tasks execute the actual regression suites by invoking *vrun* recursively, as opposed to using the Group membership paradigm, parameters defined in a Runnable at the macro-level cannot be referred to from the suite-level Runnables unless they are passed through on the *vrun* command line (as in the *allseeds* and *randcount* parameters in the example).



# Chapter 3

## Basic Operations

---

This chapter describes basic operation aspects of the VRM.

<b>Modes of Operation</b> .....	<b>58</b>
<b>RMDB Database File</b> .....	<b>60</b>
<b>Status Reports</b> .....	<b>68</b>
<b>Managing the VRMDATA Directory</b> .....	<b>87</b>
<b>Limit Concurrently Running Processes</b> .....	<b>92</b>
<b>Override Repeat Counts on Runnable Nodes</b> .....	<b>92</b>
<b>Integration with Compute Grid Systems</b> .....	<b>94</b>
<b>Specify Selected Runnables</b> .....	<b>98</b>
<b>Batch Mode Usability Functions</b> .....	<b>110</b>
<b>VRM Email Functionality</b> .....	<b>123</b>
<b>Regression Run Modification</b> .....	<b>130</b>
<b>Override Parameter Values from Command Line</b> .....	<b>132</b>
<b>Access RMDB API via the vrun Command Line</b> .....	<b>138</b>

## Modes of Operation

VRM has several major modes of operation: execution, control/abort, status, and notification. The mode is determined by the presence of command-line options.

These modes are described in the subsections below.

**Table 3-1. Modes of Operation**

Mode	Main Option	Description
Execution	-run (default)	Execute the Runnable items (tests/groups) specified on the command line.
Control/Abort	-kill	Abort a currently running <i>vrun</i> session.
Status	-status	Generate status report on a previous or current <i>vrun</i> session.
Notification	-send <event>	Inform VRM daemon of the completion of a scheduled Action/job.

<b>Execution Mode .....</b>	<b>58</b>
<b>Control/Abort Mode .....</b>	<b>58</b>
<b>Status Mode .....</b>	<b>59</b>
<b>Notification Mode .....</b>	<b>59</b>

## Execution Mode

Execution mode is used to launch one or more regression Actions, as defined in the Verification Run Manager DataBase (RMDB) file. The RMDB file is the VRM database. The RMDB file generally contains data for one or more regression test suites arranged in a hierarchical tree topology, which allows the user to group tests and other tasks in various ways.

Execution mode is the default mode unless the *-send* option (Notification Mode, see [page 59](#)) is specified.

Refer to the “[vrun Command](#)” on page 438 for information on the execution mode options.

## Control/Abort Mode

The control/abort mode allows the user to abort a currently running *vrun* process. The *-kill* option causes *vrun* to attempt to connect to the running process via its TCP/IP control port. If successful, it sends the listening process the *kill* command, which causes the regression run to terminate.

The *port@host* setting is derived from the event log of the running process; therefore, the *VRM Data* directory setting for the sending and listening process must be the same.

Refer to “[vrun Command](#)” on page 438 for information on the control/abort mode options.

## Status Mode

Status mode causes *vrun* to emit a status report on a previously completed or currently running regression run. The *-status* option causes *vrun* to read one or more event log(s), previously saved to the logs subdirectory of the designated *VRM Data* directory, and format a status report on *stdout* summarizing the status of each Action executed in that *VRMDATA* directory.

By default, only the status of the immediately previous (or current) run is shown, although a cumulative report of all activity in the *VRMDATA* directory is also available.

Refer to the “[vrun Command](#)” on page 438 for information on the status mode options.

## Notification Mode

Notification mode is used to communicate changes in job status to a master VRM process, and notify an existing VRM instance that a specific Action has completed. The user generally does not need to deal with this mode, as the notification mode command is called from the same wrapper script used to launch the job in question.

Notification mode is enabled by the *-send* command-line option that must be supplied one of the event values: *start*, *user*, *done*.

Refer to the “[vrun Command](#)” on page 438 for information on the notification mode.

## RMDB Database File

---

The Verification Run Manager DataBase (RMDB) file is the VRM database.

A RMDB file typically contains a single *rmdb* element that describes the regression. A *rmdb* file can include another *rmdb* file containing a well formed *rmdb* element.

[Figure 3-1](#) is a RMDB file example.

**Figure 3-1. RMDB File Example**

```
myregrxn.rmdb:
<rmdb>
    <runnable name="mygroup" type="group">
        <members>
            <member>test1</member>
            <member>test2</member>
            <member>test3</member>
        </members>

        <preScript launch="exec" file="mypre.sh"/>
        <postScript launch="exec" file="mypad.sh"/>
    </runnable>
    <!-- each task runnable is defined in the rmdb database -->

    <runnable name="test1" type="task">
        <execScript launch="exec" file="test1.sh"/>
    </runnable>
    <runnable name="test2" type="task">
        <execScript launch="exec" file="test2.sh"/>
    </runnable>
    <runnable name="test3" type="task">
        <execScript launch="exec" file="test3.sh"/>
    </runnable>
</rmdb>
```

This runnable is a group. It has the attribute type="group"

Group runnables have members. These members, denoted by the members element, can be tasks or other groups.

These runnables are tasks. They have the attribute type="task". They are members of the group "mygroup".

Refer to “[Creating the Basic Structure for an RMDB](#)” on page 33 for instructions on developing a *default.rmdb* file.

<b>Comments in the RMDB File . . . . .</b>	<b>61</b>
<b>Embedding Options in the RMDB File . . . . .</b>	<b>62</b>
<b>Dry Run of the RMDB Database . . . . .</b>	<b>62</b>
<b>Locating the RMDB Database . . . . .</b>	<b>63</b>
<b>Validation of the RMDB Database . . . . .</b>	<b>63</b>
<b>Split RMDB Database Among Several Files . . . . .</b>	<b>64</b>
<b>Set the Search Path for Shell Scripts . . . . .</b>	<b>65</b>
<b>Locating Script and Log Files . . . . .</b>	<b>66</b>

## Comments in the RMDB File

Certain input files can contain user comments. The RMDB database file is written using XML and accepts comments according to the XML standard. For example,

```
<rmdb>
    <!-- This is an XML comment -->
    ...
</rmdb>
```

Note that the double-hyphen character sequence (--) is not allowed within an XML comment. Comments can also span multiple lines. For example,

```
<rmdb>
    <!-- This is
        a multiline comment -->
    ...
</rmdb>
```

See “[XML Syntax](#)” on page 30 for additional XML information. Also, refer to the eXtendable Markup Language (XML) Standard for more details.

Text input files can contain comments of various types. These files are as follows:

- The *runlist* file is specified by the *-runlist* command-line option.
- The *tasklist* file is specified by the *-tasklist* command-line option.
- The *.nodelete* file is used internally by the auto-deletion functionality.

A single text input file can contain comments of any the following types:

- C-style comments, delimited by /\* and \*/.
- C++-style comments, delimited by // and a newline/EOF.
- TCL-style comments, delimited by # and a newline/EOF.

Once the comments are stripped from a text input file, the file is divided into “lines” (delimited by newline characters), any leading and/or trailing whitespace is removed, and blank lines are eliminated. The *GetFileLines* procedure (see [page 555](#)) can also be used from within a user-definable procedure to open a text input file, strip the comments, and parse the file into lines as described above. The procedure returns a list where each element corresponds to a single non-blank line in the file. The text input file format is perfect for storing lists of Actions, randomization seeds, UCDB files, or any other file that consists of a list of strings.

Note that C-style comments are newline-sensitive. That is, if a C-style comment spans characters within a single line:

```
This is /* comment /* a single line.
```

Then the output will also consist of a single line as follows:

```
This is a single line.
```

However, if the C-style comment spans multiple lines like the following:

```
These /* multi-line
comment */ tokens are on separate lines.
```

Then the output also consists of multiple lines as follows:

```
These
tokens are on separate lines.
```

## Embedding Options in the RMDB File

In some cases, an RMDB file is configured with the assumption that one or more command-line options are always used whenever *vrun* uses that particular database. In this case, these precanned options can be embedded in the RMDB file by adding an *options* attribute to the document (*rmdb*) element. For example,

```
<rmdb options="-j 1 -verbose">
  ...
</rmdb>
```

These options are parsed in the same way as on the command line—with tokens separated by whitespace, and single/double quotes used to delimit a single token irrespective of whitespace. These options are processed after the RMDB file is opened but before the actual run begins. The options are not processed in the non-running modes (that is, if the *-send*, *-status*, or *-kill* options are specified), as in those cases no RMDB file is opened.

The *-rmdb* option, which changes the database file to be used, should not be embedded in the RMDB file. Also, the options that control the behavior mode of the *vrun* process (such as *-send*, *-status*, *-kill*, and so on) are ignored if embedded in the RMDB file.

## Dry Run of the RMDB Database

VRM supports a “dry run” through a given set of selected Runnables where the scripts and other supporting files and directories in the VRMDATA area are generated but the scripts are not launched.

The *vrun* option to initiate a dry run is *-noexec* (see “[vrun Command](#)” on page 438). If this option is specified on the *vrun* command line, then the utility goes through all the motions necessary to prepare the specified Actions to run, however:

- The actual launch is suppressed.
- The event log is not generated.

- The listening socket is not opened.

There are two *vrun* options used to generate additional debug information: *-showcmds* and *-showparams*. The *-showcmds* option causes the actual commands for each Action to be emitted to the log output. The *-showparams* option causes the keyword and resulting value of each parameter expansion to be emitted to the log output.

The *-showparams* option also activates the *-showcmds* and both options also activate the *-noexec* option.

There is no switch for executing the Actions in question once the script files have been generated.

## Locating the RMDB Database

By default, VRM looks for its database in the *default.rmdb* file in the current directory. If the *MTI\_VRUN\_DB* environment variable is set, then the contents of that variable are read and used as the (relative or absolute) path to the RMDB database file.

If the *-rmdb <rmdbfile>* option is specified on the command line, then the file/path specified by that option is used as the path to the RMDB database file. If a relative path is provided, it is assumed to be relative to the directory from which VRM was invoked. The command-line option overrides the *MTI\_VRUN\_DB* environment variable. However, in both cases the override is all or nothing.

If the *-rmdb <rmdbfile>* option is specified on the command line but the file pointed to does not exist or cannot be read, then an error occurs regardless of whether or not the *MTI\_VRUN\_DB* environment variable or the default path (*./default.rmdb*) refers to a legitimate RMDB database file.

## Validation of the RMDB Database

The VRM database is an XML file. By default, the contents of an XML file are validated against the Document Type Definition (DTD) file located in the Questa release tree at *vm\_src/rmdb.dtd*. The XML file is validated against the DTD provided with the release from which the *vrun* command is launched. Validation occurs **after** the XML file has been parsed and loaded. If there are validation errors in the XML file, they are printed by the parser library (to *stdout*), an error message is emitted from VRM, and the *vrun* process exits.

The RMDB file allows for the inclusion of user-defined elements and/or attributes in the database. In order to bypass validation (either for performance reasons or to allow an RMDB file containing one or more nonstandard constructs to be used), include the *-nodtdvalidate* options on the command line (see “[vrun Command](#)” on page 438).

Validation is separate from syntactic parsing. An XLM document is “well-formed” if the opening and closing tags of each element match and the attributes are properly formed.

Validation checks that the correct elements are used in the right places and the attributes on each element are valid for that element.

Validation does not verify that the contents of the elements (commands, parameter values, and so on) accomplish the purpose they intended to accomplish.

## Split RMDB Database Among Several Files

The VRM database supports XInclude (mechanism for merging XML documents), allowing the database contents to be divided among multiple files. The “parent” file will still contain the document (*rmdb*) element and also specify one or more *xi:include* elements in the XInclude namespace. In that case, the file referred to by the *xinclude* element is read into the VRM database in place of the *xinclude* element.

**Figure 3-2. XInclude Allows Database Contents Divided Among Multiple Files**

For example, assume the *parent* file contains the following:

```
<rmdb xmlns:xi="http://www.w3.org/2003/XInclude">
  <runnable name="nightly" type="group">
    <members>
      <member>test1</member>
      <member>test2</member>
      <member>test3</member>
    </members>
    ...
  </runnable>
  <xi:include href="child.rmdb"/>
</rmdb>
```

and the included *child.rmdb* file contains the following:

```
<rmdb>
  <runnable name="test1" type="task">
    ...
  </runnable>
  <runnable name="test2" type="task">
    ...
  </runnable>
  <runnable name="test3" type="task">
    ...
  </runnable>
</rmdb>
```

These files together provide the same configuration as the following monolithic file:

```

<rmdb>
  <runnable name="nightly" type="group">
    <members>
      <member>test1</member>
      <member>test2</member>
      <member>test3</member>
    </members>
    ...
  </runnable>
  <runnable name="test1" type="task">
    ...
  </runnable>
  <runnable name="test2" type="task">
    ...
  </runnable>
  <runnable name="test3" type="task">
    ...
  </runnable>
</rmdb>
```

Note the namespace definition (*xmlns:xi*) on the document (*rmdb*) element. It is necessary to define the XInclude namespace in order for the XML parser to recognize the *xi:include* element. The namespace can be placed on any element containing the *xi:include* element, including the *xi:include* element itself.

Refer to “[Including Nested XML Files](#)” on page 583 for additional information.

## Set the Search Path for Shell Scripts

Like most Questa executables, VRM can be executed by typing the full path to the *vrun* executable, even if the Questa install directory is not included in the search path (that is, the *PATH* environment variable). If the Action scripts defined in the database are all TCL scripts (the default), then they are run from within a *vsim* shell invoked using the same path as that used to invoke VRM. In other words, everything should just work, even if the Questa install directory is not included in the user's *PATH*.

However, if the VRM database includes shell scripts, then the user needs to ensure that the *PATH* variable is correctly set (or just avoid relying on the value of the *PATH* variable entirely). The setting of *PATH* depends on the following:

- Value of the *PATH* variable when VRM is invoked.
- Whether the Action script is run on the local machine or on a server grid.
- Contents of any shebang line (starting with `#!/`) at the top of the Action script itself.

If the Action script is executed on the local machine (by the *default* execution method), then the environment under which the Action script is run is substantially the same as that under which VRM is invoked. Therefore, if the value of the *PATH* environment variable did not include the

Questa install directory when VRM was invoked, then it will not include the Questa install directory when the Action script executes and the script will not be able to invoke Questa utilities without supplying the full path to the executable. In order to avoid hard-coding tool paths into the database, it is recommended that the search path be set to include all necessary tool paths before VRM is invoked.

As an alternative, the full paths to tool executables can be parameterized and the actual paths passed into VRM using the *vrun -g* or *vrun -G* command-line options.

If the Action script is executed on a different machine (including via server grid management software), then the initial environment is determined by the shell and/or the grid management software and not by VRM. In most cases, that initialization is performed in an initialization file in the user's home directory (that is, *.cshrc* or *.profile*). Consult the documentation for your platform and/or the grid management software being used for more details.

In addition, if the shell script is to pick up the environment from VRM (in the case of execution on the local machine only), the shebang line (first line of the script) might require an option to prevent the shell from re-initializing the environment variables, especially *PATH*. For example, starting a C-shell script with the line *#!/bin/csh -f* will cause the shell to forgo environment initialization and use the environment variable values passed by the calling process. However, for scripts executed on another machine, no environment is passed from VRM so the chosen shell's environment initialization algorithm may be invoked anyway.

The safest solution that works on all platforms (without hard-coding tool paths into the database), is to make sure any shell initialization files (that is, *.cshrc*, *.profile*, and so on), on both the local machine and grid servers, initialize the *PATH* variable to include any necessary tool executable directories.

## Locating Script and Log Files

VRM maintains a working directory tree in which all Action scripts are executed. The directory hierarchy follows the Group membership hierarchy as given in the database. At the top of the tree is the *VRMDATA* directory (determined by *-vrmdata <directory>* command-line option, or *./VRMDATA* by default). Under that directory are subdirectories for each of the top-level Runnables named on the command line. For those Runnables that are Groups, subdirectories are created under the working directory for that Group (recursively until the leaf-level Tasks are reached).

Inside each of these subdirectories, several files are created. For Groups, there is a set of files for the *preScript* Action and a similar set for the *postScript* Action. For Tasks, there is a set of files corresponding to the *execScript* Action for that Task.

Each set of files consists of the following individual files (where the term *actionScript* is replaced by *preScript*, *execScript*, or *postScript* as appropriate):

- *actionScript.bat* — An optional shell script used when the Action is to be queued onto a server grid (or is launched by a non-default execution method).
- *actionScript.do* — The parameter-expanded Action script taken from the database.
- *actionScript.log* — The log output of the Action script itself.
- *actionScript.stderr* — The standard error stream from the *vish/vsim* wrapper script.
- *actionScript.stdout* — The standard output stream from the *vish/vsim* wrapper script. This file is only created when you specify the -savestdout option to *vrun*.
- *actionScript.wrap* — The *vish/vsim* wrapper script that invokes the Action script.

When user-script errors occur, the most likely place for them to be reported is in the *actionScript.log* file. If the Action failed to launch for some reason, then the problem might be found in either *actionScript.stderr* or *actionScript.stdout* (depending on the nature and origin of the error). If a VRM error occurred (that is, a problem with the database), then the error is reported on the standard output stream of the *vrun* process.

Other utilities (such as *vcover* and/or 3rd party tools) can produce their own diagnostic output. If these utilities are run as part of an Action script, then the diagnostic output most likely resides in the working directory corresponding to the Action in question (unless the Action script itself specifies another output file or path).

## Status Reports

---

This section describes the types of information reported by **vrun** about your regression run and the individual action scripts.

- [Action Script Status Reports](#) — End-of-regression information automatically written to the transcript.
- [Exit Code Status](#) — End-of-regression exit code information produced via the **-exitcodes** argument.
- [Regression Run Status Report](#) — Post-regression report, text- or HTML-based, produced via the **-status** switch.

Action Script Status Reports .....	68
Exit Code Status.....	70
Regression Run Status Report .....	71

## Action Script Status Reports

After the completion of every regression run, the following status reports are automatically written to the transcript.

- **Action Script Execution Status** — summarizes statistics about all actions prepared, launched and finished, providing a simple report of what **vrun** did during the regression run.

Actions that time-out are not counted in the finished category and any failed scripts that are re-run locally are counted for each attempt, therefore you may see regression runs where the total counts differ for the three categories.

The following is an example of this status report.

```
Action script execution status:  
Total 11 scripts prepared: 11 OK, 0 failed  
Total 11 scripts launched: 11 OK, 0 failed  
Total 11 scripts finished: 11 OK, 0 failed
```

- **Action Script Completion Status** — summarizes the completion status for the types of action scripts, where the status categories are defined as:

passed	completed successfully.
failed	completed, but encountered a failure.
launchfail	not executed due to a failure at launch-time.
timeout/queued	launched, but timed out before reporting a “start” event.

timeout/ executing	reported a “start” event, but timed out before reporting a “done” event.
killed	killed at the user's request.
skipped	skipped due to an earlier error or because the regression run was terminated at the user's request.
empty	not defined in the RMDB. For example, no commands were found in the RMDB for that action script.
noexec	not launched because the <b>-noexec</b> command line option was specified.
dropped	should have been launched but was not.

The following is an example of this status report.

```
Action script completion status:  
Total 16 execScript actions: 2 failed, 11 passed, 3 skipped  
Total 5 postScript actions: 1 empty, 3 passed, 1 skipped  
Total 5 preScript actions: 5 passed  
Total 2 triageScript actions: 2 passed
```

- **Simulation (UCDB) Status** — summarizes information based on TESTSTATUS values from the UCDB. This status report is only generated for action scripts that specify the **ucdbfile** parameter.

The following is an example of this status report.

```
Simulation (UCDB) status:  
Ok 10  
Warning 1  
Error 2  
Fatal 0  
Missing 0  
Merge error 0  
Unknown error 0  
UCDB error 0  
TOTAL 13
```

- **ActionScript pass/fail status** — high-level view of pass and fail counts for quick reference.

The following is an example of this status report.

```
ActionScript pass/fail status:  
Passed 11  
Failed 0  
Incomplete 0  
TOTAL 11
```

## Exit Code Status

Specifying the **-exitcodes** argument to the **vrun** command enables the reporting on the general status of the action scripts within the regression run.

The exit code is a bit-wise integer offering up to 16 different codes. The bit-wise description of the code is:

**Table 3-2. vrun -exitcodes Bit-Wise Definition**

Bit	Failure Description
0x08	The vrun process encountered an error.
0x04	One or more action scripts failed to run (launch error).
0x02	One or more action scripts failed to complete or completed with a non-zero exit code.
0x01	One or more simulations reported an error through the UCDB file.

## Regression Run Status Report

You can create a status report, in text or HTML format, that provides information about the most recent run executed in the current VRMDATA directory.

For both formats you can create the report in one of the following scenarios:

- After beginning a regression run — view the current status of your actions.
- After a regression run is complete — view the most recent results of your regression run.
- After several regression runs have completed — view the cumulative results of all regression runs associated with the VRMDATA directory (you must use the **-all** switch).

Action scripts selected to run but, for some reason, have not yet become eligible for launch are not listed in the status report.

The end of the status report also includes the information produced in the [Action Script Status Reports](#).

This information is also available through the [VRM Results Window](#) of the GUI.

The regression run status reports default to action-centric or test-centric mode you use for your regression suite. Refer to the section “[Test-centric Reporting and Control](#)” for more information.

<b>Creating the Text-based vrun Status Report .....</b>	<b>71</b>
<b>Creating the HTML-based vrun Status Report .....</b>	<b>72</b>
<b>Creating the TCL-Format vrun Status Report.....</b>	<b>72</b>
<b>Advanced Tasks .....</b>	<b>73</b>
<b>Regression Run Text Status Report Example.....</b>	<b>76</b>
<b>Regression Run HTML Status Report Reference .....</b>	<b>78</b>
<b>Regression Run Status Report Reference .....</b>	<b>82</b>

## Creating the Text-based vrun Status Report

For detailed information about the arguments mentioned, or an overview other arguments, refer to:

The ““[vrun Command](#)” on page 438”section.

1. Change directories to a location with a VRMDATA directory. You can alternatively use the **-vrmdata** switch to specify location of the regression run data.  
Information within this directory is used to generate the report.
2. execute **vrun-status** to write the text report to the transcript.

The report contains a portion of all the data available. You can add the following switches to change to alternate, predefined layouts:

- **-full** — shows nearly all information available. (Column titles: action, seed, status, user, host, cputime, queued, elapsed, datetime)
- **-times** — shows only information about run times for the actions. (Column titles: action, status, queued, elapsed, cputime, datetime)
- **-host** — shows only information about resource usage. (Column titles: action, seed, status, user, host, datetime).

## Creating the HTML-based vrun Status Report

For detailed information about the arguments mentioned, or an overview other arguments, refer to:

The “Status Mode Options” section of the [vrun Command](#)” section.

1. Change directories to a location with a VRMDATA directory. You can alternatively use the **-vrmdata** switch to specify location of the regression run data.

Information within this directory is used to generate the report.

2. Execute **vrun-status -html** to produce a stand-alone HTML report and supporting files in a subdirectory (*vrmhtmlreport*) of the current run directory. You can alternatively specify the **-htmldir** switch to create the directory in a different location.

Unlike the text-based report, you can instruct **vrun** to create the HTML-based status report upon completion of the regression run by adding the **-html** switch to the **vrun** command.

## Creating the TCL-Format vrun Status Report

For detailed information about the arguments mentioned, or an overview other arguments, refer to

The “Status Mode Options” section of the [vrun Command](#)” section.

1. Change directories to a location with a VRMDATA directory. You can alternatively use the **-vrmdata** switch to specify location of the regression run data.

Information within this directory is used to generate the report.

2. Execute **vrun -status -tcl** to produce a text-based report in TCL-list format.

The report contains a portion of all the data available. You can add the following switches to change to alternate, predefined layouts:

- **-full** — shows nearly all information available. (Column titles: action, seed, status, username, hostname, cputime, queued, elapsed, datetime)
- **-times** — shows only information about run times for the actions. (Column titles: action, status, queued, elapsed, cputime, datetime)

## Notes

- The date/time of the latest status event is reported in Unix epoch format; the **clock format** command may be used to convert this value to a human-readable string.
- The simulation time, CPU time, and elapsed times are reported in decimal seconds.
- Missing values are reported as empty strings.
- The **-notruncate** option is assumed; long strings will not be truncated.

## Advanced Tasks

VRM advanced tasks are described below.

### Creating Customized Layouts

You can control the order and appearance of columns in the status report with the **-columns** switch. This switch accepts a space or comma separated list of column titles, enclosed in quotes (" "), which produces a report in that specific order.

Refer to [Table 3-3](#) for a complete list of all columns.

If you specify a value that is not a valid column name, that string will appear in the report. For example, you can create the appearance of a column separator:

**vrun -status -columns "action,|,status"**

which produces a report similar to:

Actions :

Action		Status
=====	=	=====
...		
nightly/directed/dirttest1/execScript		Ok
...		

### Analyzing the TESTSTATUS of an Action

Every Action-specific UCDB contains a TESTSTATUS attribute showing the status of the test. The UCDB also stores additional information about the TESTSTATUS value:

- TSTAT\_REASON — contains the text of the last message issued before the TESTSTATUS attribute changed values to Warning, Error, or Fatal.
- TSTAT\_SIMTIME — contains the time at which the message was issued.

You can add this information to your text-based status report by adding the **-reason** switch, for example:

```
vrun -status -reason
```

The status report only lists actions where the status is Fatal, Error, or Warning, but does add the text and time information associated with TSTAT\_REASON and TSTAT\_SIMTIME. The report also suppresses the pass/fail tables typically found at the end of the report. For example:

```
Fatal/Error/Warning Actions:  
Action           Status  
Date/Time  
  
nightly/oneerror/test1_e/execScript      Error  
Mon Jun 27 09:27:27 AM PDT 2011  
Error message: This is an error message  
Error simtime: 0 ns
```

You can also include **reason** or **tstime** as arguments to the **-columns** switch to add this information to your report.

The HTML-based status report contains this information, by default, on the details page for each test.

### Reporting only Errors or Warnings

Limit the report to those actions with errors or warnings by specifying the **-errors** and/or **-warnings** options. In the case of **-errors**, actions whose status is either Fatal or Error is reported. In the case of **-warnings**, Actions whose status is Warning is reported.

### Filtering Results

The status report, by default, contains information about every action recorded in the VRM status event log. In some cases, this may entail a lot of data. However, you can use filtering techniques to limit the amount of information reported.

Use the **-filter** switch to the vrun command to specify filter expressions that control which actions are reported. You can specify multiple **-filter** switches on the command line, which acts as an AND operation. The syntax of this switch is:

```
vrun -status -filter <field><operator><string>
```

where:

- *field* — one of the report columns, refer to [Table 3-3](#)
- *operator* — one of the following operators: less than (<), equals (=), or greater than (>). Optionally, you can prepend these operators with the bang character (!) to reverse the sense of the expression. Refer to [Table 3-5](#) for more information.
- *string* — adheres to the rules defined in [Table 3-3](#), and is interpreted as a TCL regular expression. If this value includes spaces, you must surround the string in quotes.

For example, in order to produce a report that contains only those actions whose context string contains the substring *allpass*, use the following command:

```
vrun -status -filter action=allpass
```

Other filtering rules include:

- Lexical comparison, in this context, assumes strings consisting of a series of Unicode codepoints are compared according to English-language ordering rules.
- Regular expression matching follows TCL regular expression syntax.
- Numeric comparison assumes real values.
- Since an empty regular expression technically matches all strings, an exception is made for empty matching expressions. If the matching expression is blank and the field specified is a string, the expression is considered to have matched if the value of the field is also blank; otherwise, the expression does not match.
- It is suggested that you not use the **-hierarchical** switch with any of the filtering options because no attempt is made to prevent child nodes from appearing in the report with no visible parent. In other words, if intermediate parent nodes are filtered out while descendant nodes are reported, the structure of the displayed hierarchy may appear distorted.

Some examples include:

- Report actions that spent more than five minutes waiting to be executed:

```
vrun -status -filter queued>300
```

- Report actions that executed on host “fred” and took more than 30 seconds.

```
vrun -status -filter hostname=fred  
-filter elapsed>30
```

- Report actions associated with a UCDB file (that is, ucdbfile is not blank)

```
vrun -status -filter ucdbfile!=
```

- Report actions associated whose last status change was after 29 Sep 2010 (note UCDB date/time format)

```
vrun -status -filter datetime>20100929000000
```

#### Follow Status Events as They Occur

The status report has a special mode, enabled by the **-tail** switch, which reports the status events from a running **vrun** process as they occur. In this mode, rather than report the latest status for each action, the status report output contains every event for each Action. Moreover, if the regression run is still in progress (as indicated by the lack of an *end* event record), the **vrun** process that is reading the Status Event Log attempts to open a listening channel into the **vrun** process that is running the regression. If successful, the events reported via that channel are also emitted into the log output stream of the **vrun -status -tail** report generating process. This mode is intended mainly for debugging the status channel and is not designed to be useful in a production environment.

Since it only makes sense to “tail” a single regression run, the *-tail* command-line option considers only the latest Status Event Log and ignores the *-all* command-line option.

## Regression Run Text Status Report Example

The following shows an example report when using the default settings (Note that the columns have been shortened for formatting reasons).

```

# // Questa Verification Run Manager 10.6 Dec 13 2016
# //
# // Copyright 1991-2016 Mentor Graphics Corporation
# // All Rights Reserved.
# //
# // Questa Verification Run Manager and its associated documentation contain trade
# // secrets and commercial or financial information that are the property of
# // Mentor Graphics Corporation and are privileged, confidential,
# // and exempt from disclosure under the Freedom of Information Act,
# // 5 U.S.C. Section 552. Furthermore, this information
# // is prohibited from disclosure under the Trade Secrets Act,
# // 18 U.S.C. Section 1905.
# //
# # Actions:
#
# Action                                Testname   Status Coverage Date/Time
# ======  ======  ======  ======  ======
# nightly/preScript                      --        Empty     --    Thu Feb 02 ...
# nightly/directed/preScript             --        Empty     --    Thu Feb 02 ...
# nightly/directed/dirtest1/execScript   --        Passed(Ok) --    Thu Feb 02 ...
# nightly/directed/dirtest1/mergeScript  --        Passed     --    Thu Feb 02 ...
# nightly/directed/dirtest2/execScript   --        Passed(Ok) --    Thu Feb 02 ...
#
# ...
# Coverage summary:#                      Total Coverage Tplan Coverage Merge File
# ======  ======  ======  ======
#          0.00           -  .../vrm/simple/VRMDATA/merge.ucdb
# UCDB status summary:
#      6  Ok
#      0  Warning
#      0  Error
#      0  Fatal
#      0  Missing
#      0  Merge error
#      0  Unknown error
#      0  UCDB error
#      6  TOTAL
#
# Non-UCDB status summary:
#      8  Passed
#      0  Failed
#      8  TOTAL
#
# Action script completion status:
#      6  execScript actions: 6 passed
#      6  mergeScript actions: 6 passed
#      3  postScript  actions: 2 empty, 1 passed
#      3  preScript   actions: 2 empty, 1 passed
#
# ActionScript pass/fail status:
#      11 Passed
#      0 Failed
#      11 TOTAL

```

## Regression Run HTML Status Report Reference

The HTML status report contains three primary areas on the main page as well as a page for each individual action.

**Figure 3-3. Regression Run HTML Status Report**

## Verification Run Manager Status Report

<b>Regression started:</b>	Thu Mar 06 08:03:38 PST 2014
<b>Regression finished:</b>	Thu Mar 06 08:04:08 PST 2014
<b>Configuration (RMDB) file:</b>	/QuestaTestcases/vrm/simple/default.rmdb
<b>VRMDATA directory:</b>	/QuestaTestcases/vrm/simple/VRMDATA
<b>Command:</b>	vrun -nobatch -runlog

---

### Coverage Summary:

Total Coverage	Testplan Coverage	Merged UCDB
0.00	-	QuestaTestcases/vrm/simple/VRMDATA/merge.ucdb

---

### Pass/fail status summary:

Status	Count
Passed	11
Failed	0
Incomplete	0
<b>TOTAL</b>	<b>11</b>

### UCDB status summary:

Status	Count
Ok	6
Warning	0
Error	0
Fatal	0
Missing	0
Merge Error	0
Unknown Error	0
UCDB Error	0
<b>TOTAL</b>	<b>6</b>

### Non-UCDB status summary:

Status	Count
Passed	5
Failed	0
Timeout	0
<b>TOTAL</b>	<b>5</b>

### Action completion summary:

Status	Empty	Passed
execScript	0	6
mergeScript	0	3
postScript	2	1
preScript	2	1

### Action status:

Show All	Show Passed	Show Failed
Runnable	Script	Testname
nightly	<a href="#">preScript</a>	--
nightly/directed	<a href="#">preScript</a>	--
nightly/directed/dirstest1	<a href="#">execScript</a>	--
		Coverage
		0.00
		Date/Time
		Thu Mar 06 08:03:38 PST 2014
		Status
		Empty
		Empty
		Passed (Ok)

- Header — contains information about the start and finish times and the locations of the RMDB file, VRMDATA directory, and links to merged coverage reports.
- Summary reports — contains high-level status summaries:
  - Coverage summary — coverage statistics and links to merged UCDB files.
  - Pass/fail status summary — high-level count of passed, failed, and incomplete action statuses.
  - UCDB status summary — a count of actions for each possible UCDB status.
  - Non-UCDB status summary — a count of actions that are not expected to produce a UCDB file, such as compilation or post-process actions.
  - Action completion summary — a count of completed actions for the various action types.
- Action status reports — contains a table listing every Action in the regression run. You can control the appearance of this section with the various switches associated with the **-status** switch, refer to the ““vrun Command” on page 438” section.
  - Show All — displays all Actions.
  - Show Passed — filtered view to show only passing Actions.
  - Show Failed — filtered view to show only failing Actions. Actions that did not run because no commands were specified (Empty Actions) are not included in either the passing or failing lists.
- Individual action report pages — Each Action in the action status report section has a link that points to a detail page.
  - Header — a hierarchical representation of the Action, its status, and a link to its log file.
  - Per-test attributes — information for all status fields for that Action.
  - UCDB test data record attributes — (only for actions that produce a UCDB file) values of all the attributes found in the UCDB, including user-defined attributes
  - Script — the contents of the script itself (including the Runnable from which the script’s contents was taken).
  - Other information — copy of the last status event record from the status event log (“done” event for Actions that completed normally or some other intermediate record for Actions that are still running or failed to complete).

## Regression Run Status Report Reference

The Status Report Output Columns table provides an overview of all the columns available in the text and HTML reports and the VRM Results window of the GUI.

**Table 3-3. Status Report Output Columns**

Column Name	String to use with -column and -filter	Source	Datatype	Description
Action	action	event log	string	Action context string. This column supports truncation, use <b>-notruncate</b> to disable this feature.
CPU Time	cputime	UCDB file <sup>1</sup>	numeric	CPU time as reported in the UCDB file, in seconds.
Date/Time	datetime	event log	numeric	Date and time of the most recent event. For an action in the running state, this shows the start time. For an action in the completed state, this shows the end time.
Elapsed Time	elapsed	event log	numeric	Elapsed time, in seconds, from start-to-done. This is empty for any actions which is not completed.
Ext Status	extstatus	event log	string	Extended status, providing additional information beyond the Run Status
Hostname	hostname	event log	string	Hostname where Action ran. This column supports truncation, use <b>-notruncate</b> to disable this feature.
Iteration	iteration	event log	numeric	Value dependent upon on the <b>repeat</b> attribute (integer) or the <b>foreach</b> attribute (string) of a runnable, if specified.
Mergefile <sup>2</sup>	mergefile	RMDB file	string	Path to the UCDB file to which the coverage for this test was merged. This value will be blank if it was not specified for the action.
Queued Time	queued	event log	numeric	Elapsed time, in seconds, from launch-to-start. This is empty for any actions which have not been launched.

**Table 3-3. Status Report Output Columns (cont.)**

<b>Column Name</b>	<b>String to use with -column and -filter</b>	<b>Source</b>	<b>Datatype</b>	<b>Description</b>
Raw Status	rawstatus	event log	string	Status as reported in the event log
Test Status Reason	reason	UCDB file	string	Displays the value of TSTAT_MESSAGE, if any. This UCDB field can contain the text of the message that led to a change in TESTSTATUS.
<parameter>	RMDB:<parameter>	RMDB file	string	Displays the resolution of the specified parameter, such as %seed%, from the point of view of the action being reported. The <parameter> value must be a case-sensitive match, excluding the percent marks (%).
Testname	testname	RMDB file	string	testname associated with a Runnable.
Runnable	runnable	event log	string	Runnable name.
Run Status	runstatus	event log	string	Status summary
Seed	seed	event log	numeric	Random seed used.
Sim Time	simtime	UCDB file	numeric	Displays the total simulation time, in seconds, for the test. This only applies to Actions that generate a UCDB file containing the SIMTIME test data record attribute.
Status	status	event log	string	Displays a user-readable combination of the runstatus and extstatus columns.  If a UCDB file is available, this column contains a string version of the TESTSTATUS attribute.  If an action is not yet completed, it contains the current state of the action. Refer to <a href="#">Table 3-4</a> for a complete list of possible values.
Triagefile <sup>2</sup>	triagefile	RMDB file	string	Path to the TDB file in which the messages for this test were inserted.

**Table 3-3. Status Report Output Columns (cont.)**

Column Name	String to use with -column and -filter	Source	Datatype	Description
Test Status Time	tstime	UCDB file	numeric	Displays the value of TSTAT_SIMTIME, if any. This UCDB field can contain the simulation time of when TESTSTATUS changed.
UCDB File	ucdbfile	event log	string	Name and location of the UCDB file.
UCDB Status	ucdbstatus	UCDB file	string	Status of the Action as reported in the UCDB file. This value will be blank if the action is not associated with a UCDB file.
Username	username	UCDB file	string	Name of user who ran the Action.

1. Including one or more of columns that source an action's UCDB file causes vrun to read the test data records from each of the UCDB files produced by the regression run. For large regression suites, this can have an impact on report generation performance.
2. The path is the one to which the coverage would be merged or the messages would be inserted if those options take place. But the decision whether or not to merge/insert depends on the pass/fail status of the test and the behavior can be decided by the user. The fact that the paths to the merged UCDB and the TDB file show up in their respective columns does not guarantee the merge/insertion actually took place; only that there is a file to which the merge/insertion can take place.

Table 3-4 provides an overview of the possible values of the status columns in the text and HTML status reports and the VRM Results window of the GUI

**Table 3-4. Possible Values for the Various Status Columns**

Raw Status	Run Status	Ext Status	UCDB Status	Status
	Description			
eligible	Pending			Pending
	The Action is eligible to run but has not yet been launched.			
launched	Queued			Queued
	The Action has been launched but has not yet started running.			
running	Running			Running
	The Action is currently running.			
suspended	Suspended			Suspended
	The Action has started but is currently suspended.			

**Table 3-4. Possible Values for the Various Status Columns (cont.)**

Raw Status	Run Status	Ext Status	UCDB Status	Status
	<b>Description</b>			
resumed	Running			Running
	The Action is currently running (having previously been suspended).			
empty	Empty			Empty
	The Action does not define a script.			
passed/ok	Passed	Ok	Ok	Passed (Ok)
	The UCDB file has an “Ok” value in the TESTSTATUS attribute.			
passed/warning	Passed	Warning	Warning	Passed (Warning)
	The UCDB file has a “Warning” value in the TESTSTATUS attribute.			
passed	Passed			Passed
	The Action passed and no UCDB file was expected.			
failed/launch	Failed	Launch		Failed (Launch)
	The Action failed to launch (system error).			
failed/retry	Failed	Resource		Failed (Resource)
	The Action script requested to be re-run (should not happen unless re-run limit is exceeded).			
failed/unknown	Failed	Unknown		Failed (Unknown)
	The pass/fail status of the Action cannot be determined.			
failed/script	Failed	Exit Code		Failed (Exit Code)
	The Action script returned a non-zero exit code.			
failed/stderr	Failed	Stderr		Failed (Stderr)
	The Action script wrote to stderr.			
failed/ucdb	Failed	UCDB Error		Failed (UCDB Error)
	A UCDB file was expected but no UCDB file was found or the UCDB could not be read.			
failed/error	Failed	Error	Error	Failed (Error)
	The UCDB file has an “Error” value in the TESTSTATUS attribute.			
failed/fatal	Failed	Fatal	Fatal	Failed (Fatal)
	The UCDB file has a “Fatal” value in the TESTSTATUS attribute.			

**Table 3-4. Possible Values for the Various Status Columns (cont.)**

Raw Status	Run Status	Ext Status	UCDB Status	Status
	<b>Description</b>			
failed/missing	Failed	Missing	Missing	Failed (Missing)
	The UCDB file has a “Missing” value in the TESTSTATUS attribute.			
failed/merge	Failed	Merge Error	Merge	Failed (Merge Error)
	The UCDB file has a “Merge Error” value in the TESTSTATUS attribute.			
timeout/unknown	Timeout	Unknown		Timeout (Unknown)
	The Action timed out in an unknown state.			
timeout/queued	Timeout	Queued		Timeout (Queued)
	The Action timed out while before starting.			
timeout/executing	Timeout	Executing		Timeout (Executing)
	The Action started but timed out before finishing.			
killed	Killed	(n/a)		Killed
	The Action was launched but was later terminated.			
skipped	Skipped	(n/a)		Skipped
	The Action was not launched due to earlier errors.			
dropped	Dropped	(n/a)		Dropped
	The Action was not launched because the regression was terminated (or VRM internal error).			
noexec				
	The Action was not launched because the -noexec option was specified.			

**Table 3-5. Filter Operators**

Operator	String Interpretation	Numeric Interpretation
<	Field value is lexically “less than” the specified string	Field value is less than the specified value
=	Field value matches the specified regular expression	Field value equals the specified value
>	Field value is lexically “greater than” the specified string	Field value is greater than the specified value

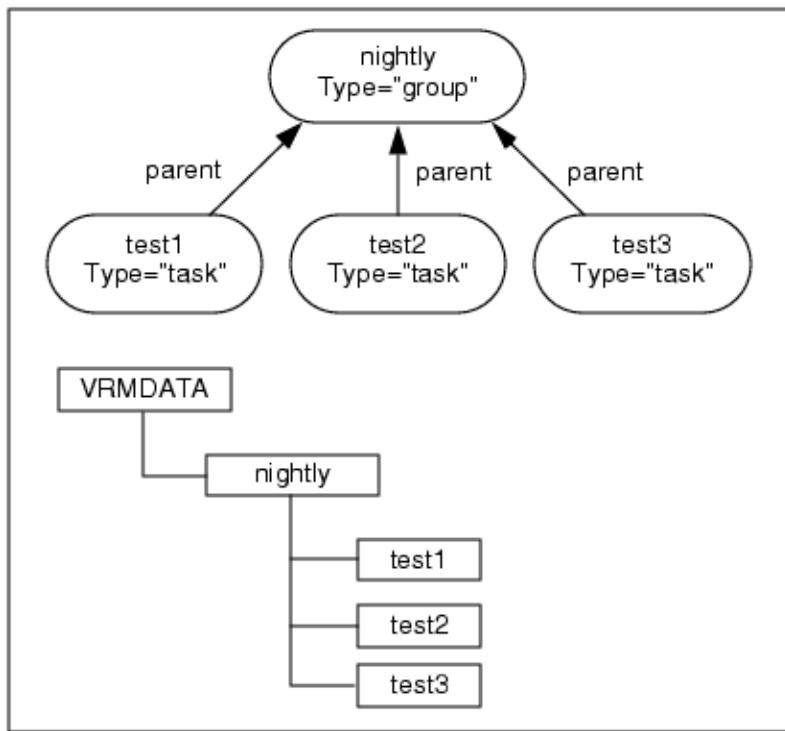
# Managing the VRMDATA Directory

VRM maintains a *VRM Data* directory tree under which each Action executed is assigned a directory based on its context chain.

By default, the head of this *VRM Data* directory tree is called **VRMDATA** and is located within the directory from which the *vrun* executable is invoked. This location can be changed using the **-vrmdata** command-line option (see [Figure 3-4](#) on page 87).

In the VRM documentation, the head of this *VRM Data* directory tree is often referred to as the **VRMDATA** directory, even if some other name is actually used as a result of the **-vrmdata** command-line option. The **VRMDATA** directory and its contents are created by VRM as needed. All Actions launched by VRM are executed from within a subdirectory of the **VRMDATA** directory.

**Figure 3-4. VRM Data Directory Tree Structure**



VRM detects when multiple regressions are trying to use the same **VRMDATA** directory. When a regression is running, a lock file is posted to the **VRMDATA** directory. If a subsequent regression pointing to the same directory is started before the first regression run has completed, a warning is posted to *stdout* and *vrun* waits 30 seconds before continuing. If the subsequent regression is launched from the VRM GUI, a warning dialog is posted asking the user to confirm or cancel the subsequent regression.

Because the **VRMDATA** directory also houses the log files for each regression run executed therein, it is common to run the same regression suite (or select parts thereof) in the same **VRMDATA** directory without deleting the **VRMDATA** directory itself.

However, in some cases it might be prudent to start from scratch with a clean **VRMDATA** directory. In this case, the *-clean* command-line option directs the *vrun* process to delete the entire **VRMDATA** directory, including any previous log files, before executing any Actions.

For more information see “[Working Directory and Script Execution](#)” on page 287.

<b>Manage a Questa work Directory and HDL Sources.....</b>	<b>88</b>
<b>Implement a Macro by Calling VRM Recursively .....</b>	<b>90</b>

## Manage a Questa work Directory and HDL Sources

Even though VRM can be used in any number of regression-like flows, the most common is the Questa compile-simulate flow. In this flow, some number of HDL source files are compiled into a working library and said library is referenced by some number of simulations. In a typical traditional flow, the compilation commands and the simulations commands are commonly run in the same directory.

Under VRM, the compilation is often associated with a *preScript* Action somewhere in a top-level Group Runnable and the simulation is associated with one or more *execScript* Actions in leaf-level Task Runnables. In the VRM implementation, each Group and each Task runs in its own **VRMDATA** directory. This presents a challenge in making sure all the Action commands can access the working library easily.

There are three directory paths that are defined and fixed on a per-run basis and are also available as predefined parameters as follows (see “[Predefined Parameter Usage](#)” on page 266 for additional information):

- **VRUNDIR** — The directory from which VRM is invoked.
- **RMDBDIR** — The directory from which the database file is read.
- **DATADIR** — The root of the *VRM Data* directory tree.

In order for the regression to be able to be run from any directory, relying on **VRUNDIR** is not recommended. That leaves two useful fixed directories.

The value of the **RMDBDIR** predefined parameter is relative to the location of the database and, therefore, makes a good location for HDL source files and other user-maintained data files. If these files are in the same directory as the database (RMDB) file, then the entire directory can be moved without affecting the RMDB file contents.

The value of the **DATADIR** predefined parameter is relative to the location of the **VRMDATA** directory and, therefore, makes a good location for generated files, including the Questa

working directory. A working directory called (%DATADIR%)/work can be defined (even in another parameter) and used as the value of a library related command-line option for those commands needing to access the working library.

[Example 3-5](#) illustrates the use of both predefined directories in a database.

**Figure 3-5. Predefined DATADIR and RMDBDIR Directories in a Database**

```

<rmdb>
<runnable name="nightly" type="group">
    <parameters>
        <parameter name="ucdbfile">../test (%seed%) .ucdb</parameter>
    </parameters>
    <members>
        <member>test1</member>
        <member>test2</member>
        <member>test3</member>
    </members>
    <preScript>
        <command>file delete -force (%DATADIR%)/work</command>
        <command>vlib (%DATADIR%)/work</command>
        <command>vlog -sv -cover bces -work (%DATADIR%)/work
                    (%RMDBDIR%)/src/top.sv</command>
    </preScript>
    <execScript>
        <command>vsim -c top -lib (%DATADIR%)/work -sv_seed
                    (%seed%)</command>
        <command>run -all</command>
        <command>coverage attribute -name TESTNAME -value
                    test (%seed%)</command>
        <command>coverage save (%ucdbfile%)</command>
    </execScript>
    <postScript>
        <command>vcover merge merge.ucdb test*.ucdb</command>
    </postScript>
  </runnable>
  <runnable name="test1" type="task">
    <parameters>
        <parameter name="seed">123</parameter>
    </parameters>
  </runnable>
  <runnable name="test2" type="task">
    <parameters>
        <parameter name="seed">456</parameter>
    </parameters>
  </runnable>
  <runnable name="test3" type="task">
    <parameters>
        <parameter name="seed">789</parameter>
    </parameters>
  </runnable>
</rmdb>
```

Note that the HDL source file is located at `(%RMDBDIR%)/src/top.sv` and that the working library into which it is compiled is located at `(%DATADIR%)/work`. The `vlog` command uses the `-work` option to specify the working library while the `vsim` command uses the `-lib` option.

The `execScript` is owned by the “*nightly*” Group so that the same script is used for all simulations. Each simulation supplies only a random seed (in the `seed` parameter of the Task Runnable associated with the simulation). The UCDB files are named for the seed and stored in the `VRMDATA` directory associated with the “*nightly*” Group.

## Implement a Macro by Calling VRM Recursively

VRM can recursively call itself, thus encapsulating key command-line options into the database. VRM can run any legal command as part of an Action script, including the `vrun` command itself. This property of VRM can be exploited to provide a robust methodology for encapsulating one or more `vrun` commands into a higher-level macro-like Runnable object.

As an example, suppose a regression suite with a top-most Group called “*top*” as follows:

```
<rmdb>
  <Runnable name="top" type="group">
    ...
  </Runnable>
  ...
</rmdb>
```

Suppose also that you want to run this regression under a `VRMDATA` directory located in the same directory as the database itself. Next suppose that you want to pass in a parameter named `src` that points to a subdirectory of that same directory, also named `src`. In addition, suppose you want to increase the default minimum timeout to 3600 seconds. The command for launching this regression might look something like the following:

```
vrun -vrmdata ./VRMDATA -gsrc=./src
-mintimeout 3600 top
```

In order to encapsulate the command-line options into the database, define a new Task Runnable that is not linked into the existing regression suite. Any references (direct or indirect) to the current working directory (including the default `RMDB` file) should be replaced with the

parameter reference (`%RMDBDIR%`) so that you can call this macro from any directory as follows:

```
<rmdb>
  <runnable name="nightly" type="task">
    <execScript>
      <command>vrun -rmdb (%RMDBFILE%) -vrldata (%RMDBDIR%)/VRMDATA
                  -gsrc=(%RMDBDIR%)/src -timeout 3600 top</command>
    </execScript>
  <runnable name="top" type="group">
    ...
  </runnable>
  ...
</rmdb>
```

Now just type `vrun nightly` and the VRM process kicks off a single `execScript` Action that launches another copy of `vrun` complete with all the correct command-line options. Note that a special variable called (`%RMDBFILE%`) is used to point to the RMDB file, since the nested `vrun` process is launched in the working directory for the “*nightly/execScript*” Action.

There is no limit to how many such macro Runnables can be defined. There is also no limit to how many times the recursive `vrun` command can be invoked in one macro Runnable. For example, if a given regression suite is typically run in two different modes (“*fast*” and “*slow*”), and if the mode is parameterized, the following configuration allows either or both modes to be run from a single, simple `vrun` command:

```
<rmdb>
  <runnable name="fast" type="task">
    <execScript>
      <command>vrun -rmdb (%RMDBFILE%) -vrldata (%RMDBDIR%)/VRMDATA
                  -gsrc=(%RMDBDIR%)/src -Gmode=fast -timeout 3600 top</command>
    </execScript>
  </runnable>
  <runnable name="slow" type="task">
    <execScript>
      <command>vrun -rmdb (%RMDBFILE%) -vrldata (%RMDBDIR%)/VRMDATA
                  -gsrc=(%RMDBDIR%)/src -Gmode=slow -timeout 3600 top</command>
    </execScript>
  </runnable>
  <runnable name="both" type="task">
    <execScript>
      <command>vrun -rmdb (%RMDBFILE%) -vrldata (%RMDBDIR%)/VRMDATA
                  -gsrc=(%RMDBDIR%)/src -Gmode=fast -timeout 3600 top</command>
      <command>vrun -rmdb (%RMDBFILE%) -vrldata (%RMDBDIR%)/VRMDATA
                  -gsrc=(%RMDBDIR%)/src -Gmode=slow -timeout 3600 top</command>
    </execScript>
  </runnable>
  <runnable name="top" type="group">
    ...
  </runnable>
  ...
</rmdb>
```

The user can also use the grouping functionality of the VRM to define the “*both*” Runnable as a Group whose members are the “*fast*” and “*slow*” Runnables. Care must be exercised in order to avoid creating a recursive loop of *vrun* commands that no single VRM invocation would be able to detect. (Note that it is also possible to avoid this problem by defining the macro Runnables in one database file and the actual Group and Task Runnables in another database file to which the nested *vrun* commands would refer.)

## Limit Concurrently Running Processes

By default, VRM launches Actions as soon as they become eligible (that is, as soon as any dependencies are met). This means that, in many cases, it is possible for VRM to attempt to launch a large number of concurrent processes at one time.

There are some situations where the user wants to control the number of concurrently running Actions. For example, your company can have an in-house limit on how many jobs one user is allowed to submit to a given grid system at any one time. The user may wish to avoid the grid system altogether and run simulations on the local machine. In the case of a multi-core machine, the user can implement a “poor-man’s grid” by launching a limited number of Actions on the local machine (usually the number of cores minus one) with the jobs run concurrently in the background.

This limit on concurrent processes is imposed by using the *-j <integer>* option on the command line. The numeric value after the *-j* option becomes the new limit. If the limit is less than or equal to zero, then no limit is imposed on the number of processes that can be running at any one time (see “[vrun Command](#)” on page 438).

## Override Repeat Counts on Runnable Nodes

The procedure for overriding the repeat count of any given Task or Group depends on how the Runnable is specified.

If the Runnable explicitly defines a *repeat* attribute that contains a parameter reference, then the repeat count can be controlled by overriding the parameter as described in “[Override Parameter Values from Command Line](#)” on page 132”. In addition, the repeat count of any Task or Group can be overridden from the command line using the *-R <name>=<integer>* option, even if a *repeat* attribute has not been defined for that Runnable.

The syntax of the *-R* option is similar to that of the *-G<name>=<value>* option. The only difference is the name passed to the option is not the name of a parameter but rather the name of the Runnable whose repeat count is to be changed. A qualifying context can also be provided. The value of the option contains the new repeat count.

The *-R* option can also be used to override a parameterized repeat count, in which case the value passed to the *-R* option has precedence over the parameter-expanded value (even if overridden by the *-G* option).

Regardless of how the repeat count override occurs, if the ultimate value is less than or equal to one (or is a non-numeric string), then the Runnable will not repeat, even if the *repeat* attribute is set on the *Runnable* element. If the value is numeric and greater than or equal to two, then Runnable is repeated that number of times, according to the algorithm described above.

## Integration with Compute Grid Systems

---

VRM does not provide compute grid management, nor is it tied to any particular compute grid management product. While this allows VRM to be used in a wide variety of end-user environments, the generalized flexibility comes at the price of tight integration.

It is possible to launch an Action using any grid-management product that provides a batch-mode grid-submission command. It is also possible to add user-definable procedures to VRM to handle grid job deletion and status reporting.

For three grid-management products, Platform Computing Corporation Load Sharing Facility (LSF), Univa Grid Engine (UGE), Oracle Grid Engine, previously known as Sun Grid Engine (SGE), and Realtime Design Automation's NetworkComputer (RTDA) a degree of integration is already built-in to VRM.

Refer to “[Aggregating Multiple Actions in a Single Job](#)” on page 336 and “[Controlling Grid Jobs](#)” on page 477 for additional information.

<b>Launch an Action Onto a Grid System .....</b>	<b>94</b>
<b>Interaction with a Job Once Launched .....</b>	<b>96</b>

## Launch an Action Onto a Grid System

Regardless of the grid system in use, the way to control how Actions are launched is through execution methods. An execution method is basically a script-launching command. When an execution method is involved, the *vrun* process writes a batch (.bat) file that contains the *vsim/vish* command needed to launch the Action. It then passes the name of this batch file to the execution method command. If this execution method command happens to be a grid-submission command, then the Action is queued to the associated grid.

Following is one possible execution method command for submitting an Action to a UGE grid:

```
qsub -N (%RUNNABLE%) -V -cwd  
-b yes -o (%TASKDIR%) -e (%TASKDIR%) (%WRAPPER%)
```

The (%WRAPPER%) parameter reference is replaced by the name of the VRM-generated batch file. The other options control things like the name of the grid job and where the output/error results are written. This command is then placed in a *method* element and inserted in the appropriate place in the RMDB file (see “[Execution Methods](#)” on page 315 for details on how execution method are selected):

```
<method name="ugegrid" action="execScript" mintimeout="300"  
gridtype="uge">  
  <command>qsub -N (%RUNNABLE%) -V -cwd -b yes -o (%TASKDIR%) -e  
  (%TASKDIR%) (%WRAPPER%)</command>  
</method>
```

The *gridtype* attribute contains a (mostly arbitrary) string that tells *vrun* how to find user-definable procedures associated with the various functions that can be called via the grid-management software.

The way the *gridtype* maps to a specific set of user-definable procedures is generating procedure names on-the-fly according to a convention. For example, each basic function one can call via the grid-management is given a name, such as *KillJob* to kill a specific job. The string in the *gridtype* attribute is title-capitalized and prepended onto the function name. So, in the above example, the *uge* *gridtype* would be title-capitalized to *Uge* and concatenated to the *KillJob* function name, giving the procedure name *UgeKillJob*.

If there is no procedure defined by this generated name, then the function in question is not defined for that *gridtype*. If there is such a function, it is called with the usual Action-based user-data hash. It is up to the user-definable procedure to turn this information into one or more grid-management commands with the appropriate arguments. Default implementations of these user-definable procedures are built-in for both LSF, UGE, SGE and RTDA. However, by selecting some other string to use in the *gridtype* attribute and supplying the proper user-definable procedures, VRM can easily be integrated with just about any existing grid-management system.

## Interaction with a Job Once Launched

Once VRM launches the batch file that queues a job to a grid system, the only control and/or visibility is via the log files produced by the job and the utilities provided by the grid system itself. In order to interact with the various log files and utilities flexibly enough to accommodate a wide variety of grid software variants, a number of user-definable procedures are used. These particular user-definable procedures, however, are named for the grid system to which they were intended to interface. In other words, each grid system supported by VRM will have its own set of user-definable procedures. Additional user-definable procedures can be defined by the user to support in-house grid systems as well.

<b>Finding the JobId.....</b>	<b>96</b>
<b>Reporting on Grid-job Status.....</b>	<b>96</b>
<b>Killing a Queued Job.....</b>	<b>97</b>

### Finding the JobId

Every grid system includes a way to identify jobs queued to the grid. In the LSF, UGE, SGE and RTDA products, these *JobId* are numeric strings.

The user-definable procedure corresponding to the function name *GetJobId* fetches the *JobId* resulting from the grid-based launch of a given Action, or a blank string if the *JobId* cannot be determined. In the default implementations for LSF, UGE, SGE and RTDA, this procedure is called by the user-definable procedure corresponding to the function name *KillJob*. However, it might also be possible to tag a job with the complete Action context chain (see [page 104](#)) when the job is first queued, in which case it might be unnecessary to extract the *JobId* string from the Action's log files.

### Reporting on Grid-job Status

The start/done and pass/fail status of Actions launched on a grid system is still communicated to the launching *vrun* process via the TCP/IP channel opened for that purpose. That means that the machine on which the launching *vrun* process is running and the machine(s) on which the Action(s) will ultimately run must be connected via a network and that the launching machine must be visible from the grid machines.

The *stdout* file generated in the working directory (*xxxScript.stdout*) contains the output of the grid-submission command and is assumed to contain the *JobId* of the resulting grid job. The only other status needed by *vrun* that is not communicated via the TCP/IP channel is the error output of the Action. This is picked up by a user-definable procedure corresponding to the function name *GetStderr*. So, in the case of our UGE example, the user-definable procedure called would be *UgeGetStderr*.

By default, this procedure looks for the latest error file named *xxxScript.bat.e* and returns its contents. The *vrun* process calls this user-definable procedure only when the Action is reported

to have failed; in which case, the error file contents supplied by this user-definable procedure are printed to the *vrun* log output.

Note that the Load Sharing Facility (LSF) has a mode in which the *stderr* file is written to a temporary location while the job is running and then, after the job is complete, the temporary file is copied into the user's working directory. Needless to say, this creates a race condition between the files appearance and when VRM read checks for *stderr* messages. In order to make LSF play well with VRM, it may be necessary to set the *LSB\_STDOUT\_DIRECT* variable (found in the */etc/lsf.conf* file) to "y" or "Y".

## Killing a Queued Job

To kill a queued/running job, a user-definable procedure corresponding to the function name *KillJob* is called. In the case of the UGE example, the procedure name is *UgeKillJob*. By default, this procedure scans *xxxScript.stdout* for a *JobId*. Upon finding a *JobId* corresponding to the Action in question, it is passed to the *qdel* command to terminate the job. The *vrun* process only calls this user-defined procedure in the event the Action fails with a timeout (in which case the *KillJob* operation is simply a precaution) or in the case where the regression is manually killed by the user.

## Specify Selected Runnables

VRM database (RMDB file) generally contains data for one or more regression test suites arranged in a hierarchical tree topology that allows the user to group tests and other tasks in various ways. Each node in this tree is known as a “Runnable” because it could, in theory, be designated as one of the top-most nodes to be executed in any given regression run. Command-line options are used to designate which part(s) of this hierarchy are to be executed.

For example, assume that you have a database with the following basic topology:

- *nightly*
  - *directedtests*
    - *alutests*
    - *cputests*
  - *unittests*
    - *test123*
    - *test456*
    - *test789*
    - *mode1tests*
      - ...more children...
    - *mode2tests*
      - ...more children...
  - *randomtests*
    - *smoketest*
    - *blocktest (repeat=4)*

The *nightly* Runnable provides a single top-level Runnable from which the entire regression suite can be launched. The three Runnable members under *nightly* (*directedtests*, *unittests*, and *randomtests*) happen, for purposes of this example, to be configured such that they can be launched stand-alone without any information from their *nightly* parent Runnable. For any given *vrun* invocation, the user can elect to run the entire trio of regression suites, or choose to run only one subclass of tests.

One of the first things the user must consider when using VRM is the selection of Runnables from the RMDB file for any given run. The easiest way to do that is to simply list the Runnables

applicable to that run on the command line as bareword arguments. Therefore, given the example above, the following command:

**vrun unittests**

cause the *unittests* sub-Group of the regression suite to be executed. The other sub-Groups are not considered unless they were also listed as descendants (members or members-of-members at any depth) of the *unittests* sub-Group. For example, this command does not cause the *randomtests* Runnable, or any of its descendants, to run.

Selecting any given Runnable from the command line also selects its children and their children, to any depth (limited only by the capacity of the application and the machine on which it is running). So the above command also selects Tasks and Groups listed in the membership list of the *unittests* Runnable, recursively to the leaves of the subtree whose head is the *unittests* Runnable.

A single test could, in theory, be run directly from the command line as follows:

**vrun smoketest**

In this case, VRM treats the leaf-level Task *smoketest* as though it is a top-level Runnable. A working directory for *smoketest* is created directly under the VRMDATA directory specified by the *-vrldata* option (as opposed to being created at VRMDATA/*nightly/randomtests/smoketest* (see “[Working Directory and Script Execution](#)” on page 287 for details). More importantly, any parameters referenced in commands within *smoketest* that are not defined within the *smoketest* Task itself (or a base runnable declared in the Task) needs to be specified as overrides or defaults on the command line or an error is reported. Inheritance of parameters and the *execScript* definition according to the “calling chain” (see “[Context Chains](#)” on page 104) cannot occur in this case because VRM does not know on which Group's behalf the *smoketest* Task is running (recall that Tasks can be members of multiple Groups).

Moreover, if there are compile steps and/or post-simulation merge steps in the *preScript* or *postScript* of the Group(s) that refer to *smoketest* (in this case *nightly* and *randomtests*), these are executed and the isolated *smoketest* Task is, therefore, likely to fail. These are issues that are easily overcome as described in the “[Context Chains](#)” on page 104 section.

<b>Showing which Runnables are Selected.</b> . . . . .	<b>100</b>
<b>Selecting Instances of Repeating Runnables</b> . . . . .	<b>100</b>
<b>Wildcards and Repeating Runnables</b> . . . . .	<b>101</b>
<b>Selecting Runnables Based on Results of Previous Run</b> . . . . .	<b>102</b>
<b>Context Chains</b> . . . . .	<b>104</b>
<b>Rooted Context Chains</b> . . . . .	<b>106</b>
<b>Globbing in Context Chains</b> . . . . .	<b>107</b>
<b>Using Regular Expressions in Context Chains</b> . . . . .	<b>107</b>
<b>Adhoc Mode</b> . . . . .	<b>108</b>

## Showing which Runnables are Selected

The *vrun* command supports a *-show* option that, rather than actually running the selected regression Actions, simply lists what is selected by the other command-line options and, therefore, what would run if the *-show* option is not used.

By default an abbreviated list of Runnables is shown, listing the Runnables directly selected by command-line arguments and their immediate members. The *-all* option lists all selected Runnables, recursively down the hierarchy to the leaf-level.

In the example that follows, the *-show* option is used to produce an abbreviated listing of the selected Runnables (but not run). In order to distinguish the *vrun* output from the typed command, the command prompt % is shown with the *vrun* command.

The *-show* listing for the example in the [Specify Selected Runnables](#) section on page 98 might look something like the following:

```
% vrun -show nightly/unittests
nightly/unittests
nightly/unittests/test123
nightly/unittests/test456
nightly/unittests/test789
nightly/unittests/mode1tests/...
nightly/unittests/mode2tests/...
```

Note that the ellipsis (...) in the listing indicates where the tree of selected Runnables is truncated. In this case, *mode1tests* and *mode2tests* are Groups whose members are implicitly selected by virtue of the selection of the Group itself. The complete list can be seen by specifying the *-all* option.

## Selecting Instances of Repeating Runnables

In the RMDB file, Groups and Tasks can be defined as either conditional or repeating. Conditional Runnables do not have any special notation, as they are either present under the specified conditions or not. However, repeating Runnables are expanded into multiple copies/instances and the Runnable selection notation needs to accommodate the selection of one or more of a set of repeating Runnables.

The Runnable attributes that govern repeating are *repeat* that specifies a simple incrementing count, and *foreach* that specifies a specific list of instance values. Each instance of a repeating Runnable is given a name that starts with the Runnable name, to which a unique instance identifier is appended after a tilde (~) character. This name is used in the context chain for repeating Runnables or their descendants.

For example, if the *randomtests* Group under *nightly* has a member Task called *blocktest* that has a *repeat* attribute of four, the following command lists the expanded instances of *blocktest*:

```
% vrun -show -include nightly/randomtests/blocktest
nightly/randomtests/blocktest~1
nightly/randomtests/blocktest~2
nightly/randomtests/blocktest~3
nightly/randomtests/blocktest~4
```

Out of these four instances of the repeating *blocktest* Task, one or more instances can be selected by itself as follows:

```
% vrun -show -include nightly/randomtests/blocktest~2
nightly/randomtests/blocktest~2
```

or one or more instances can be excluded, leaving the others selected as follows:

```
% vrun -show -include nightly/randomtests/blocktest
    -exclude nightly/randomtests/blocktest~2
nightly/randomtests/blocktest~1
nightly/randomtests/blocktest~3
nightly/randomtests/blocktest~4
```

Also, in “adhoc mode”, an instance identifier other than those defined in the RMDB file can be attached to any Runnable. See “[Adhoc Mode](#)” on page 108 for details.

## Wildcards and Repeating Runnables

Wildcards (either glob-based or regexp-based) can be used to select instance identifiers from a repeating Runnable. A Runnable will repeat when defined with a *repeat* attribute whose value is larger than one or when defined with a *foreach* attribute containing more than one comma/space separated value.

For example, if the repeating *blocktest* Task (discussed in “[Selecting Instances of Repeating Runnables](#)” on page 100) has a *repeat* attribute of 4, the instance identifiers used for the four *blocktest* instances are 1, 2, 3, and 4. Likewise, if the *blocktest* Task had instead been defined with a *foreach* attribute containing the values 2, 4, 12, and 14, then those values are used as the instance identifiers for four instances of the *blocktest* Runnable.

Wildcards can be used to select from among the repeating instances. For example, in the case of a *foreach* Runnable with values 2, 4, 12, and 14, the following command:

```
% vrun -show -include 'nightly/randomtests/blocktest~1*'
nightly/randomtests/blocktest~12
nightly/randomtests/blocktest~14
```

selects only those instance identifiers (that is, *foreach* values) that start with the digit 1.

Note that since instance identifiers can be any string and are not limited to numeric values, the wildcard comparison is lexical.

Using adhoc mode, the user can specify an instance identifier for a repeating Runnable other than those defined in the RMDB file. In the case where a *repeat* count is specified, the adhoc mode context chain can specify a count outside of the normal range of the repeat attribute. In the case of a *foreach* repeating Runnable, adhoc mode allows the user to specify any instance identifier, irrespective of whether or not that identifier is one of those defined in the *foreach* attribute (see “[Adhoc Mode](#)” on page 108).

For example, the following adhoc mode command selects an instance of the *blocktest* Runnable with an instance identifier of 9, even if 9 would not have been used as an instance identifier in the expansion:

```
% vrun -adhoc -show -include nightly/randomtests/blocktest~9  
nightly/randomtests/blocktest~9
```

Even in the case where a given Runnable has no *repeat* or *foreach* attribute specified in the RMDB file, an instance attribute can be added. For example,

```
% vrun -adhoc -show -include nightly~99  
nightly~99/directedtests/...  
nightly~99/randomtests/...  
nightly~99/unittests/...
```

Because the instance identifier is part of the context chain, adhoc mode allows any given Runnable to be included for execution multiple times, even though the Actions resulting from the second execution are likely to be exact duplicates of the first. If a Runnable is intended to be used in this manner, then one or more conditional parameters based on the value of the instance identifier should be used in the Action scripts to distinguish the otherwise redundant executions.

Note that wildcards in the instance identifier string are handled separately from any wildcards in the Runnable name section of the context chain component. In order to select instance identifiers, either explicitly or by wildcard, the instance separator character (~) must be specified explicitly in the component. Otherwise, the matching Runnable(s) are repeat-expanded as defined in the RMDB file.

## Selecting Runnables Based on Results of Previous Run

The contents of the event log from the most recent *vrun* regression run can be used as the basis for Runnable selection. As VRM executes Actions, it records their status in an event log. The user can select, for example, all Actions that passed, all Actions that failed, all Actions that were not run, or any combination of this list. In this case, instead of specifying a list of context paths,

the user can use the *-select* command-line option followed by one or more comma-separated keywords. For example,

```
vrun -select failed,notrun
```

This command selects for execution all those Actions that either failed or did not run. Multiple *-select* options can be specified in the same command. The following command is equivalent to the previous example:

```
vrun -select failed -select  
notrun
```

The value of the *-select* option must be one or more keywords recognized by the *vrun* executable. The recognized status keywords are as follows:

**Table 3-6. Status Keywords**

Keyword	Description
all	All Actions previously selected.
ran	All Actions that executed.
passed	All Actions that passed.
failed	All Actions that failed.
notrun	All Actions that neither passed nor failed.

The keywords can be abbreviated. The command *vrun -select f* is treated the same as *vrun -select failed*. If multiple keywords are combined in a single *-select* option, then the keywords must be separated with commas and/or spaces (remembering that in the case of a *-select* option containing space-separated keywords, the keyword token would have to be escaped in most shells).

The *-select* option behaves just as if the Runnables associated with any failed Actions were listed in *-include* options on the command-line. In other words, if the user types *vrun -select failed*, all the failed tests from the previous run are executed or, if there were no failed tests in the previous run, then *vrun* responds with the usual “Nothing to run” message (as if there were no Runnables listed on the command line).

Note that the event log produces a list of Actions, not Runnables. For each Action selected from the event log, the final component of that Action's context chain (the name of the Action script) is removed and the resulting Runnable context chain(s) are selected for execution.

There is no event-log equivalent to the *-exclude* option, but all the status keywords have complimentary opposites (like *passed*, *notrun* as opposed to *failed*) so this should not present a serious limitation. Also, the *-select* option can be combined with the *-include* and *-exclude* options to build-up the internal list of selected Runnables.

## Context Chains

In many cases, the Actions defined in a given Runnable depend on context from ancestor Runnables (parents or parents-of-parents at any depth). For example, a parameter needed to resolve a command in an Action script can be defined at a higher level in the hierarchy. In such cases, it is possible for the user to supply the necessary parameter value directly on the command line. However, it is more common for the given Runnable to be executed “in context”. This means that when the Runnable is selected, its ancestors are also specified on the command line as follows:

```
vrun nightly/unittests/test123
```

In this case, the only Task Runnable executed is *test123*. But it runs as a member of the Group *unittests* which, in turn, runs as a member of the Group *nightly*. The *preScript* and *postScript* Actions for both *nightly* and *unittests*, if defined, are also executed in their proper sequence. That means that any preprocessing (compilation) or post-processing (coverage merge) defined in the higher level Groups are executed along with the test itself; moreover, any parameters that are defined.

By controlling how much “context” is specified as part of the bareword argument on the command line, the user can control exactly which *preScript* and *postScript* Actions run and how much context is visible from each Action's point-of-view. For example, the following command:

```
vrun unittests/test123
```

runs the same *test123* Task and the *preScript* and *postScript* Actions defined in the *unittests* Group. But the Actions defined in the *nightly* Group will not be executed and any parameters defined in the *nightly* Runnable will not be available for use in parameter expansions (again, values for the missing parameters can always be supplied on the command line). These path-like strings are known in VRM vernacular as “context chains”. The components of a context chain are, for the most part, the names of Runnables defined in the RMDB file.

Note that this notation is required because any given Runnable can be a member of multiple Groups. For example, a test can be a member of a Group called *mode1* and, at the same time, can also be a member of another Group called *mode2*. Because Actions scripts can be parametrized and parameters can be inherited from parent Runnables, this test can, in fact, run slightly different in its two incarnations. Therefore, the context under which a given Runnable is executed is a significant factor in determining exactly which tests run and how.

The trailing component of this path-like argument does not have to be a Task. The user can just as easily run all the random tests in their respective context by typing the following command:

```
vrun nightly/randomtests
```

Except for the trailing component, all components of the context chain must be defined as Groups in the database (in other words, they must be capable of accepting a list of members).

The trailing component can be either a Group or a Task. An error is reported if a Task name is followed by additional context chain components, as in the following:

`nightly/unittests/test123/smoketest`

Any number of bareword arguments can be supplied on the *vrun* command line. The *vrun* application builds a complete picture of which Runnables are selected before executing anything. Overlapping selections do not result in multiple runs of the same test in the same context. Since the combination of the RMDB contents and the command-line options forms a static “configuration” at the time *vrun* is started, it is assumed that any Action can be uniquely defined by its context chain and that two Actions with the same context chain will run identically both times. For example, if a Task *test123* is listed as a member of the *unittests* Group, the following command:

**`vrun nightly/unittests nightly/unittests/test123`**

runs the *execScript* Action associated with the Task Runnable *test123* only once, despite the fact that the entire contents of *unittests* (including *test123*) is also implicitly selected by the first bareword argument. The *vrun* application is smart enough to know that the second bareword argument overlaps the first so this does not result in unnecessarily redundant execution.

Overlapping, however, is based on the full context of the specified Runnables. In the following example command:

**`vrun nightly/unittests unittests/test123`**

the two bareword arguments do not overlap because the second execution of *test123* is under a different context (*unittests*) than it was for the first execution (*nightly/unittests*). In this case, the *execScript* Action associated with the *test123* Runnable is executed twice.

## Including and Excluding Context Subtrees

VRM allows the user to explicitly choose Runnables from a complex regression suite.

The *-include* and *-exclude* options to the *vrun* command (see [page 438](#)) are used to run only select Runnables.

An excluded (deselected) Runnable and its descendant subtree are not considered selected for execution. Overlapping *-include* and *-exclude* options can cancel one another on the basis of the “most specific context chain”. The effect of this cancellation is static and does not depend on the sequence in which the options occur on the command line.

For example, given the following command:

```
% vrun -show -include nightly -exclude nightly/unittests -include
    nightly/unittests/test123
nightly
nightly/directedtests/...
nightly/randomtests/...
nightly/unittests
nightly/unittests/test123
```

The *nightly* Group and all its descendants are selected, except for the *unittests* Group that has explicitly been excluded from the list (*-exclude nightly/unittests*), except for the *test123* Task has been explicitly included (*-include nightly/unittests/test123*). The context chain associated with the *test123* inclusion is more specific than either of the Group context chains so it overrides the exclusion of its parent Group. The exclusion on the *unittests* Group is more specific than the inclusion on the *nightly* Group so it overrides the more general inclusion.

By using a combination of *-include* and *-exclude* options, the user can choose specific blocks of tests for execution. It is also possible to simply list out the full context chain of every leaf-level Task Runnable (picking up the Groups automatically). However, in a complex regression suite, explicitly listing every test is likely to be more onerous than selection by Group.

The selection process can also be done in stages, displaying and saving the results of the selections made thus far in an external file, and only executing the selected list when the combination is just right. Refer to ““Batch Mode Usability Functions” on page 110” for more details.

## Rooted Context Chains

Any Runnable can be used as the root of a context chain.

For example,

```
% vrun -show -include nightly
nightly/directedtests/...
nightly/randomtests/...
nightly/unittests/...
% vrun -show -include unittests
unittests
unittests/test123
unittests/test456
unittests/test789
unittests/mode1tests/...
unittests/mode2tests/...
```

In other words, the first component of a context chain can be a Runnable which, itself, is listed as a member of other Groups. In fact, this would not be unusual if the RMDB file contained multiple self-contained regression suites that are grouped (for convenience) under a single top-level Runnable.

If a given context chain should be treated as if the first component is limited to being a top-level Runnable, a leading slash (/) enforces this limitation as shown:

```
% vrun -show -include /nightly
nightly/directedtests/...
nightly/randomtests/...
nightly/unittests/...
% vrun -show -include /unittests
No matches for pattern 'unittests' in top-level Runnables
```

The definition of a top-level Runnable is a non-base Runnable that is not listed as a member of any Group. In most cases, there will be no more than a handful of top-level Runnables defined in any given RMDB file.

## Globbing in Context Chains

By default, *glob* characters (\*, ?, [<char>...], and {<string>,...}) can be used in any component of a context chain. The matching is done according to the rules specified for the TCL *glob* command.

Each context chain component is treated as a pattern and matched against the list of all possible Runnables that can occur in that position (which can be the list of all members of the Runnable corresponding to the previous component in the chain or, in adhoc mode, the list of all possible Runnables). If the match results in multiple hits, then the context chain is replicated and each match hit is expanded down the chain. Therefore, assuming no other top-level Runnable starts with the letter *n*, the following two commands produce identical results:

```
% vrun -show -include /nightly
nightly/directedtests/...
nightly/randomtests/...
nightly/unittests/...
% vrun -show -include '/n*'
nightly/directedtests/...
nightly/randomtests/...
nightly/unittests/...
```

Note that if the *vrun* command is being typed into a command shell, care must be taken to quote any context chain containing *glob* characters as most command shells detect and expand *glob* characters in tokens before the command-line arguments are passed to the application.

## Using Regular Expressions in Context Chains

While the globbing characters provide a great deal of flexibility in selecting multiple Runnables with one context chain, even more selection flexibility can be obtained by adding the *-regex* option to the *vrun* command line.

The *-regex* option enables the use of regular expressions (as defined in the TCL *regexp* command) instead of simple globbing. For example, the globbed command in the “[Globbing in Context Chains](#)” on page 107 might also have been written as follows:

```
% vrun -regex -show -include '/n.*'  
nightly/directedtests/...  
nightly/randomtests/...  
nightly/unittests/...
```

Note that instead of *n\**, the user must write *n.\** meaning the letter *n* followed by any number of characters. When the *-regex* option is specified, all components of all context chains listed on the command line are matched using regular expressions (as opposed to *glob* expressions). Other than that, processing is identical in the two modes.

The effect of the *-regex* option is global for any given run. Globbed context chains cannot be mixed with regular expression context chains in the same command.

## Adhoc Mode

By default, only context chains that represent actual nodes in the topology of the regression suite defined in the RMDB file are considered valid. For example, the following command:

```
% vrun -show -include nightly/directedtests  
nightly/directedtests/alutest  
nightly/directedtests/cputest  
...
```

only works if the *directedtests* Runnable is listed as a member of the *nightly* Group. If there was a Runnable in the RMDB file called *smoketest* that is not a member of the *nightly* Group, then the following command results in an error by default:

```
% vrun -show -include nightly/smoketest  
No matches for pattern 'smoketest' in Runnable 'nightly'
```

However, there could be a good reason to run the *smoketest* Runnable as though it were a member of the *nightly* Group. In order to create such dynamic relationships on-the-fly, the *-adhoc* option can be used to enable adhoc mode for that specific command. Adhoc mode is global to each run and relaxes the membership validation done on context chains specified on the command line for that run.

When a dynamic parent-child relationship is created in adhoc mode, the group inheritance chain follows the newly created relationship as though it has been defined in the RMDB file. That is, if the previously non-existent *nightly/smoketest* context chain is selected for execution, parameter references in Action scripts associated with the *smoketest* Task that are not satisfied by any definition within the *smoketest* Task itself (or any of its base Runnables) are searched for in *nightly* in its role as “pseudo-parent” for purposes of that specific execution. In other words, when the selected Runnables are executed, group inheritance is determined by the context chain of the Action involved and not by the membership relationships defined in the RMDB file.

Note that even in adhoc mode, each component of a context chain must match a Runnable actually defined in the RMDB file. For example, if there is no Runnable named *fred* in the RMDB file, the following command always results in an error:

```
% vrun -adhoc -show -include nightly/fred
No matches for pattern "fred" in Runnable "nightly"
```

Adhoc mode can be used to string existing Runnables together in parent-child relationships other than those defined in the membership lists in the original RMDB file. But it does not allow new Runnables to be created out of nothing.

Adhoc mode is not used when evaluating floating chains since, given any reasonable number of Groups, there can be a near-infinite number of possible adhoc contexts under which a given Runnable exists. Floating chains are only expanded against the actual topology of the regression suite defined in the RMDB file.

## Batch Mode Usability Functions

The batch mode VRM *vrun* utility supports a number of command-line options aimed at making VRM more accessible to end users. The assumption is that the end-user (who might not be the same person who built-up the RMDB file) does not know which tests and/or parametric options are available to use without studying the RMDB file itself.

The usability options provided fall into the following basic categories:

- Options that allow the user to examine what is contained within the RMDB.
- Options that help build a list of tests and/or other Actions to execute.
- Options that allow the RMDB to prompt for parametric information.
- Options that allow the user to re-execute selected tests by category (rerun failed tests).

The basic use model might include the user performing the following steps:

1. The user views a selected set of Runnables using the *-show* option.
2. The user creates a list of Runnables to be selected using the *-runlist* option.
3. The user causes the selected Runnables to be executed using the *-run* option.
4. The RMDB file causes the user to be prompted for parametric information using the *-ask* option.
5. The user might elect to re-execute some failed tests, possibly with some additional command-line options.

<b>View RMDB Contents .....</b>	<b>110</b>
<b>Build a Runlist of Runnables to Execute .....</b>	<b>112</b>
<b>Executing the Selected Runnables .....</b>	<b>113</b>
<b>Selecting Values for Embedded Options .....</b>	<b>115</b>
<b>Use Model Examples .....</b>	<b>121</b>

## View RMDB Contents

Viewing the contents of an RMDB file primarily means viewing the Runnable hierarchy. The *-show* option requests *vrun* to display some subset of Runnables defined in the RMDB file in a user-friendly format.

The Runnables displayed are those that have been selected to run, either on the current command line or in a saved runlist (see “[Show Actions that will Run](#)” on page 111). Runnables are selected based on other command-line options as discussed in “[Build a Runlist of Runnables to Execute](#)” on page 112.

Basically, the union of all the *-include*, *-exclude*, and *-action* options, as well as the contents of the runlist specified with the *-runlist* option (note that *-runlist* only supports the *-include* and *-exclude* options), and any bareword arguments on the current command line, taken together, constitute the list of selected Runnables. It is this list that determines what is displayed when the *-show* option is also given.

There are two list formats available. The default abbreviated list format shows a sorted list of selected Runnables, along with their immediate children. For simple cases where a single context chain is specified (see “[Context Chains](#)” on page 104), this provides a way to poke-down into the regression suite hierarchy one level at a time. The format is designed to allow quick exploration of the topology without overwhelming the user with data. For example, the command:

**vrun -show /\***

lists all top-level Runnables (defined as those non-base Runnables that are not members of any Group), while the following command:

**vrun -show nightly**

lists all immediate children of the *nightly* Runnable. A full context path can also be supplied as in the following command:

**vrun -show nightly/directedtests**

that shows the immediate children of the *directedtests* Runnable when running as a member of the *nightly* Group. Wildcards are supported as described in “[Wildcards and Repeating Runnables](#)” on page 101.

The other list format is invoked by including the *-all* option along with the *-show* option. The resulting extended list format lists all descendants of the selected Runnables, minus any excluded subtrees. For example, the following command:

**vrun -show -all /\***

generates a complete list of the entire RMDB topology, while:

**vrun -show -all nightly**

lists the children of the *nightly* Runnable and all their descendants recursively.

Note that the functionality of the *-all* option is somewhat different from the typical *-recursive* option supported by other Questa commands. This is because even the *vrun* abbreviated list is really recursive as it simply does not always display down to the leaf-level, in order to avoid swamping out the requested information with irrelevant details.

## Show Actions that will Run

Adding the *-actions* option to the *-show* option causes *vrun* to display a list of Actions (as opposed to Runnables) that run if the selected Runnable(s) are executed. The list of Actions is a

simple list of Action context paths in the approximate order in which they become eligible for execution. For example,

```
vrun -show -actions nightly
```

## Build a Runlist of Runnables to Execute

In some cases, selecting a complex set of Runnables to execute requires more planning and can be easy to construct if spread across several *vrun* commands. In order to create such a complex selection list, the user can add Runnables to and/or prune Runnables from a cumulative runlist before sending the selected Runnables off for execution.

To facilitate this multi-step selection process, a text file containing the currently selected Runnables can be created and maintained across multiple *vrun* invocations. The user supplies the path/name of this text file via the *-runlist* option as follows:

```
vrun -runlist  
<runlist-file> -clear vrun -runlist <runlist-file> <context-path> [...]
```

The basic idea of the runlist file is that VRM reads the file, if specified, so that the internal tree of selected Runnables matches that created by the last *vrun* invocation that specified the same runlist file. Then, the various *-include*, *-exclude*, and *-select* options, as well as any bareword arguments, are processed so as to create a new internal tree of selected Runnables based on the union of the options specified in the current command and the Runnables selected by previous commands. After that, the updated internal list of selected Runnables is rewritten to the runlist file.

The runlist file actually consists of some number of *-include* and *-exclude* options, one per line, each one specifying a context chain representing some part of the defined regression topology. If the *-clear* option is specified, then the *-include* and *-exclude* options from the runlist are not loaded. This effectively clears the runlist of the cumulative data from previous runs and starts the selection process anew. Note that any *-select* commands and/or bareword arguments on the current command line are converted to *-include* and *-exclude* options when the runlist is saved.

The *-runlist* and *-show* options can be used together to show what is selected in the current runlist without actually changing the contents of the list using the following command:

```
vrun -runlist <runlist-file>  
-show
```

The *-actions* option can also be used to show the Actions that run if a given runlist is executed using the following command:

```
vrun -runlist <runlist-file>  
-show -actions
```

As an optimization, the runlist is only written if the: (a) the *-clear* option is specified, or (b) additional *-include*, *-exclude*, and/or *-select* options (or bareword arguments) are encountered that can alter the contents of the runlist.

## Executing the Selected Runnables

By default, when the user specifies a set of Runnables via the *-include*, *-exclude*, and *-select* options or via bareword arguments, VRM builds an execution graph from the selected Runnables and executes the resulting actions.

This behavior is shown in the following example command:

```
vrun -Gparam=value nightly
```

This is because, lacking any directions to the contrary, the *-run* mode (that is, executing the selected Runnables) is the default *vrun* behavior. Certain *vrun* arguments prevent the selected Runnable(s) from executing by default, namely the following options:

- *-dumpgraph* — Dumps the execution graph (debug option).
- *-dumpgraphviz* — Dumps a Graphviz DOT file for the execution graph.
- *-expand* — Displays how arguments expand under *-regex* mode.
- *-runlist* — Loads/saves the cumulative Runnable selection list.
- *-show* — Shows what Runnables/Actions are selected.

When any of these options are specified on the command line, VRM assumes that the purpose of the *vrun* command invocation is to gather information as opposed to actually running anything. For example, the user can execute a series of *vrun* commands including the *-show* option in order to decide what to run. Then, the user can execute some number of *vrun* commands, including the *-runlist* option, in order to build up a complex runlist. Only then, is the user ready to submit the selected Runnables for execution.

The existing *-run* option can be used in conjunction with any of these options to explicitly execute the Actions resulting from expanding the list of currently selected Runnables, said execution to take place after the requested information (if any) is displayed. The Runnables that are selected (by any of the previously mentioned methods) are the ones from which the execution graph is built.

For example, to run the accumulated contents of a runlist as is, the *-run* option should be specified along with the *-runlist* option as follows:

```
vrun -runlist <runlist-file>  
-run
```

In this case, since the *-runlist* option causes the contents of the runlist to be loaded into the internal selected Runnables table, those Runnables are the ones used to build the execution graph. In other words, this command executes the Runnables already selected in the runlist.

In the case where the *-runlist* option, additional *-include* and/or *-exclude* options, and the *-run* option are all listed on the same command line, the *-runlist* options are handled first and the runlist file is updated. After that, the contents of the runlist are executed. This sequence is

guaranteed by the VRM design and is not dependent on the sequence in which the options appear on the command line. Consider, for example, the following command:

```
vrun -runlist run.rl -clear  
-include nightly/test2 -run
```

This command creates a new (empty) runlist called *run.rl*, adds the Runnable *test2* under the *nightly* context as a selected Runnable, and then executes that test. Note that whatever other options are required on the command line to run the selected tests (such as *-g*, *-G*, and so on) they must be provided on the *-runlist/-run* command line as well (as would be the case had the user been naming the Runnables on the command line without the *-runlist* option).

Note that the *-run* option already exists to put *vrun* into regression execution mode, as opposed to message sending mode. However, since regression execution mode is the default mode for the *vrun* implementation anyway, the *-run* option is usually never specified. In this case, since the *-runlist* option (by default) prevents any Actions from being launched, the *-run* command-line option re-asserts the default regression execution mode, which occurs **after** the runlist options are handled and the runlist updated.

It is not an error to specify the *-run* option, even if execution of the selected Runnables would occur by default. Therefore, the *-run* option can be used any time the intention is to execute the selected Runnables.

Note also that if the *-clear* option is not included on the command line, then any contents already in the runlist is also a part of the regression run. That is, the *-run* option simply causes the contents of the runlist resulting from the other specified command-line options to be executed.

## Selecting Values for Embedded Options

Parameters in the RMDB file can be used to select from among one or more options, using conditional and repetitive attributes on the Runnables that make up the regression suite.

Typically, the parameters that implement these embedded “options” are defined in the top-level Runnable. For example,

```
<rmdb>
  <runnable name="nightly" type="group">
    <parameters>
      <parameter name="TOP_MODULE">top</parameter>
      <parameter name="SIMULATION_MODE">batch</parameter>
      <parameter name="NUM_SIM">4</parameter>
      <parameter name="TESTCASE">fair_sequence neg_sqr_sequence
        random_sequence</parameter>
    </parameters>
  </runnable>
  ...
</rmdb>
```

In this case, if the “*TESTCASE*” parameter is used in the *foreach* attribute of a descendant Runnable, then by default all three listed values of “*TESTCASE*” are used to expand the *foreach* Runnable. The “*NUM\_SIM*” parameter is numeric and can be used in a *repeat* attribute. The “*SIMULATION\_MODE*” parameter is an enumerated option that can be set, for example, to either *batch* or *gui*. The user can override the default values of these parameters for any given *vrun* regression run by using the *-G* command-line option. But it is sometimes difficult to recall which parameters were designed to be set in this way and what the acceptable values might be.

What is needed is a way to prompt the user for values for each of these parameters at the start of the regression run. To accomplish this, the *-ask* command-line option can be added to the *vrun* command line. In *-ask* mode, the user is prompted for values of key parameters, based on attributes set in the database (RMDB) file.

When *vrun* expands a parameter on which the *ask* attribute is defined and non-empty, the value of the *ask* attribute is used as the prompt and the user is prompted for a value to use for that parameter. For example, in order to cause the user to be prompted for a value for the “*NUM\_SIM*” parameter, the following can be added to the parameter definition:

```
<parameter name="NUM_SIM" ask="Enter number of random simulations you want
  to run">4</parameter>
```

The current value of the parameter is also displayed as a default:

```
% vrun -ask nightly
Runnable nightly, parameter NUM_SIM:
Enter the number of random simulations you want to run [default: 4]:
```

The user can accept the default by simply pressing the enter key. The default is the value of the parameter as defined in the database. If the user enters only a “return” key at the prompt, then the value of the parameter as defined in the database is used verbatim.

A range of acceptable values can also be specified via the *accept* attribute. The *accept* attribute contains a pattern in a specific format that defines the style and range of acceptable values for the parameter in question. The value of the *accept* attribute can take on any one of the forms shown in [Table 3-7](#).

**Table 3-7. accept Attribute Values**

Value	Meaning
str(re)	String matching the given regular expression (see “ <a href="#">String Parameters</a> ” on page 117).
num(min, max)	Numeric value from a to b inclusive (see “ <a href="#">Numeric Parameters</a> ” on page 117).
oneof(a, b, ...)	Enumerated selection from among a, b, and so on (see “ <a href="#">Enumerated Parameters</a> ” on page 118).
anyof(a, b, ...)	Multiple selection from among a, b, and so on, or from the default value of the parameter (see “ <a href="#">Enumerated Parameters</a> ” on page 118).

Only one of these constructs can be present in the value string of the *accept* attribute for any given parameter. The parentheses are required but the content within the parentheses is, in most cases, optional. The token listed inside the parenthesis are comma/space separated lists similar to those supported elsewhere in the database. Note that though the formal definition shows commas separating values, these are comma/space separated token strings (like many other attributes in the database) and can be separated by any number or combination of commas and/or spaces.

If there is no *accept* attribute for a given parameter or if the value of the *accept* attribute is an empty string, then the parameter in question is assumed to be a free-form text field that can contain any combination of characters.

You can use parameter references within the *ask* and *accept* attributes. For example, refer to the parameter “numparam” below:

```
<rmdb>
  <runnable name="nightly" type="group">
    <parameters>
      <parameter name="MAX">10</parameter>
    </parameters>
    <runnable name="test" type="task" repeat="(%numtests%)">
      <parameters>
        <parameter name="numtests" ask="How many tests should I run?">
          accept="num(2,10)">5</parameter>
        <parameter name="numparam" ask="Pick a number between 1 and (%MAX%)">
          accept="num(1, (%MAX%))">7</parameter>
        </parameters>
      </runnable>
    </rmdb>
```

If an entry made by the user in response to a prompt does not match the pattern specified for the parameter, then the entry is rejected and the prompt is repeated. If the current (default) value of the parameter in the database file does not match the pattern given in the *accept* attribute, then it can still be used as is by pressing the “return” key in response to the prompt. The reason for this is to prevent an infinite loop (and subsequent confusion) that might occur if the default value is not accepted even though it was found hard-coded in the database itself.

## String Parameters

String parameters are the easiest. The *accept* attribute takes the form: *str(re)*, where *re* is a TCL format regular expression. If *re* is the empty string, then any number of characters of any kind are accepted with no limit (except a “newline” character cannot be entered directly).

The regular expression is anchored at both ends. That is, if the string *re* is given in the *str* construct of the *accept* attribute, then the regular expression used to check entries are *^re\$*. The entire entry must match the regular expression in order to be accepted.

## Numeric Parameters

A numeric parameter is assumed to be a single integer number. The *accept* attribute specifies minimum and maximum values for the number specified by the parameter in the form: *num(min, max)*.

With this construct in the *accept* attribute, an entry is only accepted if it is indeed numeric **and** if it falls between the minimum and maximum values specified in the *num* construct, inclusive. One or both of the numeric limits can be blank, in which case that limit is not tested. The various combinations of limit strings and their effect are shown in [Table 3-8](#).

**Table 3-8. Numeric Parameter Limit Strings**

Limit	Check
min,max	Entry is numeric, greater than or equal to min, and less than or equal to max.
min,	Entry is numeric and greater than or equal to min.
,max	Entry is numeric and less than or equal to max.
<blank>	Entry is numeric (no further restrictions).

## Enumerated Parameters

An enumerated parameter allows the user to be presented with a list of possible selections from a list of tests where all tests are run by default but the user can pass the `-ask` command-line option to `vrun`, and be allowed to choose a subset of tests for any given regression run.

In the case of the *oneof* construct (see [Table 3-7](#) on page 116), the set of possible values are listed in the *accept* attribute and exactly one of those values (the default value) should be given as the current value of the parameter itself. The prompting process assumes the entries made by the user in response to the prompts are single values from the list of possible values. As soon as the user's entry matches one of the possible values, the prompting process ends and the matching value is used for subsequent expansions of the given parameter.

Note that the list of values given in the *oneof* construct is optional. But when the *oneof* construct is used in the *accept* attribute, the value of the parameter to which it is attached is assumed to be a single-value parameter. Therefore, the current value of the parameter becomes the only possible selection and it would have already been selected.

The *anyof* construct is similar except that multiple values can be given as the default value in the RMDB file and multiple values can be selected in response to the prompt (see [Example 3-6](#) on page 119). The list of values given in the *anyof* construct is optional. If no values are given, then the list of values is assumed to be the list of comma/space separated tokens in the current value of the parameter itself and the default is to select them all.

In the case of *enum* or *foreach*, the values are listed in a numeric list and selecting (or deselecting) a given value in the list can be done by index number or by content (if the content is numeric, such as a list of seeds, selection/deselection must be performed using the index number of the value in question). Selected values are indicated by an asterisk (\*) (or lack thereof) on the left-hand side of each value. Every value item is numbered. The user can enter either the number associated with the value to select or deselect, or the user can enter the text of the item itself. In the case of a *oneof* parameter, the first match terminates the prompting (as discussed above). In the case of an *anyof* parameter, if an unselected value is entered, then it becomes selected. If a selected value is entered, it is unselected. The process continues until the user is satisfied with the selected collection of values. At that point, the user hits "return" only to enter a null selection. That is the signal to the prompting process that the current selections

should be joined into a single string and used as the semi-final value for the parameter in question (subject to TCL expansion and nested parameter expansion, if required).

If the user's response is not in the range specified in the *accept* attribute, then the prompting process is repeated until a valid value is entered or a blank string is entered to accept the default selection as is. Note that the default is never checked. This prevents an infinite loop from occurring if the value actually specified in the parameter definition in the RMDB file does not meet the criteria specified in the *accept* attribute. The *accept* attribute places a restriction only on the end-user, not on the author of the RMDB file.

[Example 3-6](#) illustrate both kinds of enumerated parameter prompts.

### **Figure 3-6. Enum and Foreach Examples**

#### **Enum Example**

Assume the parameter definition is as follows:

```
<parameter name="SIMULATION_MODE" ask="Enter the simulation mode"
           accept="oneof(batch gui)">batch</parameter>
```

The resulting prompt looks something like the following:

```
% vrun -ask nightly
Runnable nightly, parameter SIMULATION MODE:
  1) batch
  2) gui
  Enter the simulation mode [default 'batch']:
```

#### **Foreach Example**

Assume the parameter definition is as follows:

```
<parameter name="TESTCASE" ask="Enter the testcase(s) you want to run"
           accept="anyof()">fair_sequence neg_sqr_sequence
           random_sequence</parameter>
```

The prompt looks something like the following:

```
% vrun -ask nightly
Runnable nightly, parameter TESTCASE:
  * 1) fair_sequence
  * 2) neg_sqr_sequence
  * 3) random_sequence
  Enter the testcase(s) you want to run:
```

## **Interaction with Parameter Prompts**

The user is prompted for a value for a given parameters only if the parameter in question contains an *ask* attribute **and** the *-ask* command-line option is specified. The prompt for a given parameter is emitted when a reference to that parameter is first used in an Action script, another

parameter definition, or anywhere a parameter reference can be used. This means that no prompt is issued for a parameter that is never used. It also means that parameters that must be expanded to resolve the expansion graph is prompted for before the regression run physically starts. Other parameters can be prompted after the regression run has started.

A queue is maintained so that the same parameter in the same Runnable are not prompted twice (the value supplied when the parameter is first prompted is reused each time that parameter is expanded).

Parameter prompts occur in the order in which the parameters are used. If a given parameter is **not** used anywhere in the regression suite, then the user is not prompted to supply a value for that parameter. Each answer is cached by the parameter name and Runnable name. This way, if the “*NUM\_SIM*” parameter in the *nightly* Runnable is used more than once (as a result of group or base inheritance), then the user is asked only once for a value for this parameter. On the other hand, if a descendant Runnable also defines a “*NUM\_SIM*” parameter and that parameter also contains an *ask* attribute, then the user is able to supply separate values for these two independent parameters.

If the current (default) value of the parameter in question contains nested parameter references, those are resolved before the prompt so that the default value shown is the fully expanded value of that parameter. However, if the parameter in question is a TCL-type parameter, then the TCL expansion occurs after the prompting yields a new value for the parameter and the user-supplied value is passed to the TCL interpreter for expansion. Further, if the value supplied by the user during the prompting process itself contains a nested parameter, then that parameter is also expanded before the final parameter value is used in the Action script, and so on.

## Use Model Examples

The following examples illustrate the suggested use model for these features.

<b>Example of Front-end Runnable Selection .....</b>	<b>121</b>
<b>Combined Selection and Prompting Example .....</b>	<b>122</b>

### Example of Front-end Runnable Selection

Assume that when the user first starts to use the given RMDB file, the user has no idea what exists inside. A quick check of the top-level Runnables:

```
% vrun -show /*  
nightly  
unitests
```

tells the user there is probably a nightly regression suite and one or more unit tests. The user probes further:

```
% vrun -show nightly  
nightly  
nightly/directedtests  
nightly/randomtests
```

At this point, the user knows that just running *nightly/directedtests* runs all the directed tests but maybe the user does not want to run all the directed tests. The user decides to start a runlist so it can be pruned later:

```
% vrun -runlist temp.rl -include nightly/directedtests  
nightly/directedtests  
nightly/directedtests/mode1  
nightly/directedtests/mode2
```

Because *directedtests* is now selected for running (in the file *temp.rl*), the immediate children of that Runnable are also included in the list. Suppose the user only wants to test *mode2*. The user can exclude the *mode1* tests with the following command:

```
% vrun -runlist temp.rl -exclude nightly/directedtests/mode1  
nightly/directedtests  
nightly/directedtests/mode2
```

Note that the new abbreviated list (output whenever the *-runlist* option is specified) contains only the *mode2* child of *directedtests*. Note also that whenever a nested Runnable is selected, the Group(s) implied by the context chain of the nested Runnable are also selected.

Assume that the user is now satisfied and wants to run, the user can launch the tests with the following command:

**vrun -runlist temp.rl -run**

Note that this example assumes the RMDB file is *default.rmdb* (or that the path to the RMDB is in the *MTI\_VRUN\_DB* environment variable) and the user is happy with the default values for all other options. All other applicable options, like *-rmdb*, *-vrmdata*, and so on, are also available even when *-runlist* or *-show* are used, save only that *-runlist* and *-show* only perform enough front-end processing to understand the execution graph and that no Actions are actually launched unless specifically requested.

## Combined Selection and Prompting Example

This example takes the four parameters, annotated with *ask* parameters, and then runs through a typical *vrun* initialization process. Following is the annotated RMDB fragment:

```
<rmdb>
  <runnable name="nightly" type="group">
    <parameters>
      <parameter name="TOP_MODULE" ask="Enter the name of the top
         module">top</parameter>
      <parameter name="SIMULATION_MODE" ask="Enter the simulation
         mode" accept="oneof(batch gui)">batch</parameter>
      <parameter name="NUM_SIM" ask="Enter the number of random
         simulations you want to run" accept="num(1,9)>4</parameter>
      <parameter name="TESTCASE" ask="Enter the testcase(s) you want
         to run" accept="anyof()">sequence_fair sequence_neg_sqr
         sequence_random</parameter>
    </parameters>
  </runnable>
  ...</rmdb>
```

Based on the above RMDB, the following session can ensue (note in this example the user's responses are also shown):

```
% vrun -show '/*'
nightly
unitests
% vrun nightly -ask
Runnable nightly, parameter TOP_MODULE:
  Enter the name of the top module: top
Runnable nightly, parameter SIMULATION MODE:
  1) batch
  2) gui
  Enter the simulation mode [default: 1]: (...presses ENTER to accept
    the default...)
Runnable nightly, parameter NUM_SIM:
  Enter the number of random simulations you want to run [default: 4]: 1
Runnable nightly, parameter TESTCASE:
  1) fair_sequence
  2) neg_sqr_sequence
  3) random_sequence
  Enter the testcase(s) you want to run [default: 1, 2, 3]: 1
Terminating ASK mode, thanks for your help.
Run Monitor daemon listening on '60976@dvtvnc1'...
...continues on with the test...
```

## VRM Email Functionality

This functionality supports both automated status messages sent at the end of a regression run and ad-hoc messages initiated from user-definable TCL procedures at any point during the regression run.

To generate the end-of-regression status summaries you would call a TCL utility procedure from within any user-definable TCL procedure in order to simplify the construction of the body of a status message.

However, if the default status message does not meet your requirements, you can supply a manually-generated message instead.

<b>VRM Email Use Model .....</b>	<b>123</b>
<b>VRM Email Settings and Parameters .....</b>	<b>124</b>
<b>Manually Generated VRM Email .....</b>	<b>126</b>
<b>Composing Email Settings from Various Sources .....</b>	<b>127</b>

## VRM Email Use Model

You can generate an email in one of two use models, either an automated status message or one that is manually generated.

- Automated — Add a minimal set of pre-defined parameters somewhere in the RMDB to enable VRM to send a pre-formatted status email to one or more recipients at the end of the regression run.
- Manually generated — Call a TCL utility procedure from any user-defined procedure to send an email at any point during the regression run.

Both methods make use of sensible defaults for many of the email settings, while those that must be overridden may be done so via RMDB parameters or arguments passed to the relevant TCL utility procedures.

Either model consists of two steps:

- Gathering of relevant email settings.
- Composing and sending the actual message.

which are encapsulated in a single procedure that is called from within the [RegressionCompleted](#) user-definable procedure. This procedure checks for the relevant email settings and, if the required settings are found, composes and sends an email message containing the current status of the regression run. Any of these procedures may be called at any time in order to send additional messages while the regression is still running.

## VRM Email Settings and Parameters

The settings for an email message are compiled from a combination of default settings, RMDB parameters, and arguments passed to certain TCL utility procedures.

When you send an email message, you must take into account a number of settings. Some settings are required, some have default values, and some must be specified explicitly.

Table 3-9 lists the settings. Required settings are indicated by a Yes in the Req. column.

**Table 3-9. VRM Email Settings**

Setting RMDB Parameter	Req.	Default	Description
message EMAIL_MESSAGE	No	No message	The content of the body of the email message.
originator EMAIL_ORIGINATOR	No	vrun@<hostname>	<p>The email address from which the email message will appear to originate.</p> <p>This value is not checked for validity. However, depending on the filtering applied to the receiving email address(es), the value of the “From:” address may cause the email message to be dropped or marked as spam.</p>
recipients EMAIL_RECIPIENTS	Yes	none	A TCL list of email addresses to which the email message will be sent. This may be a comma- or space-separated list.
servers EMAIL_SERVERS	Yes	none	<p>A TCL list of SMTP servers (IP address or FQDN) which can forward the email message. This may be a comma- or space-separated list.</p> <p>If you specify multiple servers, the first one that accepts the message will be used and the others will be ignored.</p> <p>The listed SMTP servers are assumed to accept email input without credentials.</p>

**Table 3-9. VRM Email Settings (cont.)**

<b>Setting RMDB Parameter</b>	<b>Req.</b>	<b>Default</b>	<b>Description</b>
sections EMAIL_SECTIONS	No	none	A TCL list of zero or more keywords indicating the summary report sections to include in the email message. This may be a comma- or space-separated list.  Refer to <a href="#">VRM Email Settings — section Keywords</a> .
subject EMAIL SUBJECT	No	VRM Status Summary for <timestamp>	The content of the subject header line of the email message

The body of the email message will be based on a combination of the message setting and the sections setting. In the case of an email message initiated manually from a user-definable procedure, only the string supplied in the message setting will be used. In the case of an email message initiated automatically at the end of a regression run (or via the [AutoEmail](#) utility procedure), the string supplied in the message setting will be followed by the status summary sections specified in the sections setting (with a single intervening blank line). The keywords recognized in the sections setting are shown in the following table.

**Table 3-10. VRM Email Settings — section Keywords**

<b>section Keyword</b>	<b>Description</b>
all	Alias for specifying the keywords: exe, cpl, sim and apf.
apf	Pass/Fail status.
cpl	Completion status.
exe	Execution status.
gen	Action script generation status.  This is not included when you use the “all” keyword.  This keyword is a variant of the exe status summary which reports on scripts generated rather than on scripts executed. This is used when you specify the -noexec argument on the vrun command line, indicating that the generated scripts are intentionally not to be executed.
sim	Test, or valid UCDB, status

## Expanding Global Parameters

Parameters in an RMDB file are always defined within a specific Runnable. In general, parameter expansion happens from the point-of-view of a specific Runnable instance. However, the most obvious place to trigger the sending of an end-of-regression status email message is

from within the [RegressionCompleted](#) user-definable procedure and that procedure is called outside of any Runnable context.

In order to allow the email sending procedures to find the email-related parameters, the RMDB API enables a search for parameters by name across all Runnables. If multiple parameters of the same name are defined in two or more different Runnables, the one which occurs first in document order is selected.

Because the global parameter search is not associated with a Runnable context, any parameter references embedded in a parameter found through global search will be expanded based on another global search.

## Manually Generated VRM Email

Use the `SendEmailMessage` utility procedure, which uses the standard Simple Mail Transport Protocol (SMTP) to compose and send an email message to one or more recipients. you pass the procedure a list of key-value pairs which override the built-in defaults.

You may call this utility to send an email at any time during the regression run.

For example, If you use the following procedure to override the [RegressionCompleted](#) user-definable procedure in the RMDB file, an email message will be sent as soon as the regression completes.

```
proc RegressionCompleted {userdata} {
    upvar $userdata data
    # This part is from the default RegressionCompleted procedure
    if {[isChatty]} {
        logDebug "Regression finished at [RightNow]... sending email"
    }

    # This part sends an email message at the end of the regression

    if {[catch {SendEmailMessage [list \
        originator ${::env(USER)}@mentor.com \
        servers      [list mail.company.com] \
        recipients   [list ${::env(USER)}@mentor.com] \
        message      [GenerateStatusSummary [list exe cpl sim apf]] \
    ]} errmsg]} {
        logStuff "Email error: $errmsg"
    }
}
```

In this example, the originator and recipient email addresses are derived from the user name stored in the environment. An SMTP server for “company” is used to send the message and the default subject line is used as-is, because it is not specified.

The message itself is constructed using the [GenerateStatusSummary](#) utility procedure, where the four arguments cause the procedure to compile a message string consisting of the same summary sections which appear at the end of the vrun log output.

In the event of an error or other anomaly, the resulting message is emitted to the vrun log output.

## Composing Email Settings from Various Sources

You can compose an email from various sources through the use settings and utility procedures.

### Procedure

1. Call the [GetDefaultEmailParameters](#) utility procedure to generate a list of key/value pairs containing the default values for all email-related settings.
2. Pass that list to the [GetEmailParameters](#) utility procedure to search for settings which may be defined in parameters in the RMDB file.

The values found in the RMDB file will override the values passed to the [GetEmailParameters](#) utility procedure and a list of the combined key/value pairs will be returned.

3. Use the calling procedure to override any of the settings as necessary. The easiest way to do that would be to:
  - a. Build a TCL array (hash) from the list of key/value pairs.
  - b. Replace the values that need to be changed.
  - c. Convert the TCL array back into a list of key/value pairs.
4. Pass the final list of key/value pairs (the email settings) to the [SendEmailMessage](#) utility procedure to generate and send the email.

### Results

The [AutoEmail](#) utility procedure combines these steps into a single procedure call for those cases where such complexity is unnecessary. In the most common case, where an email message containing the final status of the regression is to be sent at the end of the regression run, only the RMDB parameters need be defined — no TCL code is necessary. The other procedures are only there to allow customization in the not-so-common case.

### Examples

#### Automatic Summary Message

The ability to automatically send a message at the end of the regression run is built into the VRM system.

To enable this, you must define certain parameters in the RMDB file. The location of the parameters is irrelevant since VRM will search the entire RMDB, using the first matching parameter that it finds for each setting. For example, the following RMDB file would send an email similar to that sent by the example in the section “[Manually Generated VRM Email](#)”.

```
<?xml version="1.0" ?>      <rmdb version="1.0" toprunnables="nightly">
    <runnable name="nightly" type="group">
        <parameters>
            <parameter name="username" type="tcl">$::env(USER)</parameter>
            <!-- the following parameters are used by the AutoEmail function -->
            <parameter name="EMAIL_RECIPIENTS">(%username%)@mentor.com</parameter>
            <parameter name="EMAIL_SERVERS">mail-na.mentor.org.com</parameter>
            <parameter name="EMAIL_SECTIONS">exe cpl sim apf</parameter>
            <parameter name="EMAIL SUBJECT">This is an automatic message</parameter>
        </parameters>
        <members>
            <member>test1</member>
        </members>
    </runnable>
    <runnable name="test1" type="task">
        <execScript launch="vsim">
            <command>runmgr::rm_message "Greetings from (%ACTION%) "</command>
        </execScript>
    </runnable>
</rmdb>
```

At a minimum, the EMAIL\_SERVERS parameter must contain the IP address or Fully Qualified Domain Name (FQDN) of at least one valid SMTP server and the EMAIL\_RECIPIENTS parameter must contain the email address of at least one valid recipient. You may omit, or leave blank, all other parameters. Of the omitted or blank parameters, some will be given default values, as defined in “[Table 3-9](#)”.

In the example, you will notice that:

- The EMAIL\_RECIPIENTS parameter makes use of the user name stored in the environment, in this case via a TCL-type parameter called username.
- The EMAIL SECTIONS parameter defines the status summary sections to be included in the email message. Note that when this parameter is defined and non-blank, the value of the parameter is assumed to consist of status summary keywords to be passed to the GenerateStatusSummary utility procedure. The output of the GenerateStatusSummary utility procedure is then appended to the body of the email message.

### Customized Automatic Summary Message

You can construct a custom message body of a VRM email to be sent out at the end of the regression.

When you define the EMAIL\_MESSAGE parameter as non-blank, the value of that parameter is included as the first part of the message body. Then, if you define the EMAIL\_SECTIONS parameter as non-blank, the specified status summary sections will be appended to the end of the message with a single intervening blank line. This results in the value of the EMAIL\_MESSAGE parameter becoming a heading for the status summary sections that follow).

```
<?xml version="1.0" ?>
<rmdb version="1.0" toprunnables="nightly">
  <runnable name="nightly" type="group">
    <parameters>
      ...
      <parameter name="EMAIL_MESSAGE">The regression has finished...</parameter>
      <parameter name="EMAIL_SECTIONS">exe cpl sim apf</parameter>
      ...
    </parameters>
    ...
  </runnable>
  ...
</rmdb>
```

You can construct a more customized message through the use of a TCL procedure defined in a usertcl element.

```
<?xml version="1.0" ?>
<rmdb version="1.0" toprunnables="nightly" loadtcl="default">
  <runnable name="nightly" type="group">
    <parameters>
      ...
      <parameter name="EMAIL_MESSAGE" type="tcl">[ComposeEmailMessage]</parameter>
      ...
    </parameters>
    ...
  </runnable>
  ...
  <usertcl name="default">
    proc ComposeEmailMessage {} {
      set message [list]

      lappend message "This is the regression status as of [RightNow] :"
      lappend message ""
      lappend message [GenerateSummaryStatus [list exe cpl xim apf]]

      return [join $message "\n"]
    }
  </usertcl>
</rmdb>
```

In this case, the usertcl procedure calls the GenerateStatusSummary procedure to compose the major part of the message body. However, you could use just about any available data and/or procedures to compose the message body. The procedure should return a string with embedded newline characters to separate the lines in the message.

## Regression Run Modification

---

Use the vrun -modify command line to modify select parameters and attributes of a regression run after the regression has started.

**Parameter and Attribute Modification .....** [130](#)

**Modifying an Active Regression from the Command Line .....** [131](#)

## Parameter and Attribute Modification

You can modify select parameters and attributes during an active regression run.

The attributes and parameters you can modify are defined as follows:

- Operational Attributes

The maxrunning, maxclubbing, maxarray, and window attributes of the method element.

Changes to these attributes take effect immediately upon completion of the vrun -modify command.

Values for these attributes always take precedence over those originally specified in the method element of the RMDB.

- RMDB Parameters

parameters that you would typically control with the -G argument to vrun.

- Context-specific Parameters — This is when you use the <context> argument to the vrun -modify command.

- Global Parameters — This is when you do not use the <context> argument.

Changes of this form require VRM to regenerate internal script files that have been already generated before any actions, with which those files are associated, are launched.

Changes to parameters affect the operational attributes indirectly, therefore all named queues are reset such that they can be reevaluated according to the values of the eligible actions.

---

 **Note**

NOTE: It is suggested that you not change parameters that control post execution analysis

---

# Modifying an Active Regression from the Command Line

Use the vrun -modify command line to modify an active regression run.

## Prerequisites

- Have an active regression run.

## Procedure

1. Determine which parameters or attributes you need to modify and craft the necessary <keyword>=<value> pair.
2. Determine whether you need to modify the default arguments to the vrun -modify command line.
  - a. (optional) Determine which VRMDATA directory you want to modify, if different from the default.
  - b. (optional) Determine which status event log you want to analyze, if you are running multiple regression runs from the same VRMDATA directory.
3. Invoke a new vrun command with the -modify argument and any supporting arguments.

If there is a problem with your command line, vrun issues an error response in the log output

## Results

- vrun regenerates any Actions that are impacted by the modification and are queued for launch.
- Actions already launched are not affected by the command.
- Original regression run continues with newly modified Actions.

# Override Parameter Values from Command Line

---

VRM supports `-g` (parameter default) and `-G` (parameter override) options. These options have the same syntax and similar behavior to options of the same name in Questa `vsim`. The exact syntax of these options is as follows:

```
-G<name>=<value> // note that equal sign (=) is required  
-g<name>=<value> // note that equal sign (=) is required
```

There must be no spaces between any of the components of either option. The *name* is the name of the affected parameter. The *value* is the value to be assigned to that parameter. Parameters can be overridden with values containing an equal sign (=).

The order in which the parameters are applied is not defined. The options, however, are stored in a hash prior to being applied to the execution graph. If two or more `-g` options (or two or more `-G` options) specify the same *name* component, then the one appearing last on the command line overwrites the first option.

The `-g` (lower-case) option specifies a value for a given parameter that is used if no other value is defined for that parameter in the database. In other words, a `-g` option provides a default value that can be overridden by a parameter definition in the Runnable on whose behalf the parameter is being expanded or in any Runnable on the base or group inheritance chains starting from said Runnable. The only value with a lower priority than the `-g` option would be a reference-specific default named in the parameter reference itself.

The `-G` (upper-case) option specifies a value for a given parameter that is used irrespective of any other value present in the database. In other words, the `-G` option provides an override value for a parameter that is used in place of any definition specified in the Runnable under examination. If a parameter with the *export* or *ask* attribute is overridden by the `-G` command-line option, this override only affects the value of the parameter and the *export* or *ask* metadata is preserved.

Note that either the `-g` or the `-G` option can provide a context to limit the effective geography to which the override/default applies. A parameter override applied to a given Runnable can apply to other Runnables as well, if the Runnable to which the override was applied is encountered as part of a base/group inheritance search starting from the other Runnable. In all cases, a more local definition takes priority over an override provided at a higher level in the regression suite topology. That is, a matching local parameter definition always has priority over a `-G` override that is “attached” to the regression suite tree at a higher (less local) point in the hierarchy.

The algorithm, in theory, works in two layers (implementation can be different for performance reasons). The first layer walks the inheritance chains as usual (parents first, then base elements). At each Runnable, first check for an applicable command-line override (`-G`) option whose context matches that specific Runnable (using the context partial-match algorithm, see

page 136). If a *-G* override is found, then the search terminates and the override value is used. If not, the list of parameters defined in the Runnable itself is consulted. If this does not result in a match, then the search moves on to the next Runnable following the usual inheritance search order.

If, after searching all the Runnables from which a parameter value can be inherited, a defined value is not found, then the default (*-g*) option list is consulted. This search again follows the standard inheritance rules so that a default supplied to any Runnable on the base or group inheritance chains can be used to satisfy the search, but only if there is not a more “local” default setting available. If the *-g* search also fails to come up with a defined value for the parameter in question, then the reference-specific default defined in the parameter reference (if any) is used and, failing that, an error is generated and the parameter reference is replaced with an empty string.

The important thing to remember is that the inheritance search order is always dominant. A more “local” definition (closer to the Runnable on whose behalf the search is being performed) always takes precedence over more “distant” inherited definitions, per the search rules, regardless of whether that “distant” definition is an override or a normal parameter definition. For each Runnable searched, an override is used if one is provided. Otherwise, the local definition is used. And, failing that, the search continues to more “distant” Runnables along the inheritance chains. That means the override options are consulted independently for each Runnable, rather than collectively before the normal search begins.

The *-g* defaults, on the other hand, are not consulted at all if any other definition (local or inherited) exists. That means exhausting the entire inheritance “network” (base Runnables and group ancestors) before considering a look at the *-g* default options. That is the theory, but in practice there can be some benefit in collecting the default (*-g*) values, if any, during the initial search in order to save walking the inheritance network a second time.

<b>Limiting Parameter Overrides to a Specific Context .....</b>	<b>133</b>
<b>Context Partial-Match Algorithm .....</b>	<b>136</b>

## **Limiting Parameter Overrides to a Specific Context**

Both the *-g* and *-G* options are composed of the following components (the *-G* option is used in this example but the same pattern and rules apply to the *-g* option):

*-G [<context>] <key>=<value>*

The *value* is the string value to be used as the new definition of the parameter in question. The *key* is the name of the parameter being defined and is also the string used in a parameter reference to identify which *key/value* pair to use for the text substitution. These two components are required, at a minimum, in order to define the *key/value* pair that makes up a parameter definition. Since a *key* is always required, the final component of the (possibly path-like) string between the *-g* or *-G* and the equal sign is always assumed to be the name of the parameter being controlled.

The optional “context” part of the format determines where the parameter override/default is to be applied. If a context is not provided, as in the following example:

```
-Gfred=parrot
```

then the specified value (string *parrot*) is used to resolve any reference to the parameter *fred* no matter where it occurs in the database. This is a global override, as it does not specify a particular context.

Following is a slightly more specific variant:

```
-Gtask1/fred=parrot
```

In this case, the scope of the override is limited to the Runnable *task1*. So any time an instance of the Runnable *task1* is searched for the parameter *fred*, the search terminates and the override value (*parrot*) is used to replace the parameter reference in question. Note that the “context” is the single Runnable name *task1*. The final *fred* component is assumed to be the name of the parameter and is not part of the context string.

A context chain (see [page 104](#)) can also be specified as in the following example:

```
-Ggroup1/task1/fred=parrot
```

In this case, the override is only effective for instances of the Runnable *task1* that is invoked as members of the Group *group1*. This example does not place any limits on how the Group *group1* itself came to be invoked. It can be a top-level Runnable or it can be invoked as a member of a higher level Group.

If, on the other hand, the override is to apply only to instances of the Runnable *task1* that is invoked by the top-level Group *group1*, the following syntax is used:

```
-G/group1/task1/fred=parrot
```

The leading slash (/) on the context makes this a “rooted” context that only matches if the left-most component of the context path was, in fact, invoked as a top-level Runnable from the *vrun* command line. In the case where the *group1* Group is invoked as a member of another Group (for example, */top/group1/task1*), this override does not apply, as the *group1* Group is not invoked as a top-level Runnable.

In the case of non-overlapping override/default options applied to different parts of the regression suite topology, the more local (that is, more specific) option already has priority by virtue of the inheritance search order rules. No other priority rule is needed.

In the case of override/default options with overlapping context strings (for example, *-G/top/group1/task1/fred=123*, *-Gtask1/fred=456*, and *-Gfred=789*), the more specific context has priority. A global override (one which specifies no context) is considered to be the least specific so context-specific overrides always have priority over global overrides. In addition, rooted contexts are considered more specific than non-rooted contexts. This allows the user to provide

a general override for a parameter, to be used for the majority of the regression suite, and still be able to provide a different override value for that same parameter for Runnables in specific parts of the regression suite topology.

For example, consider the following group of Runnables:

```

<runnable name="top" type="group">
  <parameters>
    <parameter name="fred">666</parameter>
  </parameters>
  <members>
    <member>group1</member>
    <member>group2</member>
  </members>
  ...
</runnable>
<runnable name="group1" type="group">
  <members>
    <member>task1</member>
  </members>
  ...
</runnable>
<runnable name="group2" type="group">
  <members>
    <member>task1</member>
    <member>task2</member>
  </members>
  ...
</runnable>
<runnable name="task1" type="task">
  ...
</runnable>
<runnable name="task2" type="task">
  ...
</runnable>

```

A user can, from the command line, put a global override on a given parameter, an override of that same parameter for the *task1* Task, and an override on that same parameter for the *task1* Task when invoked as part of the *group1* Group. The command-line options required to set these overrides are as follows:

```

-Gfred=123 -Gtask1/fred=456
-Ggroup1/task1/fred=789

```

In this case, for the instance of the Task Runnable *task1* that is called from the Group *group1*, the value of the parameter *fred* resolves to 789 because of the more local (that is, more specific) override given on the command line. For the *task1* Task invoked by the Group *group2*, this same override does not apply because the context does not match. In this case *fred* resolves to 456 because of the override on *fred* as part of the *task1* Runnable. For the *task2* Runnable, however, neither of the context-specific overrides match so the global override on the parameter *fred* causes it to resolve to 123. In no case is the value 666 found in the top Runnable used, even though it is on the group inheritance chain. The reason is that this Runnable was never

consulted, the parameter *fred* having been resolved at a more local level (the global override *I23* is “more local” than the parameter definition in the top Runnable, even for scripts being run on behalf of the top Runnable).

However, had the command line been as follows:

```
-gfred=123 -gtask1/fred=456
-ggroup1/task1/fred=789
```

none of the default values given on the command line would be used because the parameter *fred* would always resolve to *666* due to the definition found in the top Runnable (note that, in this example, all instances of the *task1* and *task2* Runnables are descendants of the top Group Runnable).

In the event multiple override/default command-line options are supplied for the same parameter in the same context on the same command line, as in the following example:

```
-Gtask1/fred=234 nightly1 -Gtask1/fred=567
nightly2
```

priority is given to the command-line option occurring last on the command line. In other words, the override/default options are processed in the order given and the second override value (in this example) overwrites the first. All the command-line options are processed before execution begins so the second override value will apply to the execution of both the *nightly1* and the *nightly2* top-level Runnables, irrespective of the order in which the top-level Runnables appear with respect to the override/default options on the same command line.

## Context Partial-Match Algorithm

As a result of base and group inheritance, the visibility of any given VRM parameter is not limited to a specific place in the topology like Verilog parameters. A single parameter definition can effect a significant chunk of database geography depending on where in the database it is defined.

Suppose, for example, that one defines a complex multi-level tree of Groups and Tasks and the parameter name *fred* does not occur anywhere in the database. If a parameter named *fred* were introduced into the middle of that tree, that parameter would be visible from the Runnable to which it was added and from every Runnable that refers to the Runnable containing the *fred* parameter as a base Runnable and to every member of every member (that is, every descendant) of the Runnable containing the *fred* parameter. Inheritance makes that so. Parameters are defined, not assigned. And the process of expanding a parameter reference involves a two-dimensional search using a well-defined set of rules for determining which of several possible definitions for the parameter under expansion is to be used in place of the parameter reference itself.

The same is also true of parameters “defined” with the *-g* and/or *-G* command-line options. Each of these options “binds” a parameter definition onto one or more specified Runnables.

Any other Runnable from which the affected Runnable is visible (via the inheritance search chains) will also see that new parameter. If a given parameter override is intended to be effective over a specific sub-hierarchy of Runnables, and if that sub-hierarchy of Runnables shares a common Group somewhere in the regression suite hierarchy, then the override can be accomplished by binding a single override parameter onto the common Group Runnable at the top of the sub-hierarchy. This assumes that another parameter definition of the same name does not occur elsewhere in the sub-hierarchy of Runnables.

For example, given a context chain (see [page 104](#)) something like *top/group1/task1* involving three Runnables (Group *top*, Group *group1*, and Task *task1*), the three main variations of context are as follows:

```
/top, /top/group1, /top/group1/task1
```

These chains are “rooted” and only apply to the example chain if *top* is invoked as a top-level Runnable. The difference between the three is a matter of scope only. The */top* context makes the override visible from any of the three Runnables whereas */top/group1/task1* makes it visible only from the *task1* Runnable (but only if the Runnable is invoked as a member of the Group *group1* that is invoked as a member of the Group *top*).

Following are simple Runnable names:

```
top, group1, task1
```

A context such as this is used to add a parameter definition to a given Runnable no matter where it was used. For example, if *task1* is included as a member of multiple Groups, the context */top/group1/task1* would only match when *task1* is invoked as a member of the Group *group1* that is, itself, invoked as a member of the top-level Group *top*. A context of *task1*, on the other hand, would match (and thus define the override for) every invocation of the *task1* Runnable in the execution graph.

Following are Runnable hybrids:

```
top/group1, group1/task1
```

The first one, for example, matches any instance of the *group1* Runnable that is invoked as a member of the Group *top*. In this case, it does not matter whether *top* is called as a top-level Runnable or not. It also does not matter whether *top* is named as a member of a single Group or a dozen. It matches any context chain that ends in *top/group1*, and its effect (because of the effects of inheritance) are visible to all descendants of *group1*, which means the override is visible from any context chain that, when written out in the usual notation, contains *top/group1* as a substring.

In the case of repeating Runnables, a context chain is assumed to match a Runnable if it would have matched in the non-repeating case. It is not possible to override a parameter in only some instances of a repeating Runnable without overriding that parameter in all instances of the repeating Runnable.

Using the previous example, if *task1* repeats twice, the expanded context chains for the two invocations are *top/group1/task1~1* and *top/group1/task1~2*. The *~1* and *~2* instance-specific strings are added at runtime when the execution graph is built and do not appear in the database. To determine whether a given context matches, these instance-specific strings will be removed from the context chain so that both repeating instances will match the context string *top/group1/task1*.

Parameter override/default values added via the *-g* and *-G* command-line options are considered to have been defined within those Runnables whose names and/or contexts match those of the command-line option. However, other Runnables can still pick up the override/default value if the Runnable to which the override/default is applied is on that Runnable's base or group inheritance chains. The rules governing the interaction between the *-g* and *-G* options and inheritance are specified in “[Inheritance, Overrides, and Search Order](#)” on page 271.

## Access RMDB API via the vrun Command Line

The *-apicmd* option can be used to issue a command to the database (RMDB) API. If this option appears on the command line, all options and arguments following it are consumed as part of the API command.

Only one such command can be issued to the API for any given *vrun* invocation. The result of the command appears on the standard output of the *vrun* process.

The following example causes *vrun* to emit a list of all Groups defined in the *my.rmdb* RMDB database file:

```
vrun -rmdb my.rmdb -apicmd  
groups
```

The following (slightly more complex) example fetches the value of the *ucdbfile* parameter from the Runnable called *mytest* (from the same RMDB file):

```
vrun -rmdb my.rmdb -apicmd  
Runnable mytest param ucdbfile
```

The API commands are documented in the Appendix entitled “[VRM Database \(RMDB\) API](#)” on page 573.

# Chapter 4

## Graphical User Interface (GUI)

---

This chapter describes the VRM graphical user interface.

<b>GUI Environment .....</b>	<b>139</b>
<b>VRM Cockpit Window .....</b>	<b>142</b>
<b>VRM Results Window.....</b>	<b>159</b>
<b>VRM Project File and Configurations.....</b>	<b>169</b>
<b>Invoke Batch-mode Regressions During a GUI Session.....</b>	<b>169</b>
<b>Initial Start-up and Project Management.....</b>	<b>171</b>
<b>Start-up and the toprunnables Attribute.....</b>	<b>172</b>
<b>Relationship Between Configurations and VRM Data Directories.....</b>	<b>173</b>
<b>Menus .....</b>	<b>175</b>
<b>GUI Use Model.....</b>	<b>186</b>
<b>Effect of Projects and Configurations on Batch-mode VRM Regressions.....</b>	<b>207</b>
<b>Create RMDB File from Existing UCDB File.....</b>	<b>209</b>

## GUI Environment

Invoke Questa SIM Verification Run Manager in GUI mode by typing the following:

```
vrun -gui
```

This invokes the GUI shown in [Figure 4-1](#).

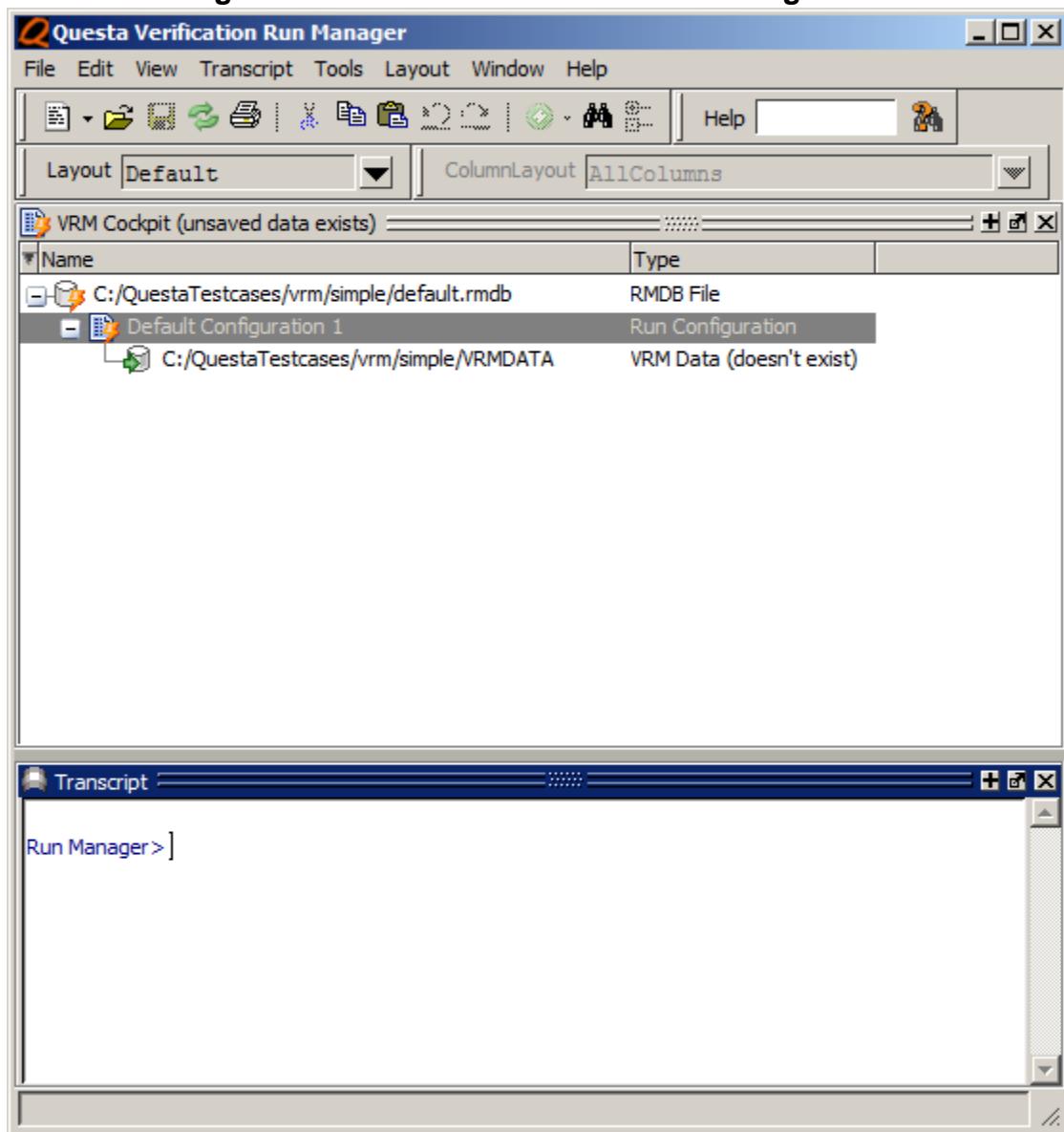
---

### Note

 The screen captures in this section are made as if you are invoking from the `<install_dir>/examples/vrm/simple/` directory using the `default.rmdb` RMDB file.

---

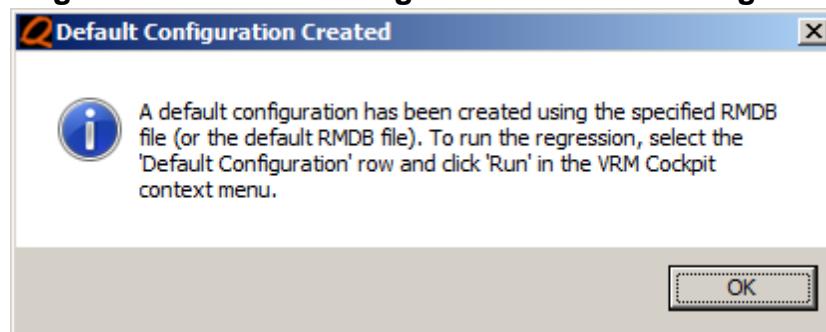
**Figure 4-1. Questa Verification Run Manager GUI**



### Default Configuration Created Dialog Box

The first time you open the VRM GUI you will receive this dialog box, informing you of the creation of a default configuration.

**Figure 4-2. Default Configuration Created Dialog Box**



## VRM Cockpit Window

---

When a VRM GUI session first opens, assuming a regression run is not also being launched by the same **vrun** command, the only window initially visible is the VRM Cockpit window. This window displays the contents of the currently loaded project file, including the RMDB files in use by the project and the pre-canned configurations attached to each RMDB file. This window also supports the creation of new configurations, the importation of regression run history from one or more *VRM Data* directories, the launching of new regression runs, and the opening of a VRM Results window for viewing the results of previously launched regression runs.

There is only one VRM Cockpit per VRM GUI session and only one project file can be loaded at any one time (although there is no limit to how many project files a user can create and maintain).

Each configuration is associated with a single RMDB file and calls out a single *VRM Data* directory. Once a configuration is created, the RMDB with which the configuration is associated cannot be changed. However, the *VRM Data* directory setting can be edited via the **Edit VRM Configuration** window. If the *VRM Data* directory setting for a given configuration is changed, the user is given a choice of using the new setting as is (and possibly creating a new *VRM Data* directory if one does not already exist at that location) or renaming the previous *VRM Data* directory to match the newly specified path.

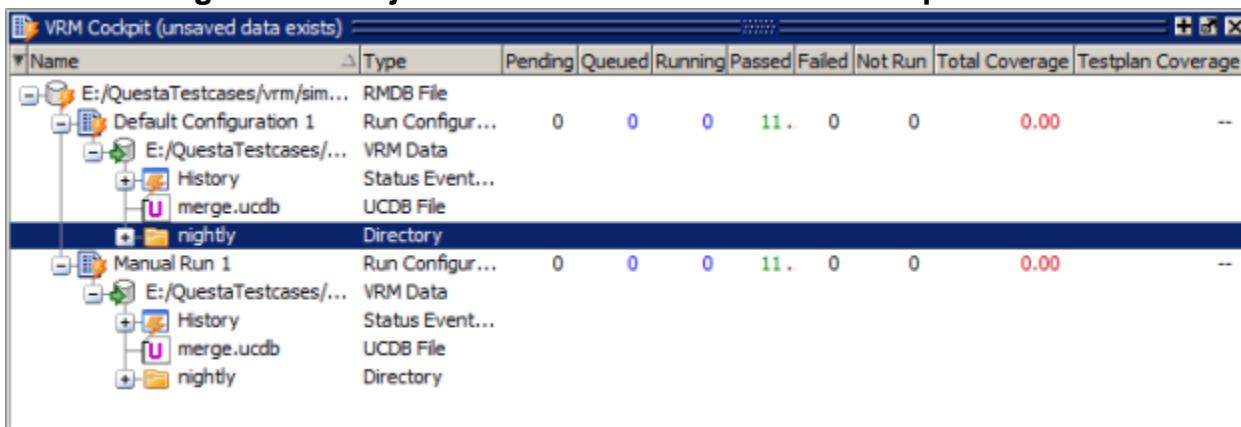
Note that if the *VRM Data* directory setting for a given configuration is changed, there may be no way to view the history contained within the previous *VRM Data* directory in the GUI unless the directory is renamed or some other configuration also refers to that same *VRM Data* directory.

Each *VRM Data* directory contains its own history. A single *VRM Data* directory can be used by more than one configuration (though that is not necessarily a recommended practice). By default, the GUI attempts to avoid overlapping *VRM Data* directories by auto-generating a unique path for *VRM Data* directory with each new configuration (which can be changed by the user).

The **VRM Cockpit** window display is a hierarchical browser populated with various kinds of data. The top-most level shows a list of the RMDB files registered in the project file. Under each RMDB entry is a list of named configurations associated with that RMDB. Under each configuration is a single node representing the *VRM Data* directory used by that configuration.

[Figure 4-3](#) shows a project file loaded into the **VRM Cockpit** window.

**Figure 4-3. Project File Loaded into the VRM Cockpit Window**



The **VRM Cockpit** shown in [Figure 4-3](#) behaves as follows when you double-click on a row:

- RMDB File — Opens the RMDB in a source window
- Run Configuration — Opens the **Edit VRM Configuration** window
- VRM Data — Expands/collapses the row
- History — Expands/collapses the row
- Status Event Log — Opens a **VRM Results** window
- Directory — Expands/collapses the row
- File — Attempts to open the file. Note that files assumed to be text files are opened in a source window. However, some files will open in a Browser (for example, UCDB files).

Under the *VRM Data* directory is a *History* row (see [Figure 4-3](#)), along with a listing of every file and subdirectory contained within the top-level *VRM Data* directory. These files and directory rows act as regular file browser entries, allowing the user to push down into the working directory of any Runnable in the regression suite that has been executed at least once in this *VRM Data* directory.

Under the *History* row is a single entry for each regression run executed in this *VRM Data* directory. Each of these entries is tied to the Status Event Log for a single regression run.

Note that every Status Event Log under a given *VRM Data* directory is represented as an entry under the *History* row, even if the regression run which created that Status Event Log was not initiated from the configuration under which it appears. Moreover, since regressions can be launched with one-time configuration changes and the user can edit the configurations at any time, there is no guarantee that each of the historical runs listed under a given configuration were run with the same set of command-line options. Each Status Event Log carries a record of the exact command-line options and Runnable selections used to launch that specific regression run and the GUI supports launching a new regression run using the exact options used in a

previous run. Therefore, the association of historical regression logs with specific configurations should be considered nothing more than an organizational convenience.

<b>Adding RMDB Files to the Project File . . . . .</b>	<b>144</b>
<b>Adding New Configurations to the Project File . . . . .</b>	<b>144</b>
<b>Edit VRM Configurations . . . . .</b>	<b>147</b>
<b>Importing Existing Regression History . . . . .</b>	<b>155</b>
<b>Initiating a Regression Run . . . . .</b>	<b>155</b>
<b>Attach to Running Regression and/or Viewing the Completed Results . . . . .</b>	<b>156</b>
<b>Viewing Regression Results Over Time . . . . .</b>	<b>157</b>

## Adding RMDB Files to the Project File

In order to define pre-canned configurations, it is necessary to reference at least one RMDB file in the project file.

1. Select the VRM Cockpit window.
2. Select the **VRM Cockpit > Add Item > RMDB File** menu item.
3. In the Select RMDB File dialog box, select the .rmdb file to associate with the configuration and click **Open**.

The RMDB file entry is also saved to the project file automatically (if a project file is loaded). An RMDB file can be added to a project even though no configurations exist that are associated with that RMDB file. However, any given RMDB file may only be added to a given project once (duplicates are not allowed). The RMDB files are displayed in the order they are added to the project.

## Adding New Configurations to the Project File

A named configuration is a collection of selected Runnables and command-line options. It is essentially a user-named template encapsulating a specific batch-mode *vrun* command.

### Creating a Configuration from the RMDB File Entry

1. Select an .rmdb file (Type: RMDB File) in the VRM Cockpit window.
2. Select the **VRM Cockpit > Add Item > Run Configuration** menu item.

This displays the **Create VRM Configuration** dialog box.

3. Define general information about the configuration.

The Create VRM Configuration dialog box is prepopulated with default information. Update the fields in the General tab as desired.

- a. Alter this information as desired. For example, in the VRM Configuration Name text entry box, change **Run Template 1** to **VRMDATA1**.
  4. Select the runnables to include in the configuration.
    - a. Click the **Runnables** tab to select the runnables you want to be part of the configuration.
    - b. Click the **Expand All** button in the lower-right corner to show all runnables in the available for the configuration.
    - c. Click the check box associated with the “nightly” entry to select all child runnables. Refer to the section “[Edit VRM Configurations](#)” for detailed information about the other tabs available.
  5. Click OK to finalize the creation of the configuration.
- This displays the **Create VRM Data Directory** dialog box.
6. Click **Yes** in the **Create VRM Data Directory** dialog box to create the necessary directory structure for the new configuration.
- Your new configuration is now included in the hierarchy under your .rmdb file.

## Cloning a Configuration

1. Select a configuration (Type: Run Configuration) in the VRM Cockpit window.
  2. Select the **VRM Cockpit > Clone Configuration** menu item.
- This displays the **Edit VRM Configuration** dialog box.
3. Define general information about the configuration.
- The Edit VRM Configuration dialog box is prepopulated with default information specific to cloning a configuration. Specifically, the VRM Configuration Name text entry box contains the string “Clone of <configuration\_name>”.
- a. Alter this information as desired.
  4. Select the runnables to include in the configuration.
    - a. Click the **Runnables** tab to select the runnables you want to be part of the configuration.
- This tab is prepopulated with your settings for the configuration you are cloning.
- b. Change the selected runnables as desired.
- Refer to the section “[Edit VRM Configurations](#)” for detailed information about the other tabs available.
5. Click OK to finalize the creation of the configuration.

## Creating a Configuration from a Previously-Run Configuration

Pre-requisite: You must have already run a configuration for the active project.

1. In the VRM Cockpit, expand the existing configuration through the History branch.
2. Select an event log from the History (Type: Status Event Log).
3. Select the **VRM Cockpit > Capture Configuration** menu item.

This displays the **Edit Regression Options** dialog box.

4. Define general information about the configuration.

The Edit Regression Options dialog box is prepopulated with default information specific to capturing a configuration. Specifically, the VRM Configuration Name text entry box contains the string “Run Template 1”.

- a. Alter this information as desired.
5. Select the runnables to include in the configuration.
  - a. Click the **Runnables** tab to select the runnables you want to be part of the configuration.

This tab is prepopulated with your settings for the configuration you are cloning.

- b. Change the selected runnables as desired.

Refer to the section “[Edit VRM Configurations](#)” for detailed information about the other tabs available.

6. Click Save to finalize the creation of the configuration.

## Creating a Configuration Manually

When you manually initiate a regression run is manually (either by typing additional command-line arguments on the original *vrun* command used to launch the GUI or by typing a *vrun* command into the **Transcript** window), VRM creates a new configuration automatically.

The option settings of this configuration is derived from the *vrun* command used to launch the batch-mode regressions.

## Edit VRM Configurations

---

This section describes the contents of the dialog box used to set up configurations in the VRM Cockpit window. This dialog box may have different names depending on how you arrive at it, including:

- Create VRM Configuration dialog box, when “[Creating a Configuration from the RMDB File Entry](#)” on page 144 .
- Edit VRM Configuration dialog box, when “[Cloning a Configuration](#)” on page 145 or when editing a selected configuration through the **VRM Cockpit > Edit Configuration** menu item.
- Edit Regression Options dialog box, when “[Creating a Configuration from a Previously-Run Configuration](#)” on page 146.

The rest of this section will refer to all occurrences as the **Edit VRM Configuration** dialog box.

This **Edit VRM Configuration** dialog box contains multiple tabs, defined in the following sections, that allow you to select Runnables for execution and specify various command-line options related to a regression run. Refer to the section [Edit VRM Configuration Dialog Box Tabs](#).

Any changes made to the options or Runnable selections in the **Edit VRM Configuration** dialog box are saved to the project file when the **OK** button is pressed. If no project file is open, then the changes are saved in the **VRM Cockpit** only and can be written to a project file by selecting the **File > Save Project As** menu item.

If unsaved changes are detected upon exit from the application, you are prompted to save the VRM Cockpit data to a project file at that time. Refer to the sections “[Initial Start-up and Project Management](#)” on page 171 and “[Project Control](#)” on page 176 for more details.

When closing this dialog box, if the VRM Data directory, from the General tab, does not exist, you are prompted about creating the directory. If the VRM Data directory path has been changed and a directory exists at the previous VRM Data directory path, then you are prompted as to whether you want to use the new name as is (possibly creating a new VRM Data directory) or rename the existing directory to the new location.

<b>Quick Access to the Edit VRM Configuration Dialog Box .....</b>	<b>147</b>
<b>Edit VRM Configuration Dialog Box Tabs.....</b>	<b>148</b>

## Quick Access to the Edit VRM Configuration Dialog Box

You can use the Edit VRM Configuration dialog box to make one-time edits to the runnable selections and/or command-line options before launching a regression run by using the menu items under **VRM Cockpit > Run Special**.

- Edit Config and Run — use this to alter the selected configuration and run the regression upon closing the dialog box. Your changes are saved to the selected configuration.
- Clone Config and Run — use this clone the selected configuration and run the regression upon closing the dialog box. Your changes are saved to a clone of the selected configuration.
- One-time Config and Run — use this to temporarily alter the selected configuration and run the regression upon closing the dialog box. Your changes are not saved to the selected configuration.

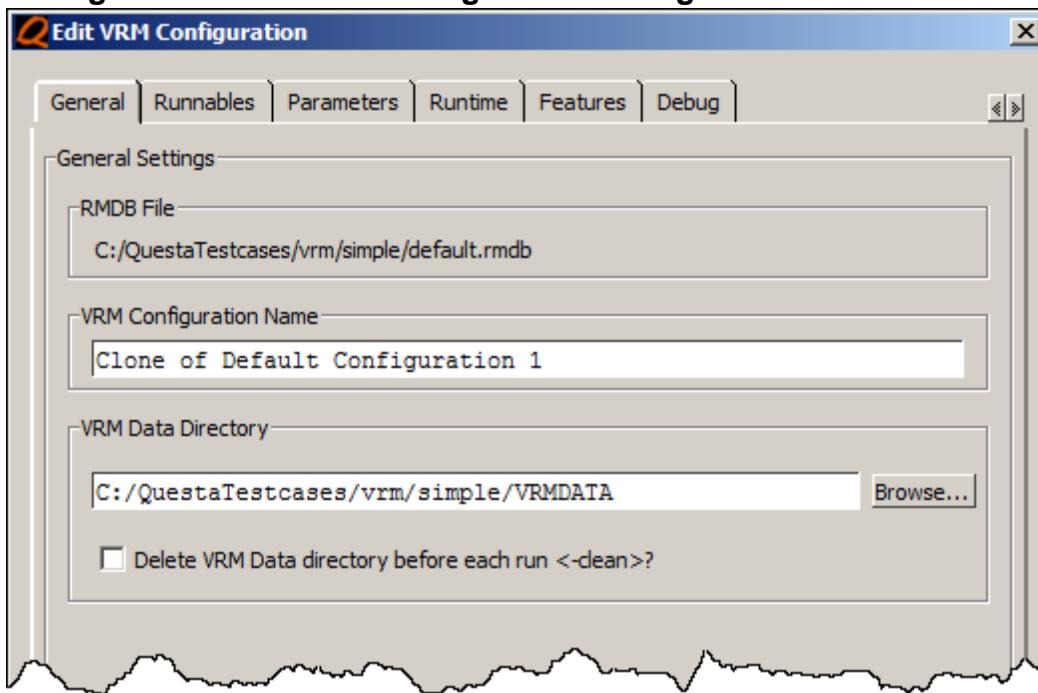
## Edit VRM Configuration Dialog Box Tabs

This section defines the contents of the Edit VRM Configuration dialog box.

### General Tab

Use this tab to alter basic settings of the configuration.

- RMDB File — shows the directory location of the associated RMDB file.
- VRM Configuration — change this string to specify a name for your configuration, where any string is valid.
- VRM Data Directory — change the location where status event logs and associated files are written. You also have the option to instruct VRM to delete this VRM Data directory before each run (equivalent to the -clean option to vrun command).

**Figure 4-4. Edit VRM Configuration Dialog Box — General Tab**

### Runnables Tab

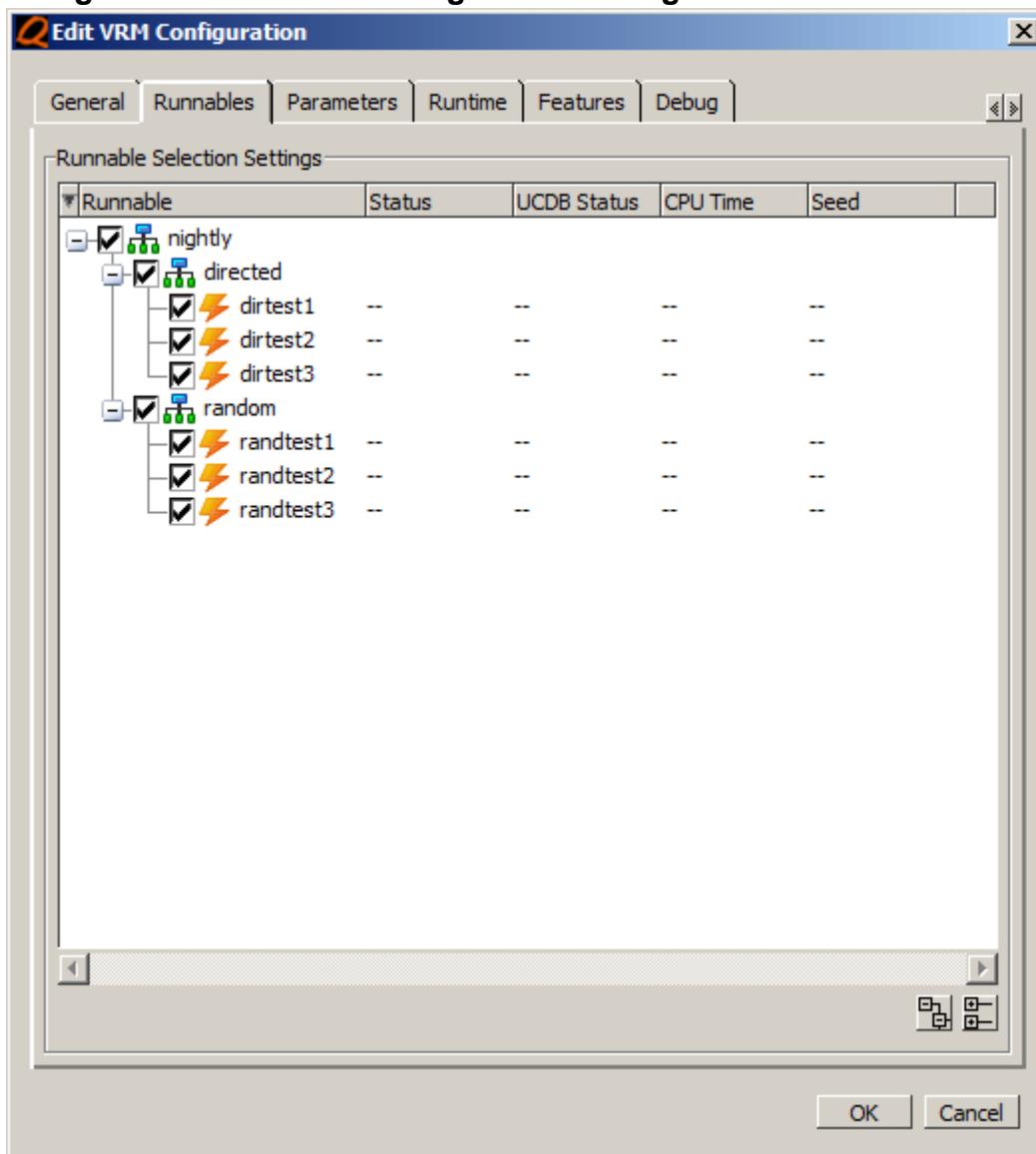
Use this tab to select which runnables are to be run. You can choose from two ways to select the runnables:

- Select Runnables by Name — This method allows you to choose runnables by selecting check boxes in a hierarchical view. Your selections are equivalent to the -include and -exclude options to vrun.

The columns contain information about the execScript of the runnable if at least one regression run has been executed in the designated VRM Data directory. This is to be used as an aid when selecting runnables based on their last-known status.

- Select Actions by Status — This method allows you to choose runnables by selecting different categories. Your selections are equivalent to the -select option to vrun, as described in the section “[Semi-automatic \(Two-command\) Mode](#)”.

**Figure 4-5. Edit VRM Configuration Dialog Box — Runnable Tab**



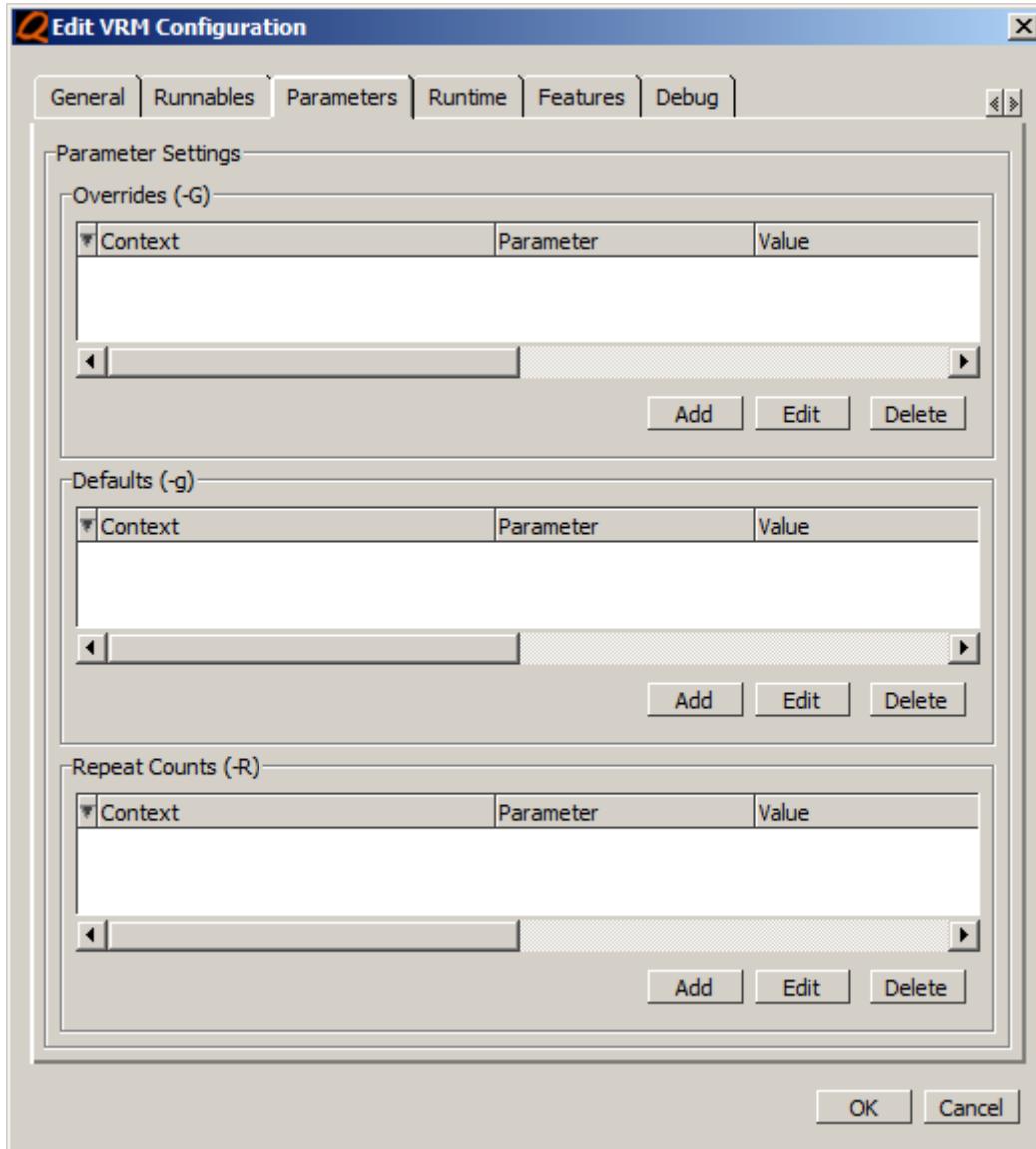
If no Runnables are selected in the **Runnables** tab at the time the contents of the **Edit VRM Configuration** are saved to the project file, and if there is a *toprunnables* attribute in the document element of the RMDB file, then the Runnables listed in the *toprunnables* attribute in the RMDB file is selected in the configuration.

The Runnable selection may be inspected and/or modified by invoking the **Edit VRM Configuration** window again. If one or more Runnables are selected when the contents of the **Edit VRM Configuration** window are saved to the project file, then the selections are not modified, even if a *toprunnables* attribute exists. To restore the default selection from the RMDB file, open the **Edit VRM Configuration** window, deselect all Runnables in the **Runnables** tab, and save the configuration.

## Parameters Tab

Use this tab to control the behavior of parameters for your regression runs.

**Figure 4-6. Edit VRM Configuration Dialog Box — Parameters Tab**

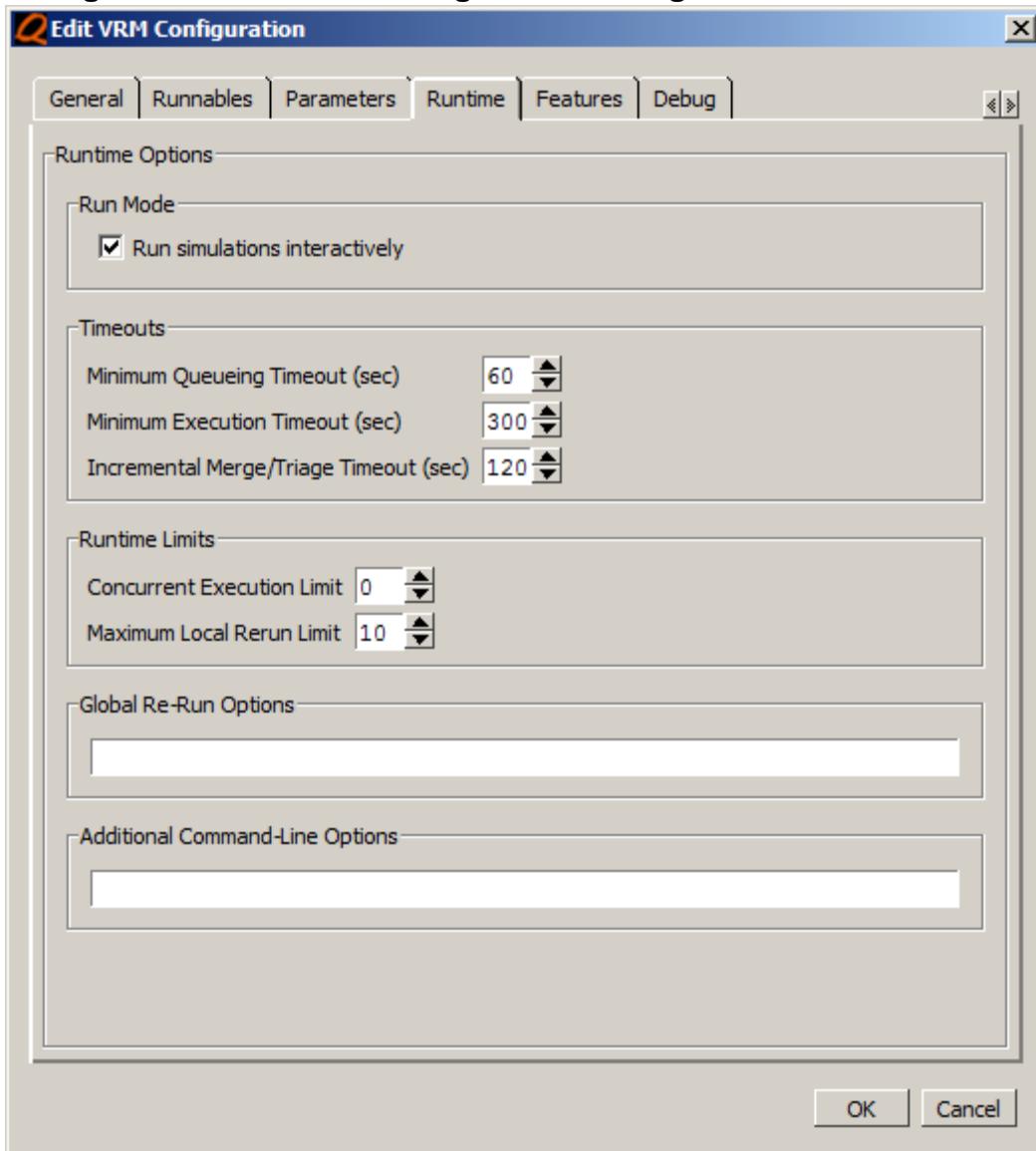


- Overrides (-G) — specifies a value for a given parameter that is used irrespective of any other value present in the database. For additional information, see “[Override Parameter Values from Command Line](#)” on page 132.
- Defaults (-g) — Specifies a value for a given parameter that is used if no other value is defined for that parameter in the database. For additional information, see “[Override Parameter Values from Command Line](#)” on page 132.
- Repeat Counts (-R) — Overrides the repeat count of any Task or Group. See “[Override Repeat Counts on Runnable Nodes](#)” on page 92 for additional information.

### Runtime Tab

Use this tab to control the runtime behavior of the regression run.

**Figure 4-7. Edit VRM Configuration Dialog Box — Runtime Tab**



- **Run Mode** — The **Run simulations interactively** option instructs regression runs that use this run configuration to launch simulations in interactive (or GUI) mode.
- **Timeouts** — Control settings related to the `-mintimeout` and `-inctimeout` options to `vrun`. For additional information, see “[Timeouts](#)” on page 311
- **Runtime Limits group** — Control settings related to the `-j` and `-maxrerun` options to `vrun`. See “[Limit Concurrently Running Processes](#)” on page 92 for details. Also see “[Named Execution Queues](#)” on page 333
- **Global Re-Run Options** — Control arguments to the `-rerun` option to `vrun`.

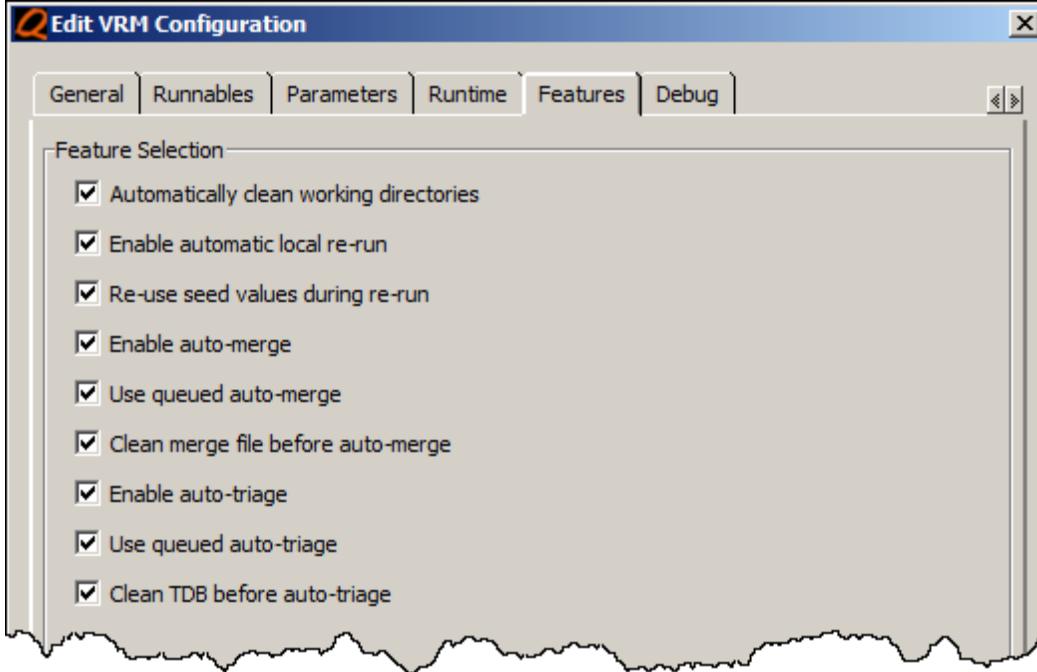
- Additional Command-Line Options — enter **vrun** options that are not otherwise covered in this or the other tabs of the dialog box.

If you enter an option into this text-entry box that has a setting elsewhere in the dialog box, then the value of that option is reflected in the widget corresponding to that option the next time the **Edit VRM Configuration** window is opened.

#### Features Tab

Use this tab to disable various default settings of the regression run.

**Figure 4-8. Edit VRM Configuration Dialog Box — Features Tab**



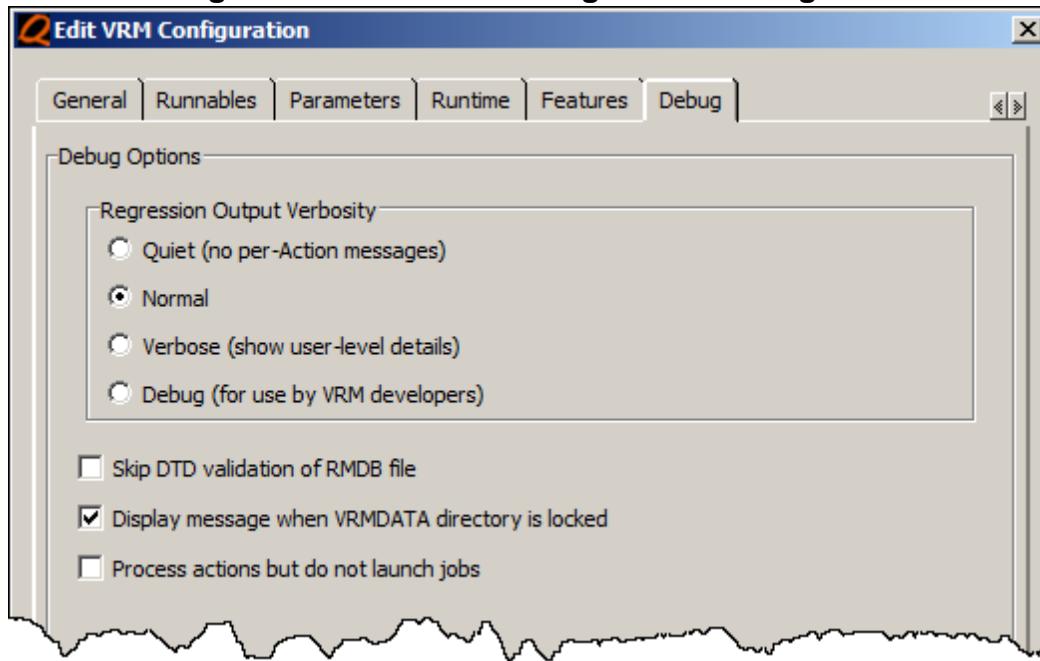
- Automatically clean working directories — disabling this is the equivalent of adding the **-noautoclean** option to **vrun**.
- Enable automatic local re-run — automatically re-queues individual failed actions. Disabling this is the equivalent of the **-nolocalrerun** option to **vrun**.
- Re-use seed values during re-run — Refer to “[Reuse of Random Seeds in Automated Rerun](#)” on page 358 for additional information. Disabling this is the equivalent of the **-noreuseseeds** option to **vrun**.
- Enable auto-merge — Disabling this is the equivalent of the **-noautomerge** option to **vrun**.
- Use queued auto-merge — controls queued auto-merge activities, see “[Enabling and/or Disabling Coverage Merge Flows](#)” on page 396 for more information. Disabling this is the equivalent of the **-noqueuemerge** option to **vrun**.

- Clean merge file before auto-merge — Disabling this is the equivalent of the -noautomergedelete option to vrun, see “[Automatic Deletion of Merge/Triage Files](#)” on page 394 for additional information.
- Enable auto-triage — Disabling this is the equivalent of the -noautotriage option to vrun, see “[Automatic Deletion of Merge/Triage Files](#)” on page 394 for additional information.
- Use queued auto-triage — Disabling this is the equivalent of the -noqueuetriangle option to vrun, see “[Automated Results Analysis](#)” on page 398 for additional information.
- Clean TDB before auto-triage — Disabling this is the equivalent of the -noautotriageclean option to vrun, see “[Automatic Deletion of Merge/Triage Files](#)” on page 394 for additional information.

#### Debug Tab

Use this tab to control debug settings of the regression run.

**Figure 4-9. Edit VRM Configuration Debug Tab**



- Regression Output Verbosity — See “[Message Verbosity and Debugging](#)” on page 460 for information.
- Skip DTD validation of RMDB file — Control validation of RMDB file against Document Type Definition (DTD) in the release directory. The equivalent vrun switch is -nodtdvalidate to disable this functionality. See “[Validation of the RMDB Database](#)” on page 63 for additional information.
- Display message when VRMDATA directory is locked — Control whether VRM displays a message in this instance. The equivalent vrun switch is -nolockmsg to disable this functionality.

- Process actions but do not launch jobs — Generate necessary files, but do not launch the scripts. The equivalent vrun switch is -noexec to enable this functionality. See “[Dry Run of the RMDB Database](#)” on page 62 for additional information.

## Importing Existing Regression History

Sometimes a user has an existing *VRM Data* directory that was created by a regression run launched without a GUI configuration (as would be the case for nightly regressions initiated by a cron script). The current implementation makes no direct provision for inspecting *VRM Data* directories that are not associated with a configuration, but the obvious workaround is to create a dummy configuration and set the *VRM Data* directory to point to the directory being used by the non-GUI regression runs. The directory then appears under the dummy configuration and, under that, the historical Status Event Logs from that *VRM Data* directory are visible.

However, the options in the dummy configuration remain set to their default values rather than to the values used for the regressions whose results are stored in that *VRM Data* directory. To remedy that, a context-menu item exists to import a given *VRM Data* directory into the project file. To import a *VRM Data* directory

1. Select the **VRM Cockpit > Import > VRM Data Directory** menu item to display the **Select VRM Data Directory** navigator.
2. From the navigator, select the VRM Data directory to import.

This creates a new configuration with the name “Imported Results <n>”.

If the RMDB file associated with that Status Event Log is not already in the project file, then it too is added. This is similar to the dummy configuration workaround described above except that the settings from the most recent regression run are imported into the configuration.

Subsequent regression runs manually launched against that *VRM Data* directory will appear under the *History* row for that configuration in subsequent GUI sessions (to force said runs to show up in the current GUI session, it may be necessary to invoke the **Refresh Display** context-menu item in the **VRM Cockpit**). The newly imported configuration (and the entries under the *History* row) may also be used to launch new regression runs from the GUI.

## Initiating a Regression Run

Following are the ways to initiate a new regression run from the VRM GUI:

- Select a configuration in the **VRM Cockpit** and use the context menu to select **Run**
- Select a historical regression run (that is, Status Event Logs) in the **VRM Cockpit** and use the context menu to select **Run Again**
- Use the context menu in the **VRM Results** to select **Run Again...**

The first initiates a regression run using the command-line options and Runnable selections stored in the selected configuration. The other methods use the command-line options and

Runnable selections that were used for the selected historical regression run. Each of these menu items also has a corresponding menu item (**One-time Edit and Run Again...**) by which a **Edit VRM Configuration** window is opened in order to allow the user to edit the command-line options and/or the Runnable selections prior to launching the new regression run. Any modifications made with the **One-time Edit and Run Again...** menu items affect only the new regression run and not the saved configurations or the Status Event Logs.

When a new regression run is invoked from the GUI using any of these methods, a new **VRM Results** window is opened to monitor the status of the regression run. The output of the batch-mode *vrun* process is captured in a file and may be viewed by selecting the View > Vrun Output menu item.

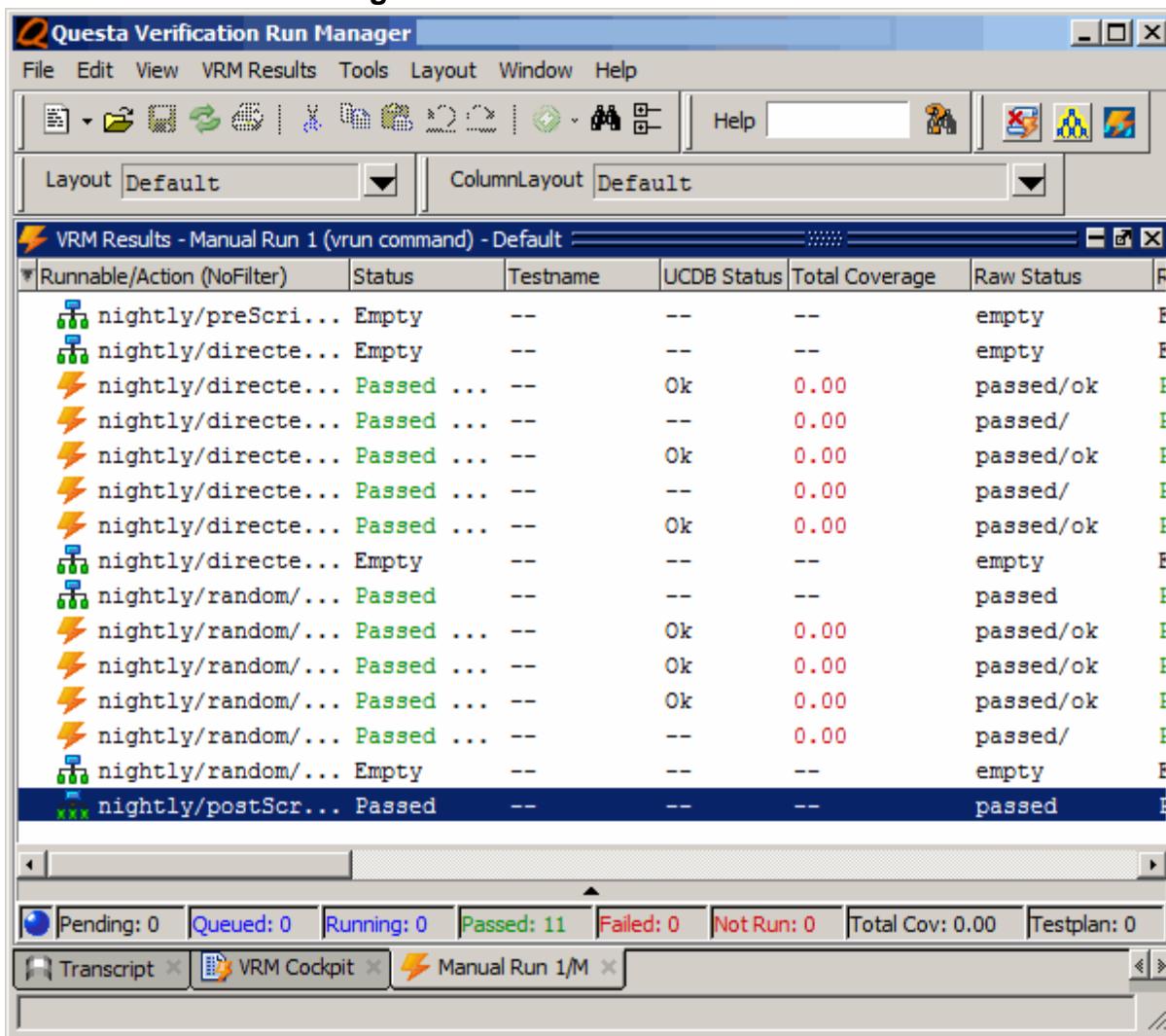
## Attach to Running Regression and/or Viewing the Completed Results

In order to view the results of a regression that was previously launched, it is necessary to find the Status Event Log entry representing that regression run in the **VRM Cockpit**. The Status Event Log entry for any given regression run is under the *History* row, which is under the *VRM Data* directory in which the regression run was executed. The *VRM Data* directory row is under one of the configuration entries.

Once the appropriate Status Event Log entry is located, select that row in the **VRM Cockpit** and select the **VRM Cockpit > View Results** menu item.

This opens a new **VRM Results** window (see [Figure 4-10](#)) containing the results of the selected regression run. If the selected regression run is still in progress, the GUI will attempt to connect to that regression run via the status socket *port@host* recorded in the Status Event Log. If the connection is successful, the **VRM Results** window contents reflects the real-time status of the ongoing regression.

**Figure 4-10. VRM Results Window**



## Viewing Regression Results Over Time

As each regression completes, its resulting status is loaded into a database for future analysis. This history can be viewed by selecting one of the configuration entries in the **VRM Cockpit** window and then select the **VRM Cockpit > View History** menu item.

The History (see [Figure 4-11](#)) shows a complete list of all the Actions executed within the corresponding *VRM Data* directory. Under each **Runnable/Action** row is a list of all the regression runs during which that particular Action was executed and the status of the Action in question for each of those regression runs.

**Figure 4-11. VRM Results History Window**

Runnable/Action	Status	UCDB Status	Queued Time	Hostname	Elapsed Time	Sim Time
nightly/directed/dirtest1...	Passed	Ok	--	ORW-L...	--	0 ns
Mon Feb 27 2:26:10 PM ...	Passed	Ok	03.00	ORW-L...	07.00	0 ns
Wed Feb 22 3:15:20 PM ...	Passed	Ok	03.00	ORW-L...	05.00	0 ns
Wed Feb 22 9:35:32 AM ...	Passed	Ok	03.00	ORW-L...	06.00	0 ns
+ nightly/directed/dirtest2...	Passed	Ok	--	ORW-L...	--	0 ns
+ nightly/directed/dirtest3...	Passed	Ok	--	ORW-L...	--	0 ns
- nightly/directed/postScri...	Empty	--	--	--	--	--
Mon Feb 27 2:26:30 PM ...	Empty	--	--	--	--	--
Wed Feb 22 3:17:05 PM ...	Empty	--	--	--	--	--
Wed Feb 22 3:15:39 PM ...	Empty	--	--	--	--	--
Wed Feb 22 9:35:52 AM ...	Empty	--	--	--	--	--
+ nightly/directed/preScrip...	Empty	--	--	--	--	--
+ nightly/postScript	Passed	--	--	ORW-L...	--	--

## VRM Results Window

In any single VRM GUI session, there is only a single **VRM Cockpit** window. However, there may be any number of **VRM Results** windows. Each **VRM Results** window is generally associated with a single regression run (regardless of whether said regression run is currently ongoing or has already completed). If the regression with which a given **VRM Results** window is associated is rerun from the context menu of the **VRM Results** window itself, the same **VRM Results** window may be “re-purposed” to monitor the newly launched regression run. If a regression run is launched by any other means, or if a historical regression run is opened, a new **VRM Results** window is invoked. A **VRM Results** window may be closed and/or minimized as the user wishes.

The bulk of the **VRM Results** window consists of a multi-column table where each row represents either one Runnable or one Action and the first column is the context string associated with that Runnable or Action. The first column may be displayed as a hierarchical tree of Runnables or as a flat list of Actions (this view mode is user-selectable). When viewing a hierarchical tree of Runnables, the *preScript* and *postScript* Actions for each group Runnable may be displayed as children of the Runnable or not displayed at all (this view mode is also user-selectable).

Note that the *Empty* status means the Action node could exist in the execution graph but no script for that particular Action was defined in the RMDB. This happens mostly with the *preScript* and *postScript* Actions associated with a group Runnable.

When a given table row represents an Action, the *Status* column and other columns display various bits of data associated with that Action. When the row represents a group Runnable (visible in hierarchical mode only), the other columns are blank. The data columns show, among other things, the current status of the Action, the times the Action spent queued and/or executing, the name of the host on which the Action ran (or is running), the CPU time spent, and other interesting bits of data. Which columns are populated depends on the current status of the Action.

Note that the time-related columns (*Queued Time*, *Elapsed Time*, *Sim Time*) sort in time order rather than lexical order.

Below the Action status table is a row of counters with an animated “running light” on the left-most side of the. The running light indicates the state of the overall regression run (such as Queued, Running, or Passed). The counters indicate how many Actions exist in each of the listed categories as follows:

- The **Pending** category includes tests that are not yet eligible to launch (they are waiting for something else to finish. For example,

**Waiting:1**

indicates that there is one job pending.

- The **Queued** category includes tests that are eligible and have been launched, but have not yet reported that they are running (these tests are probably in the grid queue).
- The **Running** category indicates the tests that are running. The running tests are listed in the **Status** column.
- The **Passed** category counts tests that completed successfully.
- The **Failed** category counts tests that have failed.
- The **Timeout** category counts tests that failed due to a time out.
- The **Killed** category counts tests that were terminated.
- The **Skipped** category counts tests that were skipped.
- The **Dropped** category counts tests that were dropped.
- The **Coverage** category indicates the total coverage of the design at the time the UCDB was generated.
- The **Testplan** category indicates the coverage with respect to the testplan(s) merged into the UCDB.

You can alter these categories as described in the section “[Counter Control in the VRM GUI](#)”

The contents of the **VRM Results** window are read-only from the user’s point of view and are always associated with a single regression run.

The **VRM Results** window for a given regression run only shows those Actions that were part of the run, according to the pattern of selected Runnables. In the case where a given **VRM Results** window is monitoring an ongoing regression run, new status events are passed from an internal scoreboard to the **VRM Results** window. In the majority of cases, a row corresponding to the Action whose status is being reported will already exist in the table. However, in some cases (such as dynamically repeating Runnables), status may be reported for Actions that were not defined in the original list. In those cases, the table is reloaded from the scoreboard in order to place the new row in its proper position in the table prior to updating its status.

You can alter the view of the Runnables or Actions through the View menu. Specifically the following options

- Hierarchical Mode — (default) shows a hierarchical view of all actions.
- Test-only Mode — (only available if you specify -testname on the vrun command line). shows listing of tests, rather than actions, as defined by the use of the testname parameter for execScripts.
- Flat Action Mode — shows a flat view of all the actions.
- Exclude pre/postScript — shows a flat view of actions that are not preScript or postScript.

The title of each **VRM Results** window consists of the name of the configuration under which the regression run is tracked and an optional date-of-launch if the window is monitoring a regression that was not launched in the current GUI session or which has completed. For regressions launched from a configuration entry in the **VRM Cockpit**, only the configuration name is displayed. For regressions launched manually (that is, from the initial GUI command line or from the transcript prompt), a new uniquely named configuration is created and this name is displayed as the title. For **VRM Results** opened to view the status of an existing regression listed under the *History* node of a configuration, the name of the parent configuration and the date and time of launch is displayed.

<b>Control Other Functions from the VRM Results Window .....</b>	<b>161</b>
<b>VRM Results Window Filters.....</b>	<b>162</b>
<b>VRM Results History Window .....</b>	<b>168</b>

## Control Other Functions from the VRM Results Window

There are other VRM GUI functions that can be invoked from the **VRM Results** window. In general, these functions are associated with a particular Action which means one or more Action rows must be selected prior to using the context-menu to invoke the particular function. The list of functions supported includes the following:

- Rerun regression (with option to edit Runnable selection and/or command-line options)
- Suspend/resume/kill entire regression run
- Suspend/resume/kill selected Actions
- View the *vrun* output for the regression run
- View the log file for one or more selected Actions
- View any file in the working (**VRMDATA**) directory for the selected Action
- View the UCDB file for selected Actions in the UCDB Browser
- View the merge file for selected Actions in the UCDB Browser
- View the merge file for selected Actions in the Verification Tracker

Following are the cross-functional connections:

- Open the UCDB file for a selected Action in a *vsim -viewcov* session
- Open the WLF file for a selected Action in a *vsim -view* session

## VRM Results Window Filters

The VRM Results window provides an interface for filtering results based on pre-defined or user-generated filters.

To filter the information in the VRM Results window, you specify one or more data fields along with a comparison operator and a target value for each. These terms can be combined in various ways to form a logical expression. Data rows which match the logical expression are displayed, rows which do not match are suppressed.

<b>Configuring Filters for the VRM Results Window.....</b>	<b>162</b>
<b>Applying Filters in the VRM Results Window .....</b>	<b>163</b>
<b>Filter Expressions Dialog Box.....</b>	<b>165</b>
<b>Create Filter Expression Dialog Box .....</b>	<b>166</b>

## Configuring Filters for the VRM Results Window

Control the information that appears in the VRM Results window based on the following configuration tasks you can perform related to filters in the VRM Results Window.

### Procedure

1. Right-click in the VRM Results window and select **Filter Expressions > Configure Filter Expressions**.
2. Perform any of the following configuration tasks through the Filter Expressions dialog box.

Create a new filter:

- Click the Create button
- Complete the fields in the Create Filter Expression dialog box. Refer to the Create Filter Expression dialog box section for detailed information

Import or Export an existing filter:

- From the Names box, select the filter you want to import or export.
- Click the Import or Export button to display a file navigation dialog box.
- Select an existing file to import from or create a new file to export to.

Edit an existing filter:

- From the Names box, select the filter you want to edit.
- Click the Edit button to display the Edit Filter Expression dialog box.

- Alter the fields of the Edit Filter Expression dialog box. Refer to the Create Filter Expression dialog box section for detailed information

Remove an existing filter:

- From the Names box, select the filter you want to remove
- Click the Remove button
- Accept the confirmation dialog box

Create a copy of an existing filter with a new name:

- From the Names box, select the filter you want to copy
- Click the Copy button:
- Specify the name of the new filter in the Copy Filter Expression dialog box

Rename an existing filter:

- From the Names box, select the filter you want to rename
- Click the Rename button
- Specify the new name of the filter in the Rename Filter Expression dialog box

## Related Topics

[Create Filter Expression Dialog Box](#)

[Filter Expressions Dialog Box](#)

## Applying Filters in the VRM Results Window

Control the information that appears in the VRM Results window based by applying a pre-existing or user-created filter through the following steps.

### Prerequisites

- Start, or complete, a vrun regression.

### Procedure

1. Change the focus to the VRM Results window
2. Right-click in the VRM Results window and select **Filter Expressions > FilterName** where *FilterName* is the name of the filter you want to apply.
3. Alternative Procedures
  - Use the **VRM Results > Filter Expressions > FilterName** menu item.

- Expand the Filter Expression drawer at the bottom of the VRM Results window by clicking on the black up-arrow. Once open, you can select a **FilterName** from the drop-down box.
- Right-click in the VRM Results window and select **Filter Expressions > Configure Filter Expressions**, then, from the Filter Expressions dialog box, select the filter and click Apply.

## Results

- The VRM Results window refreshes to show results that fit the applied filter.
- The header of the Runnable/Action column of the VRM Results window shows, in parentheses, the **FilterName**.

## Related Topics

[Create Filter Expression Dialog Box](#)

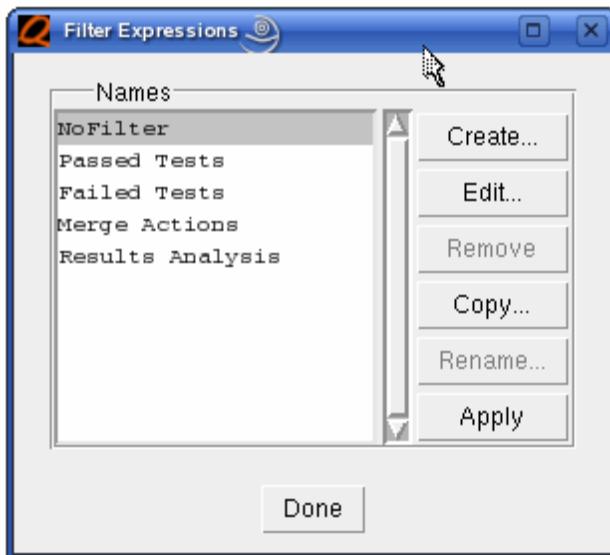
[Filter Expressions Dialog Box](#)

## Filter Expressions Dialog Box

Invocation: **VRM Results > Filter Expressions > Configure Filter Expressions**

Portal for configuring and applying filters for the VRM Results window

**Figure 4-12. Filter Expressions Dialog Box**



### Objects

- Names

List of existing filters.

- NoFilter — filter that removes any filtering applied to the window. You can modify, but not delete or rename this filter.
- Passed Tests — show only tests that have passed.
- Failed Tests — show only tests that have failed.
- Merge Actions — show actions that involved a merge action.
- Results Analysis — show actions that involve a triage action.

- Buttons

Actions for configuring filters ([Configuring Filters for the VRM Results Window](#)) or applying filters ([Applying Filters in the VRM Results Window](#))

### Related Topics

[Configuring Filters for the VRM Results Window](#)

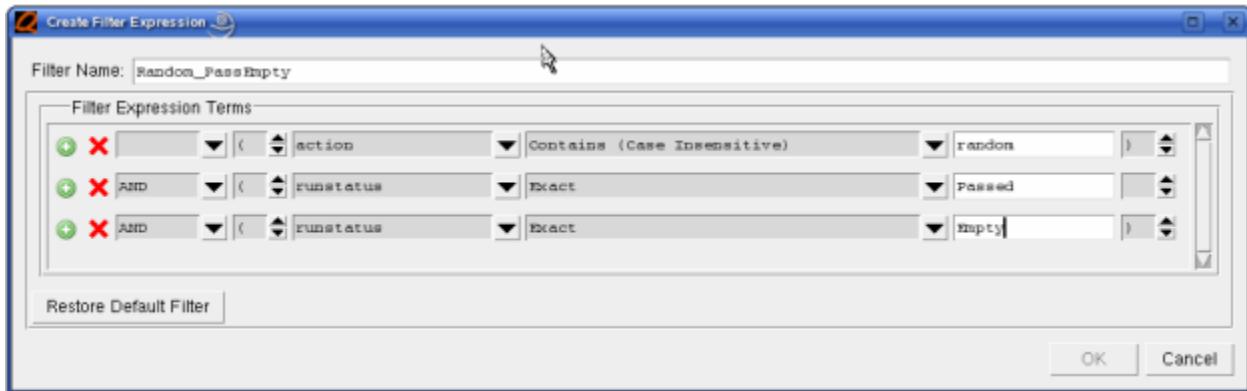
[Applying Filters in the VRM Results Window](#)

## Create Filter Expression Dialog Box

Also applies to the Edit Filter Expression Dialog Box

Dialog box used for building filter expressions.

**Figure 4-13. Create Filter Expression Dialog Box**



### Objects

- Filter Name

Text-entry box for defining the name of the filter. You cannot edit this field when using the Edit Filter Expression dialog box.

- Filter Expression Terms

Logical entries for defining your filter, where the key elements are:

- Add and Remove buttons — either add a filter row below the current row or remove that rule filter row.
- Logical Operator box— specifies a logical argument for combining adjacent rules. Your choices are: AND, OR, AND NOT, and OR NOT. For the first filter row, only NOT is available.
- Open Parenthesis box— controls rule groupings by specifying, if necessary, any open parentheses. The up and down arrows increase or decrease the number of parentheses in the field.
- Field Name box — specifies that your filter value applies to a specific column of the VRM Results window.
- Comparison Operator box— specifies whether the Field Name should or should not contain a given value.
  - For numeric- and time-based comparison operators, your choices are: ==, !=, <, <=, >, >=, Contains, Doesn't Contain, and Exact.
  - For text-based comparison operators, your choices are: Contains, Doesn't Contain, or Exact.

- Target Value box— specifies the filter value associated with your filter rule.
- Closed Parenthesis box— controls rule groupings by specifying, if necessary, any closed parentheses. The up and down arrows increase or decrease the number of parentheses in the field.

If any part of your filter expression is not formed properly, the OK button will be disabled and any offending section is highlighted in red.

**Table 4-1. Create Filter Expression Dialog Box - Field Box Options**

Field	Type	Description
action	string	VRM Action context string
cputime	numeric	CPU time required for Action (UCDB <sup>1</sup> )
datetime	2	Date/time of latest status change event
elapsed	numeric	Time elapsed between “start” message and “done” message from Action wrapper
extstatus	string	Provides extended information, when available.
hostname	string	Name of the host (machine name) on which the Action was executed
mergefile	string	Path to the merged UCDB file associated with the Action
queued	numeric	Time elapsed between Action launch and “start” message from Action wrapper
rawstatus	string	Status as reported in the status event log
reason	string	Text of the first error encountered by the Action (UCDB)
runstatus	string	Status summary. Such as: pending, queued, running, passed, or failed.
simtime	numeric	Simulation time of the Action upon completion (UCDB)
seed	string	Seed used in the simulation (UCDB, execScript only)
status	string	Current status of the Action
testname	string	Testname of the Action (UCDB)
triajefile	string	Path to the .tdb file associated with this Action
tstime	numeric	Time of the first error encountered by the Action (UCDB)
ucdbfile	string	Path to the UCDB file associated with this Action
ucdbstatus	string	Value of the UCDB TESTSTATUS attribute (UCDB)

**Table 4-1. Create Filter Expression Dialog Box - Field Box Options (cont.)**

Field	Type	Description
username	string	Name of the user who executed the test (UCDB)

1. The fields marked “UCDB” contain values retrieved from the test-data record of the UCDB file associated with the Action. In the event no UCDB file is generated by a given Action, these fields will be blank for that Action.
2. The “datetime” field is stored in ISO8601 format, specifically “YYYYMMDDTHHMMSS”. where: “YYYY” is the four-digit year, “MM”, “DD”, “HH”, “MM”, and “SS” are the two-digit month, day, hour, minute, and second, and “T” is just a “T”. This format makes it possible to do a lexical comparison between dates so “datetime < 20131024” would filter for Actions whose last status change happened before 24 Oct 2013.

## Related Topics

[Configuring Filters for the VRM Results Window](#)

[Applying Filters in the VRM Results Window](#)

## VRM Results History Window

The **VRM Results - VRMDATA (history)** window is a slightly modified version of the **VRM Results** window.

The primary differences are as follows:

- The **VRMDATA (history)** window may not include the status counters or the running light.
- While the **VRM Results** window shows only selected Runnables, the **VRMDATA (history)** shows all Runnables defined in the RMDB.
- Each Action node can be expanded to show the pass/fail status of that Action for every historical regression run.
- Some of the extra data columns may not be able to display values.

The reason the **VRM Results -VRMDATA (history)** window shows all Runnables defined in the RMDB is so that the user cannot only see what parts of the regression suite have been executed but also which parts of the regression suite have never been executed. In some cases (such as data fields that are read from the UCDB file), the values in some columns represent the most recent value for that particular data field. This is because the history database stores a subset of all the available data but does not archive the UCDB files or other collateral files.

Note that the history record associated with any given regression run is only added to the history database once the regression run has completed. To see an up-to-date list of the status of each Action during the regression run, a **VRM Results** window pointing to the regression run in question should be used.

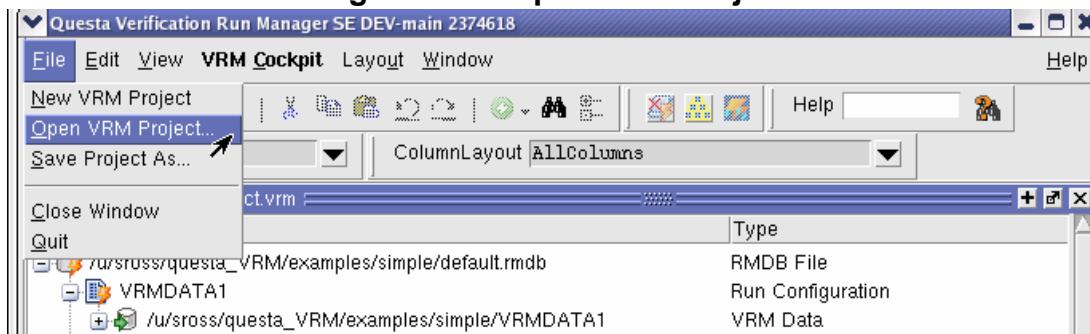
## VRM Project File and Configurations

The VRM Project file is made up of one or more configurations, each of which is associated with a specific RMDB file. Each configuration contains data to be used for regression runs that are launched via that configuration. This data includes a pointer to the *VRM Data* directory to be used for the regression results, the pattern of included/excluded Runnables to be executed, and the *vrun* command-line options to be used for the batch-mode regression run. Whenever a configuration is added, deleted, or edited, the new configuration data is saved to the VRM Project file. There is no limitation on how many project files a user may define or where they may be stored. However, only one project file can be loaded into a single VRM GUI session at any one time.

The **VRM Cockpit** window is opened any time a project file is loaded. Closing the **VRM Cockpit** window will close the project file and, likewise, closing the project file will close the **VRM Cockpit** window. To open an existing project file (see [Figure 4-14](#)), do the following:

**select -> File -> Open VRM  
Project...**

**Figure 4-14. Open VRM Project**



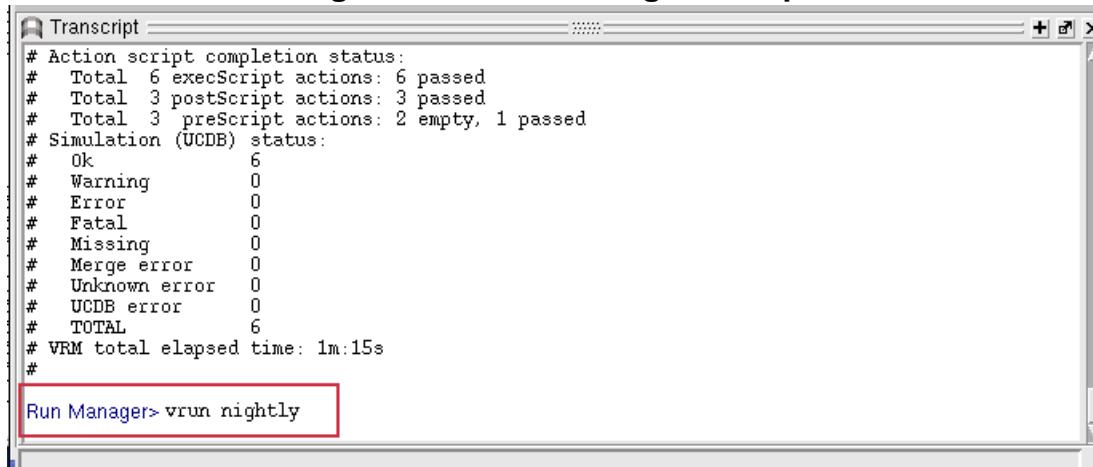
If another project file is opened at the time a new one is opened, the previously opened project file is automatically closed. This allows the user to jump back-and-forth between projects quickly. Any **VRM Results** windows that are open when the project file is closed remain open.

## Invoke Batch-mode Regressions During a GUI Session

The most common use model for launching a regression run is to use the **VRM Cockpit** context-menu to select **Run** on a configuration or **Run Again** on a history mode, or to use the **VRM Results** context-menu to invoke **Run Again** on the regression represented by that window. In this case, the status of the new regression is monitored via a new **VRM Results** window and the log output of the resulting batch-mode regression is not displayed in the **Transcript** window (the log output is captured in a file and may be viewed as a file upon request).

There are two ways in which a new regression run can be invoked by the user typing in a *vrun* command manually. The first is by typing the *vrun nightly* command in the **Transcript** window at the **Run Manager** prompt (see [Figure 4-15](#)). In this case, it is assumed that the user is interested in seeing the log output, just as if a batch-mode *vrun* command had been typed at a shell prompt. The GUI monitors the log output and, if the *vrun* command results in a regression run, a **VRM Results** window is opened to monitor the status of the regression run (in addition to the log output being displayed in the **Transcript** window).

**Figure 4-15. Run Manager Prompt**



The other way a new regression run can be invoked manually is by adding additional command-line options to the *vrun -gui* command when the VRM GUI is first invoked. Once the GUI is initialized, the invoking command-line is examined for extra options. If any extra options (other than *-gui*, *-project*, or any of the verbosity related options like *-debug* and/or *-verbose*) are included in the command line, these options are used to launch a batch-mode *vrun* process, the output of which is displayed in the **Transcript** window. If the resulting *vrun* command initiates a regression run, then a **VRM Results** window is opened to monitor the status of the regression run (in addition to the log output being displayed in the **Transcript** window).

Note that, in some cases, the output of the batch-mode *vrun* process may be informational (for example, the command *vrun -show -all nightly*, which emits a list of Actions under the “nightly” Runnable in lieu of initiating a regression run). Informational commands display their output in the **Transcript** window but no **VRM Results** window is created (since no regression run was launched). In this way, a GUI user can still take advantage of the informational modes of *vrun* such as *-status* or *-help*.

Note that viewing the log output of a batch-mode *vrun* process in the **Transcript** window only works for regression runs launched in the current GUI session. Regression runs launched from another VRM GUI session should generate a log file that can be viewed as a file. Regression runs launched outside the GUI infrastructure may or may not capture their log output in a file, depending on the command used to launch the process.

## Initial Start-up and Project Management

Users migrating from previous versions of the VRM GUI or from an entirely batch-mode flow may want to use the GUI to track the progress of a single regression run without having to set up a Project file with a single configuration. To accommodate those users while providing a smooth transition path to the Project file use model, the VRM GUI supports a “legacy mode” at start-up.

Any time the VRM GUI is launched, the **VRM Cockpit** window is created automatically. This singleton **VRM Cockpit** window remains in existence for the duration of the GUI session and closing the **VRM Cockpit** window also closes the application. If a VRM Project file is specified using the *-project* command-line option, then the contents of the Project file are loaded into the **VRM Cockpit** window. The title bar of the **VRM Cockpit** window displays the name of the loaded project file. Any changes to a loaded project are automatically saved to the current project file.

If you do not specify a Project file with the *-project* file option, the following priority order determines the startup behavior:

1. If a *default.vrm* project file exists in the current directory, it is automatically loaded.
2. If a *default.rmdb* RMDB file exists in the current directory, it is used to create an in-memory project.
3. A project selection dialog box is displayed allowing you to choose from the eight most recently used project files.

If a project file is selected via the **Open VRM Project...** dialog, the project file is loaded as if it were specified on the command line. If the **Cancel** button is selected on the window, the **VRM Cockpit** remains blank and the title bar will not show any loaded project file. If RMDB files or configurations are added to the **VRM Cockpit**, they will be reflected in the **VRM Cockpit** (and the internal data structures) but will not be saved to any external file. If the user attempts to load another project or exit the application while unsaved project data exists, a dialog will warn the user and prompt the user to save the current **VRM Cockpit** data to a project file. If the user turns down this offer, then the unsaved project data is discarded.

If a regression run starts as the result of the command-line options used to invoke the GUI, a **VRM Results** window is opened to monitor the results of the newly launched regression run. In this case, the **VRM Results** window is opened in the same tab-set as the **VRM Cockpit**. Since the window layout is similar to that used in 10.0 version, this is called “legacy mode.” The newly launched regression is added to the **VRM Cockpit** as a new configuration but the new **VRM Cockpit** content is not saved to disk unless a project is loaded. When the user exits the tool, the user is prompted as to whether or not to save the data to a project file and, if yes, then presented with the **Save As...** dialog.

A **Save Project As...** menu item allows the user to save the current RMDB and configuration data to a Project file irrespective of whether that data came from a loaded project file or was the result of a manually launched regression run.

**select -> File -> Save Project  
As...**

When the **Save Project As...** menu item is used to write to a project file, that file becomes the new “loaded project” and any subsequent changes are saved to that file automatically.

When starting up the VRM GUI, any command-line options besides `-gui`, `-project`, and the verbosity options (`-quiet`, `-verbose`, and `-debug`) are considered non-GUI options. The presence of non-GUI options on the command line used to start the VRM GUI are passed to a batch-mode *vrun* process. This allows the user to start a GUI-monitored regression simply by adding the `-gui` option to the existing *vrun* command. However, the command-line option data is saved to the **VRM Cockpit** (and a **VRM Results** window is opened) only if the non-GUI command-line options result in a regression run being launched.

## Start-up and the `toprunnables` Attribute

The batch-mode *vrun* application supports a *toprunnables* attribute in the *rmdb* element of the RMDB file. In batch-mode, this attribute is used if no other Runnables are selected for execution on the *vrun* command line. In other words, this attribute lists one or more Runnables that are considered selected unless the *vrun* command line selects an alternate set of Runnables. This allows *vrun* to be used like *make* (typing just the command *vrun* at the shell prompt to initiate a regression run).

When a *vrun* command line includes the `-gui` option to launch a GUI session, the *toprunnables* attribute is ignored. The reason is that there would otherwise be no way to open a GUI session if there was a *default.rmdb* file in the current directory that happened to contain a *toprunnables* attribute.

Instead, the *toprunnables* attribute is used to select Runnables in the configuration if there are no other Runnables selected. Without the *toprunnables* attribute, new configurations are created with nothing selected for execution and the user must select which Runnables to run when that configuration is used to launch a regression. If the *toprunnables* attribute is found in the RMDB file, then new configurations are created with the Runnables listed in the *toprunnables* attribute preselected. The user can, of course, change the selected Runnables for that configuration. Or the user can return to the default selection by de-selecting all Runnables and saving the configuration. This provides a behavior that closely matches that which is followed by the batch-mode *vrun* application.

## Relationship Between Configurations and VRM Data Directories

A *VRM Data* directory provides a structured area within the file system where the Actions that make up a regression run can be launched concurrently without the possibility of file collisions. Subdirectories are created under the *VRM Data* directory based on the hierarchical topology of the regression suite. Every regression run that is launched in a given *VRM Data* directory is logged in the *logs* directory under the top-level *VRM Data* directory. This is true regardless of whether the regression run is launched from within a GUI session, manually at a shell prompt, or by an automated process such as cron.

Each configuration points to a single *VRM Data* directory. There is no limit to how many configurations may point to a single *VRM Data* directory. In some cases, such as when the regression suite contains several subsets or tests that may be run individually, it might make sense to point multiple configurations at a single *VRM Data* directory. In other cases, it may make sense to change the *VRM Data* directory for a given configuration, such as when launching an experimental run that should not be preserved as part of the history of that configuration (though, in that case, a clone configuration, pointing to some other *VRM Data* directory, may be preferable).

The thing to remember is that configurations have no inherent association with *VRM Data* directories except through the user-configurable *VRM Data* Directory setting in the **General** tab of the **Edit VRM Configuration** window (see [Figure 4-4](#) on page 149). If multiple configurations point to the same *VRM Data* directory, the *History* row under each configuration is populated with the superset of all regressions run in that *VRM Data* directory and the same set of Status Event Logs and other files appear under all configurations that point to that *VRM Data* directory. This is true even if the configurations pointing to a given *VRM Data* directory are based on different RMDB files. Moreover, the user may change the settings in any given configuration after one or more regression runs have been executed.

Each Status Event Log carries with it a record of the RMDB file, the Runnable selections, and the command-line options used during that regression run. Viewing the results contained in any given single Status Event Log should be presented correctly, whether those results agree with the current project file settings for that configuration or not. However, viewing the cumulative history of a *VRM Data* directory that contains a heterogeneous collection of Status Event Logs based on various configurations or various RMDB files could present a challenge.

Therefore, every new configuration that is created is initially assigned a unique *VRM Data* directory, based on the *VRM Data* directories that already exist in the directory from which the GUI session was launched. Accepting this suggested *VRM Data* directory setting, and allowing the GUI to create the new *VRM Data* directory at the time the configuration is created, helps ensure that each configuration's history is kept separate from that of all other configurations. This is the most general use model, although the tool continues to provide the same flexibility in the GUI that is provided in the batch-mode *vrun* application because it is recognized that there

can be legitimate reasons for not enforcing a one-to-one correspondence between configurations and *VRM Data* directories.

## Menus

---

The next several sections describe the menus used to initiate operations in GUI mode. To review general terminology, every window type in Questa has an “active window menu” that shows up in the menu bar of the main window whenever an instance of that window type is activated.

The active window menu also shows up in response to a right-mouse button (RMB) click, in which case it is referred to as the context menu. When a window is “undocked” (that is, moved from the application’s main window to a top-level window of its own), the individual items of the context menu are divided among the menu bar menus in the undocked window, according to the functionality of those menu items. In addition, some context-menu items also appear under the menu bar menus in the application main window.

**Menu Item Descriptions .....** 176

## Menu Item Descriptions

The VRM GUI application main menu, the **VRM Cockpit** undocked and context menus, and the **VRM Results** undocked and context menus share much of their functionality. The following sections describe groups of menu items, in no particular order. Following the menu item descriptions is a “map” of the contents of the primary VRM menus in outline format.

Common menus (such as the Window and Help menus) are not documented here, as they mostly replicate functionality found in the existing *vrun* and *vsim* applications. Context menus attached to windows inherited from *vsim* (such as the UCDB Browser, Verification Tracker, Results Analysis, Transcript window, and so on) are also not documented here for the same reason. Refer to the *Questa Sim User’s Manual* for information.

Note in this document, the menus specific to the various GUI windows are referred to as “context menus.” This is a term that refers not only to the real context menu (the one invoked by the Right Mouse button) but also to the “Active Window Menu” which appears in the main-window menu bar when an instance of a given window type is currently active. In the case of undocked windows, the components of the context menu are distributed among the menu bar menus attached to the undocked pane’s window.

<b>Project Control</b> .....	<b>176</b>
<b>Project Creation and Maintenance</b> .....	<b>178</b>
<b>Tree Control</b> .....	<b>179</b>
<b>Regression Launch</b> .....	<b>179</b>
<b>View VRM Cockpit Contents</b> .....	<b>181</b>
<b>Viewing Regression Contents</b> .....	<b>182</b>
<b>Job Control</b> .....	<b>184</b>
<b>Display Refresh</b> .....	<b>185</b>

## Project Control

The GUI menu times provide control over VRM projects.

### New VRM Project

This menu item clears the **VRM Cockpit**, including the “loaded project” flag. This allows the creation of new project content (that is, configurations) from scratch. The new project is not associated with a project file until it is saved using the **Save Project As...** menu item (or answering yes when prompted to save the data when loading a new project file or quitting the application). If unsaved data already exists when this menu item is invoked, the user is prompted to save the unsaved data. This menu item is always enabled.

**select -> File -> New VRM Project**

## Open VRM Project...

This menu item presents a dialog for the selection of a project file. The eight most recently used project file paths are available in a drop-down box for easy selection. An existing project file may be selected via a file browser by clicking the **Browse...** button. A new project file can be created by entering the new file path/name into the window's entry box. If unsaved data already exists when this menu item is invoked, then the user is prompted to save the unsaved data. This menu item is always enabled.

**select -> File -> Open VRM  
Project...**

## Save Project As...

This menu item presents a dialog for the selection of a project file into which the current **VRM Cockpit** contents is saved. If the selected file already exists, the user is prompted to verify whether it is OK to overwrite the existing file. This menu item is always enabled.

**select -> Save Project As...**

## Import — VRM Data Directory

This menu item presents a dialog via which an existing **VRMDATA** directory may be specified. The command-line options and Runnable selections from the most recent Status Event Log located in this directory are used to create a new configuration pointing to the **VRMDATA** directory in question. This menu item is always enabled.

**select in VRM Cockpit -> Import  
-> VRM Data Directory...**

## Import — VRM Project File

This menu item allows the user to select an existing project file whose contents is imported over the top of the project data already loaded into the **VRM Cockpit**. Duplicate RMDB files will not be added to the **VRM Cockpit**. Configurations whose name and RMDB file match a configuration already present in the **VRM Cockpit** will overwrite the existing configuration. This menu item, in conjunction with the **Export** menu item described below, allow users to share their configurations. This menu item is always enabled.

**select in VRM Cockpit -> Import  
-> VRM Project File...**

## Export...

This menu item causes the RMDB files and configurations selected in the **VRM Cockpit** to be written to a file of the user's choice. The RMDB files for selected configurations are written to the new project file even if they were not originally selected. This menu item is enabled whenever one or more RMDB files or *RunConfiguration* rows are selected in the **VRM Cockpit**.

**select in VRM Cockpit -> Export...**

## Project Creation and Maintenance

The following menu items are found in the active window/context menu of the **VRM Cockpit** window.

### Add Item — RMDB File

This menu item allows the user to select an RMDB file to add as a top-level item in the **VRM Cockpit**. This menu item is always enabled.

**select in VRM Cockpit -> Add Item -> RMDB File...**

### Add Item — Run Configuration

This menu item presents a dialog whereby the user may set-up the command-line options and Runnable selections for a new configuration under the selected RMDB file. This menu item is enabled whenever a single RMDB row is selected in the cockpit.

**select in VRM Cockpit -> Add Item -> Run Configuration...**

### Remove Selected Items

This menu item removes the selected RMDB file and/or configuration rows from the **VRM Cockpit**. If an RMDB file row is selected, all configurations under that RMDB file row are also removed from the **VRM Cockpit** (in this case, the user is prompted for verification before the actual removal). Only the **VRM Cockpit** entries are removed; the actual RMDB files, VRMDATA directories, and Status Event Logs are not deleted from the file system. This menu item is enabled whenever one or more RMDB files or configuration row is selected in the **VRM Cockpit**.

**select in VRM Cockpit -> Remove Selected Items**

### Edit Configuration...

This menu item invokes the **Edit VRM Configuration** window populated with the command-line options and Runnable selections from the configuration selected in the **VRM Cockpit**. If the window is closed with the **OK** button, then the new settings will replace the old settings (and the configuration is renamed if the name string was changed). This menu item is enabled whenever a single configuration row is selected in the **VRM Cockpit**.

Note that if the user changes the configuration name so that it is no longer unique, a warning dialog appears and the user is returned to the **Edit VRM Configuration** window to change the name to a unique name. No two configurations may exist with the same name in any given project or project file.

**select in VRM Cockpit -> Edit Configuration...**

## Clone Configuration

This menu item invokes the **Edit VRM Configuration** window populated with the command-line options and Runnable selections from the configuration selected in the **VRM Cockpit** but, unlike the **Edit Configuration...** menu item, a new unique name is generated and the **OK** button causes a new configuration to be added to the **VRM Cockpit**. This menu item is used to make an exact or nearly exact copy of an existing configuration. This menu item is enabled whenever a single configuration row is selected in the **VRM Cockpit**.

**select in VRM Cockpit -> Clone Configuration...**

## Capture Configuration

This menu item invokes the **Edit Regression Options** window populated with the command-line options and Runnable selections extracted from the regression run *History* row selected in the **VRM Cockpit**. This provides a way to generate a new configuration from a previous VRM regression run (possibly a manually invoked run imported with the **Import...** menu item. This menu item is enabled whenever a single regression *History* row is selected in the **VRM Cockpit**.

**select in VRM Cockpit -> Capture Configuration...**

## Tree Control

The following menu items are found in the **Edit** menu of the main window menu bar, in the **Edit** menu of the undocked **VRM Cockpit** window, in the **Edit** menu of the undocked **VRM Results** window, and in the active window/context menus the **VRM Cockpit** window.

### Select — All/None

These menu items control the selection of items in the active window.

### Expand — Selected/All

These menu items control the expansion of items in the active window.

### Collapse — Selected/All

These menu items control the collapse setting of items in the active window.

## Regression Launch

The following menu items are found in the **Run** menu of the undocked **VRM Cockpit** window and **VRM Results** window and in the active window/context menus of the same two windows. The behavior differs based on what has been selected. If a *Run Configuration* row is selected in the **VRM Cockpit** window, a regression run is launched based on the command-line options

and Runnable selections stored in the selected configuration. If a regression run *History* line is selected in the **VRM Cockpit** window, a regression run is launched based on the command-line options and Runnable selections in effect when the selected regression run is executed.

If the **VRM Results** window is active, a regression run is launched using the same command-line options and Runnable selections in effect for the regression run whose status is being displayed by the active window.

If the regression run to be launched is based on a previous regression run (either because a regression run *History* row is selected in the **VRM Cockpit** or because the **VRM Results** window is active), the menu will read **Run Again** instead of **Run**.

If the cockpit window is active, this menu item is enabled whenever a single *RunConfiguration* or regression run *History* row is selected. If the **VRM Results** window is active, this menu item is always enabled.

### Run (Again)

This menu item causes a regression run to be launched.

**select -> VRM Cockpit -> Run  
Again**

### Edit Config and Run (Again)

This menu item causes the **Edit VRM Configuration** window to appear populated with the command-line options and Runnable selections that would be used to launch the regression if a regression were actually launched. Canceling this menu will abort the run. The **OK** button text is changed to **Run** to indicate that the configuration setting will not be saved, but will only be used to launch the regression run at hand. This menu item can be used to make one-time changes to the configuration settings before launching the regression run.

**select -> VRM Cockpit -> Run  
Special -> Edit Config and Run...**

### Clone Config and Run (Again)

This menu item invokes the **Edit VRM Configuration** window populated with the command-line options and Runnable selections from the configuration selected in the **VRM Cockpit**, a new unique name is generated, and the **OK** button causes a new configuration to be added to the **VRM Cockpit**. This menu item is used to make an exact or nearly exact copy of an existing configuration.

**select -> VRM Cockpit -> Run  
Special -> Clone Config and Run Again...**

## One-time Edit and Run (Again)

This menu item invokes the **Edit Regression Options** window populated with the command-line options and Runnable selections extracted from the regression run *History* row selected in the **VRM Cockpit**. This provides a way to generate a new configuration from a previous regression run.

**select -> VRM Cockpit -> Run**  
**Special -> One-time Edit and Run...**

## View VRM Cockpit Contents

The following menu items are found in the active window/context menu of the **VRM Cockpit** window.

### View Results

This menu item opens a **VRM Results (view)** window and loads it with the results from the regression run *History* row selected in the **VRM Cockpit**. If the selected regression is still running, the **VRM Results** window will connect to the running regression after reading the Status Event Log file and updating the status display accordingly. This menu item is enabled whenever a single regression run *History* row is selected in the **VRM Cockpit**.

**select -> VRM Cockpit -> View**  
**Results**

### View History

This menu item causes a **VRM Results (history)** window to be opened and loaded with a summary of the history of all the regression runs executed in the VRMDATA directory attached to the currently selected *RunConfiguration* row in the **VRM Cockpit**. This menu item is enabled whenever a single *Run Configuration* row is selected in the **VRM Cockpit**.

**select -> VRM Cockpit -> View**  
**History**

### View Vrun Output

This menu item opens a source window and loads it with the standard output from the *vrun* process associated with the selected regression run. The data loaded is a snapshot of the contents of the file at the time the file is read; it does not “follow” the output in real time if the regression is still running. This menu item is enabled whenever a single regression run *History* row is selected in the **VRM Cockpit**.

**select -> VRM Cockpit -> View**  
**Vrun Output**

## Viewing Regression Contents

The following menu items are found in the **View** menu of the main window menu bar.

### Action Log File

This menu item opens a source window and loads it with the log output of the VRM Action currently selected in the **VRM Results** window. This menu item is enabled whenever an Action row is selected in the **VRM Results** window.

```
select -> VRM Results -> View  
-> Action Log File
```

### Vrun Output

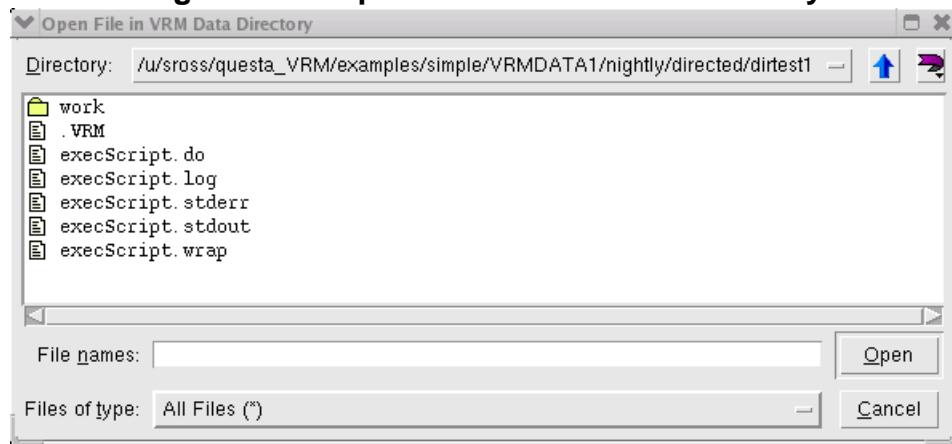
This menu item opens a source window and loads it with the standard output from the *vrun* process associated with the active **VRM Results** window. The data loaded is a snapshot of the contents of the file at the time the file is read; it does not “follow” the output in real time if the regression is still running. This menu option is always enabled.

```
select -> VRM Results -> View  
-> Vrun Output
```

### File in VRM Data

This menu item opens the **Open File in VRM Data Directory** window (see [Figure 4-16](#)). It allows the user to browse the VRMDATA directory contents and open an arbitrary file. If a file is selected in the file browser, the file type is consulted to determine how to “open” that file in a GUI window. This menu item is always enabled.

**Figure 4-16. Open File in VRM Data Directory**

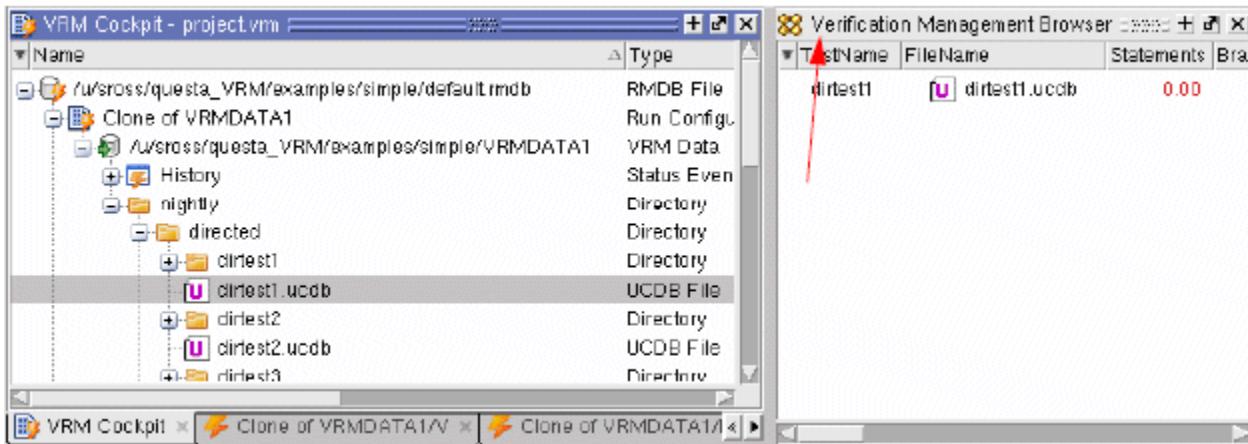


```
select -> VRM Results -> View  
-> File in VRM Data...
```

## UCDB Browser

This menu item invokes the UCDB file generated by the VRM Action selected in the **VRM Cockpit** window and loads it into the **Verification Management Browser** (see [Figure 4-17](#)). The Browser is opened if it is not already open. This menu item is enabled whenever a single VRM Action row is selected in the **VRM Cockpit** window and the selected Action produced a UCDB file.

**Figure 4-17. Verification Management Browser**



select -> VRM Cockpit -> View  
> UCDB Browser

## Merge File in UCDB Browser

This menu item finds the merged UCDB file associated with the VRM Action or group selected in the **VRM Results** window (as indicated by the *mergefile* parameter in the RMDB file) and loads that file into the **UCDB Browser**. The **UCDB Browser** is opened if it is not already visible. This menu item is enabled whenever a single VRM Action or group is selected in the **VRM Results** window and a *mergefile* parameter is defined for the selected Action or group in the RMDB file.

## Merge File in Verification Tracker

This menu item finds the merged UCDB file associated with the VRM Action or group selected in the **VRM Results** window (as indicated by the *mergefile* parameter in the RMDB file) and loads that file into the **Verification Tracker**. The **Verification Tracker** is opened if it is not already visible. This menu item is enabled whenever a single VRM Action or group is selected in the **VRM Results** window and a *mergefile* parameter is defined for the selected Action or group in the RMDB file.

## Results Analysis Menu

This menu item finds the Triage Database (TDB) file associated with the VRM Action or group selected in the **VRM Results** window (as indicated by the *triagefile* parameter in the RMDB

file) and loads that file into the **Results Analysis** window. The **Results Analysis** window is opened if it is not already visible. This menu item is enabled whenever a single VRM Action or group is selected in the **VRM Results** window and a *triagefile* parameter is defined for the selected Action or group in the RMDB file.

## Counters Follow Filter

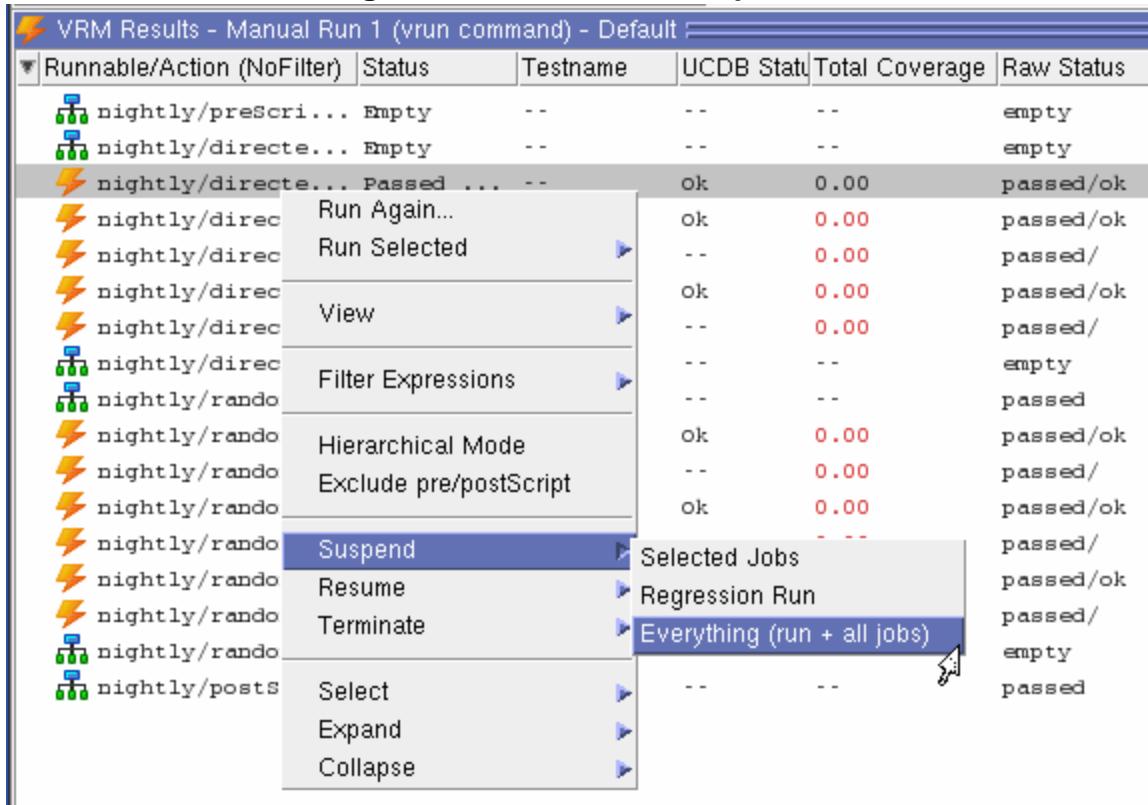
By default, the counters at the bottom of the VRM Results window to match the filter (**VRM Results > Filter Expressions**). You can enable/disable this feature with the **VRM Results > Counters Follow Filter** menu item.

## Job Control

The following menu items can be found in the active-window/context menu of the **VRM Results** window.

Refer to [Figure 4-18](#). These menu items are used to control an active regression run and/or the jobs it has launched.

**Figure 4-18. Job Control Options**



## Suspend

The menu items under this sub-menu are used to pause, but not kill, a regression run and/or one or more jobs.

## Resume

The menu items under this sub-menu are used to restart a regression run and/or one or more jobs which were paused by a menu item under the **Suspend** sub-menu.

## Terminate

The menu items under this sub-menu are used to terminate (that is, kill) a regression run and/or one or more jobs.

Under each of these sub-menus are three menu items which all have similar semantics as follows:

### Selected Jobs

The operation indicated by the menu item in question is applied to the jobs associated with the VRM Actions elected in the **VRM Results** window.

### Regression Run Menu Item

The operation indicated by the menu item in question is applied to the regression run as a whole. Individual jobs that have already been launched are not affected.

### Everything (run + all jobs)

The operation indicated by the menu item in question is applied to both the regression run as a whole and to all individual jobs that have already been launched.

## Display Refresh

Use this menu to alter the display.

### Refresh Display

This menu item causes the contents of the **VRM Cockpit** window to be reloaded. This may be necessary if manual changes have been made to the contents of any of the **VRMDATA** directories associated with configurations currently loaded into the **VRM Cockpit**. Changes that are initiated from within the VRM GUI (such as manually launching a regression run by entering a *vrun* command at the transcript prompt) will automatically refresh the **VRM Cockpit** display so as to reflect the current **VRMDATA** directory contents. However, once loaded, the VRM GUI does not watch for asynchronous changes from outside the VRM GUI itself.

## GUI Use Model

---

The use model assumes the user is running Questa on Linux®<sup>1</sup> or some other shell-based platform.

<b>Getting Started with the GUI .....</b>	<b>187</b>
<b>Start a New Regression Run and/or View Results of Previous Regression Run .....</b>	<b>192</b>
<b>Analyze Results and Rerun Tests Based on the Analysis.....</b>	<b>203</b>
<b>Counter Control in the VRM GUI.....</b>	<b>204</b>

---

1. Linux® is a registered trademark of Linus Torvalds in the U.S. and other countries.

## Getting Started with the GUI

These start-up procedures will get you up and running quickly.

<b>Launching a New Regression from the VRM GUI Command Line .....</b>	<b>187</b>
<b>Building Up a New Project File from Nothing .....</b>	<b>187</b>
<b>Generate a Project File by Invoking an Existing Batch-mode vrun in GUI Mode .....</b>	<b>190</b>
<b>Generate a Configuration by Invoking vrun at Transcript Prompt.....</b>	<b>190</b>
<b>Configuring the GUI to Open a Text File in an External Editor .....</b>	<b>190</b>

## Launching a New Regression from the VRM GUI Command Line

The most common way a new user would migrate from batch-mode use (or from 10.0 use) to 10.1 GUI-based use is to append the *-gui* command-line option onto an existing batch-mode command. For example,

**vrun nightly -gui**

In this case, the GUI is launched and the presence of the “nightly” command-line option triggers a regression run. The command-line options used in this regression run are stored to a configuration in the **VRM Cockpit** although, unless the *-project* command-line option is also specified, no project file will be loaded. Once the regression run starts, the **VRM Results** window is invoked, thus obscuring the **VRM Cockpit** window. Because of the similarity between this arrangement and the 10.0 VRM GUI, this mode is internally referred to as “legacy a.

The user may view the **VRM Cockpit** and/or save the project data into a project file at any time. If the user fails to save the data, then the user is prompted to save the data on exit.

## Building Up a New Project File from Nothing

When a user first starts out with the VRM GUI, most likely the user does not have a project file.

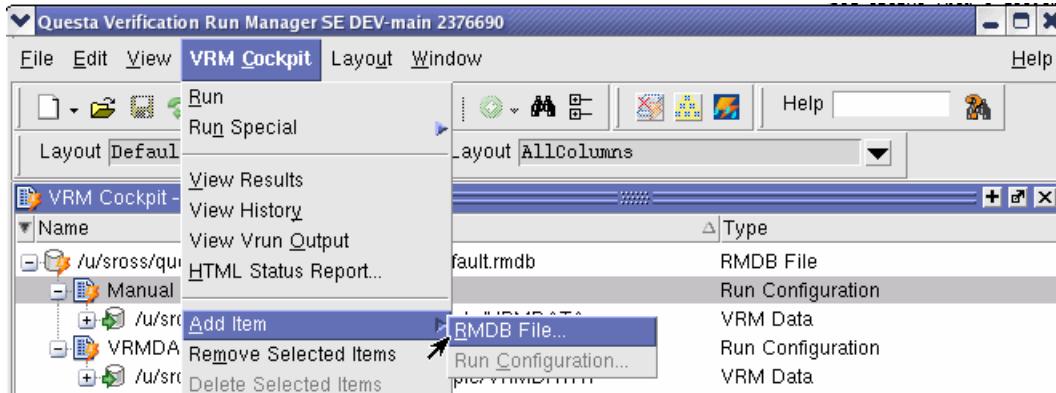
Following are the steps to create a project file:

1. Type *vrun -gui* at the shell prompt.
2. The GUI prompts the user for a project file.
3. The user should enter an appropriate file name (a *.vrm* extension is added if necessary).
4. The GUI informs the user that the file does not exist and asks whether it is OK to create the file.
5. A blank project file is created.

At this point, the following steps can be used to add one or more RMDB files to the project:

1. Select **Add Item -> RMDB File...** from the **VRM Cockpit** context menu (see [Figure 4-19](#)).
2. This invokes the **Select RMDB File** window.
3. The user should select the RMDB file with which to work.

**Figure 4-19. Add Item RMDB File**



The following steps can be used to add one or more configurations under a given RMDB file:

1. Select **Add Item -> Run Configuration...** from the **VRM Cockpit** context menu.
2. This invokes the **Create VRM Configuration** window with a unique dummy name (**Run Template 1** as shown in [Figure 4-20](#)).
3. The user should enter an appropriate name for the configuration, select (or verify) the **VRMDATA** directory path, and configure the command-line options to be used for this configuration.
4. When the user presses **OK** to close the window, the configuration is created and displayed under the selected RMDB file.

**Figure 4-20. Create VRM Configuration Name**

Each time a new configuration is created, a unique *VRM Data* path is generated and loaded into the **Create VRM Configuration** window as a suggestion. If the user decides to accept this suggested directory, or if the user changes it to a new directory which does not already exist, a dialog appears asking whether the directory should be created. It is OK not to create the directory at this point, since the first regression run launched using this configuration will automatically create the *VRM Data* directory. However, leaving the directory uncreated at this point could lead to the same directory name being suggested for a subsequent configuration.

The contents of the project file are updated every time an RMDB or configuration is added to or deleted from the project view in the **VRM Cockpit** and any time the command-line options or Runnable selections are changed in any of the named configurations. There is no predetermined limit to the number of RMDB files that can be added to a single project, nor to the number of configurations that can be added to each RMDB file. Both RMDB files and configurations may be deleted from the project by selecting the item to be deleted and using the **Remove Selected Items** of the **VRM Cockpit** context menu.

In order to bypass the dialog prompting for a project in subsequent runs, the *-project* command-line option can be used. For example,

```
vrun -gui -project myproject.vrm
```

## Generate a Project File by Invoking an Existing Batch-mode vrun in GUI Mode

As mentioned above, any time a batch-mode regression is launched at the same time that a GUI session is launched, a configuration corresponding to the command-line options used to launch the batch-mode regression is added to the **VRM Cockpit**.

If the *-project* command-line option was also used to open a valid project file, then this new configuration will automatically be saved to the project file. If no project file is loaded, then the user may use the **File -> Save Project As...** main-window menu item to save the configuration (currently in the **VRM Cockpit**) to a new project file.

## Generate a Configuration by Invoking vrun at Transcript Prompt

This section describes generating a configuration by invoking *vrun* at the **Transcript** window prompt with a blank project file.

If the user has an existing batch-mode *vrun* command and a running GUI session, the user can type the *vrun* command into the transcript window (see [Figure 4-15](#) on page 170) at the prompt (or copy-and-paste the command into the **Transcript** window). If a regression run is launched as a result of the *vrun* command, a **VRM Results** window is opened to display the status of the run and a new configuration is created in the **VRM Cockpit** using the Runnable selections and command-line options specified in the *vrun* command. If a project file has been opened, then the new configuration is saved to the project file automatically.

Whenever a batch-mode regression run is initiated by manually typing a *vrun* command, whether it is at the shell prompt when the GUI session is initiated or at the **Transcript** window prompt of a running GUI session, the output of the batch-mode *vrun* process is echoed to the **Transcript** window. If the *vrun* command requests information (as opposed to launching a regression run), then that information is displayed in the **Transcript** window.

## Configuring the GUI to Open a Text File in an External Editor

Set up your environment to open text files in an external editor.

### Prerequisites

- Know the directory of your text editor's executable.
- Have a Verification Run Manager GUI open.

## Procedure

1. Select **tools > Edit Preferences**.

The **Preferences** dialog box opens.

2. Select the **By Name** tab.
3. Expand the Main hierarchy.
4. Select the Editor entry.
5. Click **Change Value**.

The **Change Main Preference Value** dialog box opens.

6. Enter your text editor's executable path in the text entry box.
7. Click **OK**.
8. Click **OK**.

## Results

You can now use the **Open in External Editor** context menu item when you right-click in an open text file.

# Start a New Regression Run and/or View Results of Previous Regression Run

This section explains how to start new or view previous regression runs.

<b>Launch a New Regression Run from a Configuration Row .....</b>	<b>192</b>
<b>View Results of a Previously Launched Regression Run.....</b>	<b>193</b>
<b>Launch a New Regression Run from a History Row .....</b>	<b>194</b>
<b>Launch a New Regression from VRM Results Window of a Previous Regression Run</b>	<b>195</b>
<b>Launch a New Regression from the Transcript Prompt .....</b>	<b>195</b>
<b>Import the Results of Manually Launched Regression Runs into a Project .....</b>	<b>196</b>
<b>Create a New Configuration that Points to an Existing VRM Data Directory .....</b>	<b>198</b>
<b>Change Value of the VRM Data Directory After Regression Data is Accumulated...</b>	<b>198</b>
<b>Launch Multiple Regression Runs in a Single VRM Data Directory.....</b>	<b>200</b>
<b>Delete File and/or Status Event Log from the VRM Data Directory .....</b>	<b>201</b>

## Launch a New Regression Run from a Configuration Row

To launch a new regression run based on a configuration stored in the currently loaded project file, the user does the following:

1. Select the appropriate configuration in the **VRM Cockpit** (only one configuration may be selected).
2. Use the **Run** context-menu item to initiate the regression run.

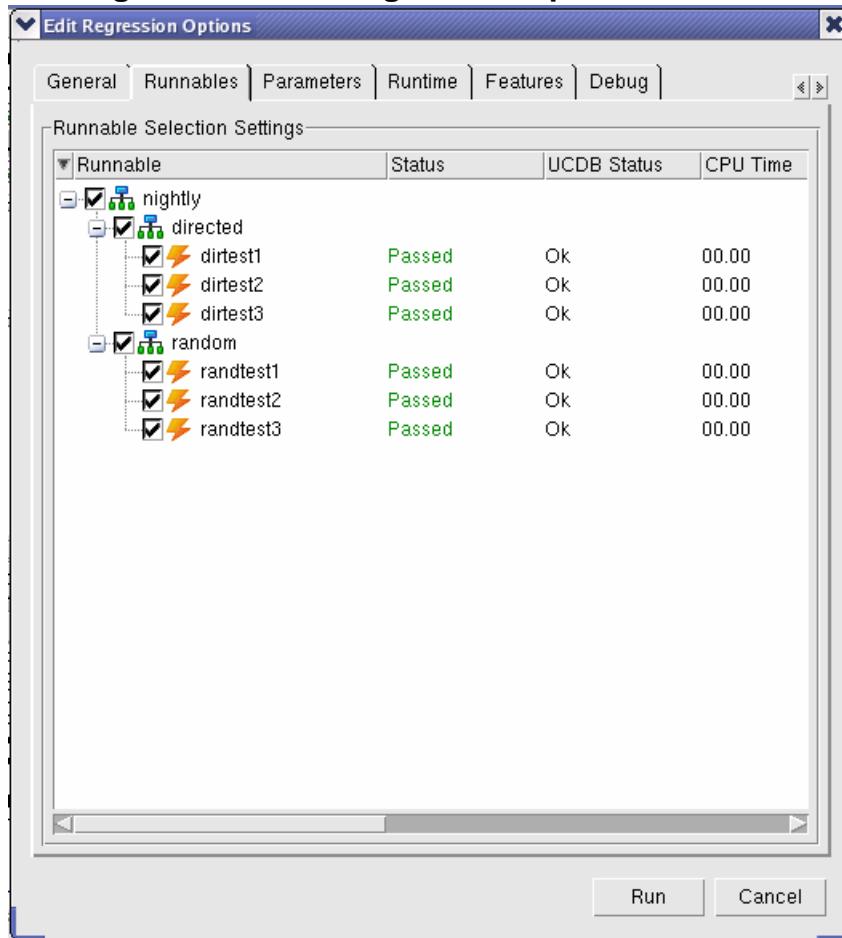
At this point, the **VRM Results** window opens to track the progress of the regression run. The window may be closed at any time without affecting the progress of the batch-mode regression run. If the user wishes to make a one-time change to the Runnable selections and/or command-line options prior to launching the regression run (for example, wants to change the output verbosity to *-verbose* for this one run without changing it in the configuration itself), then execute the following steps:

1. Select the appropriate configuration in the **VRM Cockpit** (only one configuration may be selected).
2. Use the **Run Special -> One-time Edit and Run...** context-menu item in the **VRM Cockpit** to initiate the regression run.

The **Edit Regression Options** window opens (see [Figure 4-21](#)) and the user can make changes to the **Runnable** selections and/or command-line options. The window's **OK** button changed to **Run** to emphasize that the changes are not being saved in the associated configuration but are only being used for this particular regression run.

Note that any time a batch-mode regression run is launched from a GUI menu selection, the output of the resulting *-vrun* process is not echoed to the transcript. The *vrun* output is saved to a log file under the *VRMData* directory and can be viewed via the **View Vrun Output** context-menu item in the **VRM Cockpit** window or the **View -> Vrun Output** context-menu item in the **VRM Results** window.

**Figure 4-21. Edit Regression Options Window**



## View Results of a Previously Launched Regression Run

The child entries listed under the *History* row for a given configuration (found under the single *VRM Data* row under the configuration itself) represent the various Status Event Logs generated by regression runs executed in that particular *VRMData* directory. The Status Event Logs each contain a complete record of a single batch-mode regression run. The timestamp listed in the **Name** column for that Status Event Log row shows the date and time at which the batch-mode regression run began (the batch-mode *vrun* process ensures this timestamp is unique for each regression run). The status of a given regression run may be viewed using the following steps:

1. Select the appropriate Status Event Log row in the **VRM Cockpit**.

2. Select **View Results** context-menu item in the **VRM Cockpit** window to view the regression results, which invokes the **VRM Results** window.

Multiple Status Event Log rows may be selected. The **VRM Results** window opens for each selected Status Event Log row containing the status of the batch-mode regression run that created that Status Event Log. If the regression run is still in progress, the **VRM Results** will attempt to attach to the status event stream of the running regression by connecting to the *port@host* socket listed in the Status Event Log. If such a connection is successful, the ongoing status of the regression run is shown in the **VRM Results** window in real time.

## Launch a New Regression Run from a History Row

Each Status Event Log contains enough information to rerun the associated regression run in exactly the same manner as when the Status Event Log was generated. To re-execute a given regression run, do the following:

1. Select the appropriate Status Event Log row in the **VRM Cockpit** (only one Status Event Log row may be selected).
2. Use the **Run Again** context-menu item in the **VRM Cockpit** to initiate the regression run.

At this point, the **VRM Results** window is opened to track the progress of the regression run. The **VRM Results** window may be closed at any time without affecting the progress of the batch-mode regression run.

If the user wishes to make a one-time change to the Runnable selections and/or command-line options prior to launching the regression run (for example, change the output verbosity to *-verbose* for this one run without changing it in the configuration itself), the user does the following steps:

1. Select the appropriate Status Event Log row in the **VRM Cockpit** (only one Status Event Log row may be selected).
2. Use the **Run Special -> One-time Edit and Run Again...** context-menu item in the **VRM Cockpit** to initiate the regression run.

The **Edit Regression Options** window opens and the user can make changes to the Runnable selections and/or command-line options. The window's **OK** button changes to **Run** to emphasize that the changes are not being saved in the associated configuration but are only being used for this particular regression run.

Note that any time a batch-mode regression run is launched from a GUI menu selection, the output of the resulting *-vrun* process is not echoed to the transcript. The *vrun* output is saved to a log file under the *VRM Data* directory and may be viewed via the **View -> Vrun Output** in the **VRM Cockpit** window.

## Launch a New Regression from VRM Results Window of a Previous Regression Run

A given regression run may be re-executed from the **VRM Results** window in which the results of the original regression run are currently being viewed, whether the **VRM Results** window was opened as a result of the original run or as a result of viewing the results of a previous run. The **Run Again...** context-menu item re-executes the regression run exactly as it was executed before, while the **VRM Cockpit -> Run Special -> One-time Edit and Run Again...** allows the user to modify the Runnable selection and/or command-line options for this one regression run only (as described in the sections above). The difference between re-executing a regression run from the **VRM Results** window and re-executing it from the **VRM Cockpit** window is that a re-execution from the **VRM Results** window reuses the existing **VRM Results** as is, thus overlaying the results of the new regression run onto the results from the original run.

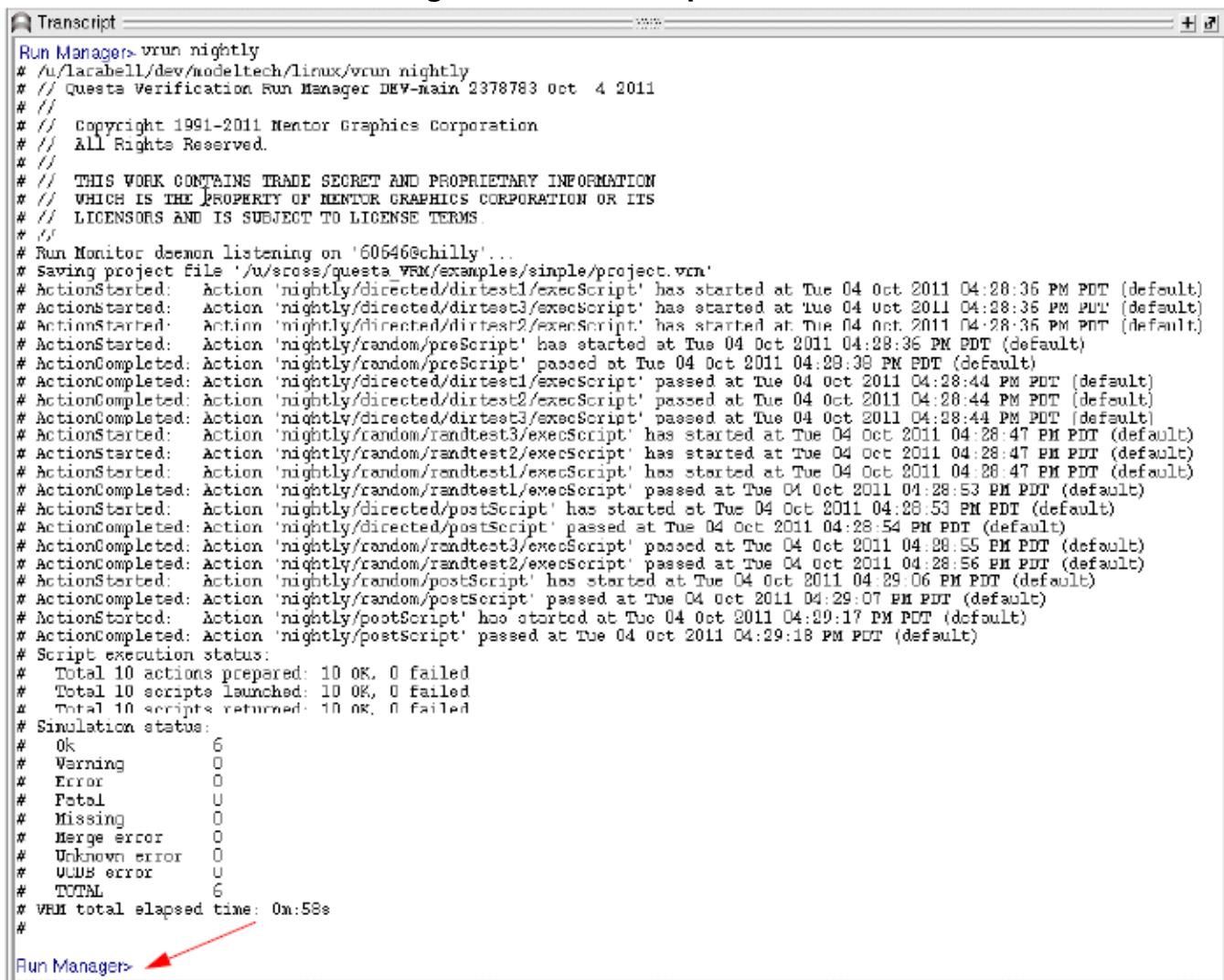
The **Run Selected** context-menu item transfers the selected Actions to the Run Configuration you select from the sub-menu and opens the Edit Regression Options dialog box, allowing you to initiate a re-run using that Run Configuration.

## Launch a New Regression from the Transcript Prompt

A batch-mode regression may be launched manually from within a GUI session by typing the *vrun* command and any appropriate command-line arguments (except for the *-gui* option) into the **Transcript** window at the prompt.

If the *vrun* command initiates a regression run, then the **VRM Results** window is opened to monitor the status of the resulting regression run and a new configuration corresponding to the manual *vrun* command is added to the **VRM Cockpit** window (see [Figure 4-22](#) on page 196). In addition, the output of the *vrun* process appears in the **Transcript** window. If the *vrun* command requests information rather than launching a regression run, then the output of the command is shown in the **Transcript** window, but no **VRM Results** window is opened and no configuration is created.

Figure 4-22. Transcript Window

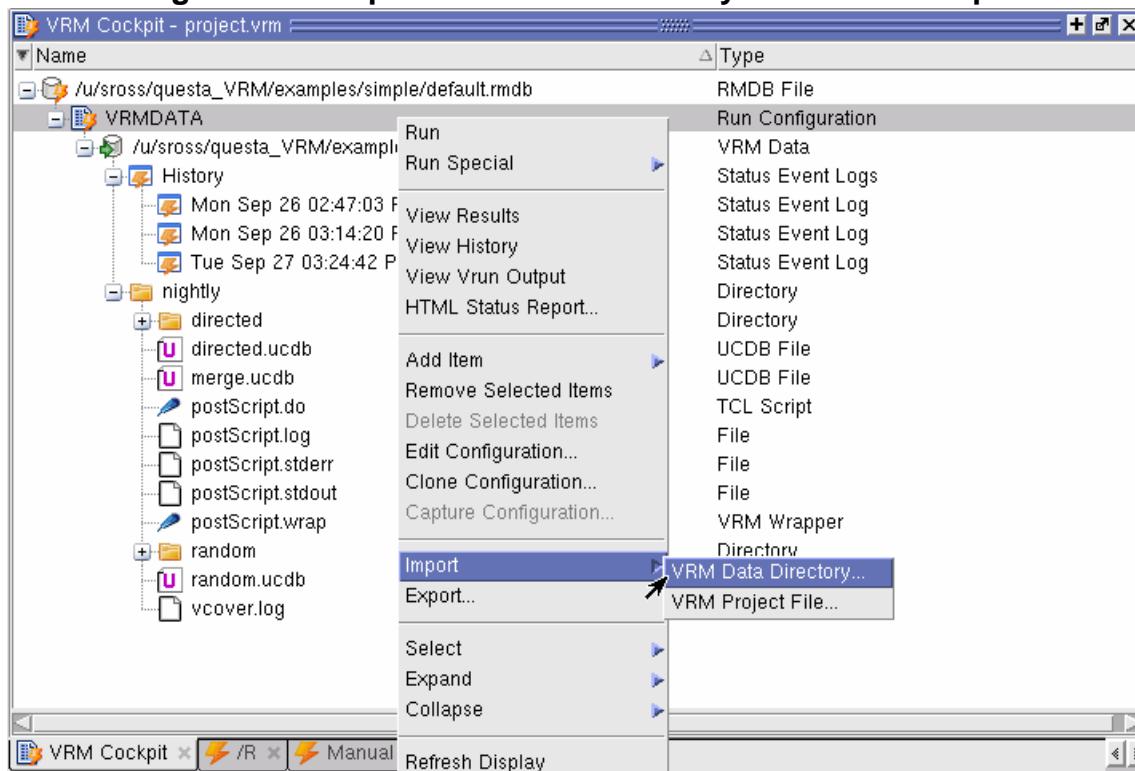


```
Transcript
Run Manager> vrun nightly
# /u/lacabell/dev/modeltech/linux/vrun nightly
# // Questa Verification Run Manager DEV-Main 2378783 Oct 4 2011
# //
# // Copyright 1991-2011 Mentor Graphics Corporation
# // All Rights Reserved.
# //
# // THIS WORK CONTAINS TRADE SECRET AND PROPRIETARY INFORMATION
# // WHICH IS THE PROPERTY OF MENTOR GRAPHICS CORPORATION OR ITS
# // LICENSORS AND IS SUBJECT TO LICENSE TERMS.
# //
# Run Monitor daemon listening on '60546@chilly'...
# Saving project file '/u/sross/questa_VRM/examples/simple/project.vrm'
# ActionStarted: Action 'nightly/directed/dirtest1/execScript' has started at Tue 04 Oct 2011 04:28:36 PM PDT (default)
# ActionStarted: Action 'nightly/directed/dirtest3/execScript' has started at Tue 04 Oct 2011 04:28:36 PM PDT (default)
# ActionStarted: Action 'nightly/directed/dirtest2/execScript' has started at Tue 04 Oct 2011 04:28:36 PM PDT (default)
# ActionStarted: Action 'nightly/random/preScript' has started at Tue 04 Oct 2011 04:28:36 PM PDT (default)
# ActionCompleted: Action 'nightly/random/preScript' passed at Tue 04 Oct 2011 04:29:38 PM PDT (default)
# ActionCompleted: Action 'nightly/directed/dirtest1/execScript' passed at Tue 04 Oct 2011 04:28:44 PM PDT (default)
# ActionCompleted: Action 'nightly/directed/dirtest2/execScript' passed at Tue 04 Oct 2011 04:28:44 PM PDT (default)
# ActionCompleted: Action 'nightly/directed/dirtest3/execScript' passed at Tue 04 Oct 2011 04:28:44 PM PDT (default)
# ActionStarted: Action 'nightly/random/randtest3/execScript' has started at Tue 04 Oct 2011 04:28:47 PM PDT (default)
# ActionStarted: Action 'nightly/random/randtest2/execScript' has started at Tue 04 Oct 2011 04:28:47 PM PDT (default)
# ActionStarted: Action 'nightly/random/randtest1/execScript' has started at Tue 04 Oct 2011 04:28:47 PM PDT (default)
# ActionCompleted: Action 'nightly/random/randtest1/execScript' passed at Tue 04 Oct 2011 04:28:53 PM PDT (default)
# ActionStarted: Action 'nightly/directed/postScript' has started at Tue 04 Oct 2011 04:28:53 PM PDT (default)
# ActionCompleted: Action 'nightly/directed/postScript' passed at Tue 04 Oct 2011 04:28:54 PM PDT (default)
# ActionCompleted: Action 'nightly/random/randtest3/execScript' passed at Tue 04 Oct 2011 04:28:55 PM PDT (default)
# ActionCompleted: Action 'nightly/random/randtest2/execScript' passed at Tue 04 Oct 2011 04:28:56 PM PDT (default)
# ActionStarted: Action 'nightly/random/postScript' has started at Tue 04 Oct 2011 04:29:06 PM PDT (default)
# ActionCompleted: Action 'nightly/random/postScript' passed at Tue 04 Oct 2011 04:29:07 PM PDT (default)
# ActionStarted: Action 'nightly/postScript' has started at Tue 04 Oct 2011 04:29:17 PM PDT (default)
# ActionCompleted: Action 'nightly/postScript' passed at Tue 04 Oct 2011 04:29:18 PM PDT (default)
# Script execution status:
# Total 10 actions prepared: 10 OK, 0 failed
# Total 10 scripts launched: 10 OK, 0 failed
# Total 10 scripts returned: 10 OK, 0 failed
# Simulation status:
# Ok          6
# Warning     0
# Error        0
# Fatal        0
# Missing      0
# Merge error  0
# Unknown error 0
# VCD error   0
# TOTAL       6
# VRM total elapsed time: 0m:58s
#
Run Manager>
```

## Import the Results of Manually Launched Regression Runs into a Project

If in a given environment, one or more batch-mode regression runs have been launched outside of the GUI environment, the results stored in the *VRM Data* directory can be imported into the **VRM Cockpit** (and, thus, into the current project file, if one is loaded) by doing the following:

**Import -> VRM  
Data Directory...**

**Figure 4-23. Import VRM Data Directory into VRM Cockpit**

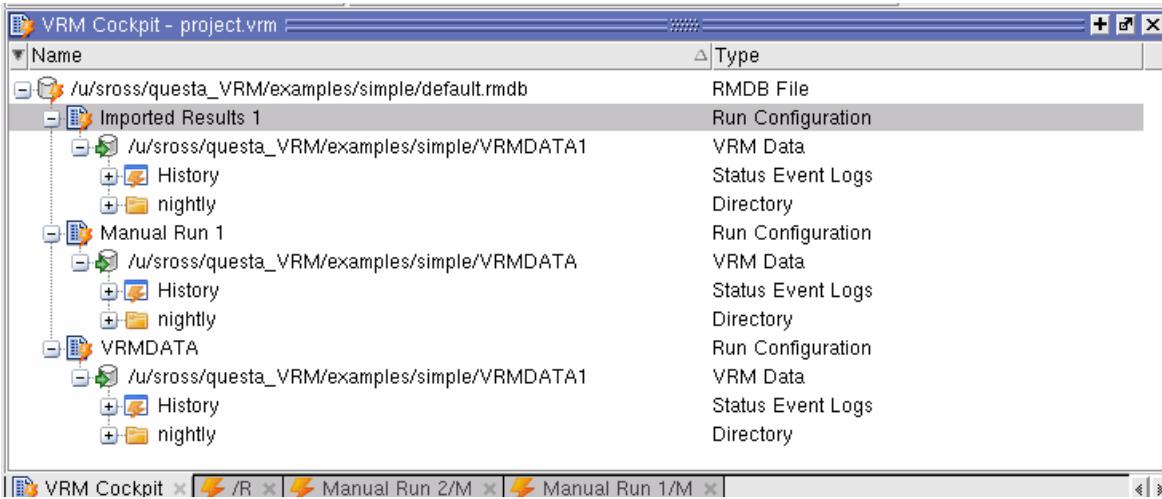
This invokes the **Select VRM Data Directory** window as shown in [Figure 4-24](#). Select a VRM Data directory for import and select **OK**.

**Figure 4-24. Select VRM Data Directory Window**

As a result, the application looks for the Status Event Log generated by the most recently launched batch-mode regression run. If no Status Event Logs are found, then no action is taken. If a Status Event Log is identified, then the RMDB file, Runnable selections, command-line options, and VRM Data directory path are extracted from this Status Event Log. If the RMDB file is not already present in the VRM Cockpit data, it is added. Then a new configuration

named **Imported Results <n>** (where <n> is an integer selected to ensure uniqueness) is created. As shown in [Figure 4-25](#), the new configuration is named *Imported Results 1*.

**Figure 4-25. Imported Results**



This configuration points to the selected *VRM Data* directory and is initialized with the Runnable selections and command-line options extracted from the selected Status Event Log. These changes are also reflected in the project file if a project file is loaded.

Since everything under the *VRM Data* directory row is based on the contents of the *VRM Data* directory, the *History* row should contain entries for every Status Event Log saved in the selected *VRM Data* directory. Subsequent regression runs based on this configuration are launched in the selected *VRM Data* directory.

## Create a New Configuration that Points to an Existing VRM Data Directory

Each configuration points to a single *VRM Data* directory and this directory is where the **VRM Cockpit** searches for the results of previous regression runs. In most cases, when a new configuration is added to the project file, a new (and unique) directory is chosen to serve as the *VRM Data* directory for regression runs launched via that configuration. If the new configuration is configured to use an existing *VRM Data* directory, the *History* of that configuration is read from that *VRM Data* directory, regardless of whether the previous regression runs were launched from the GUI or manually by the user (or by a periodic batch-mode regression script).

## Change Value of the VRM Data Directory After Regression Data is Accumulated

Assume a configuration exists that points to a given *VRM Data* directory. Assume also that several regression runs have been launched from this configuration and the Status Event Logs

are stored in the same *VRM Data* directory. Finally, assume the change of *VRM Data* directory associated with this configuration by doing the following steps:

1. Select the configuration in the **VRM Cockpit**.
2. Use the **Edit Configuration...** context-menu item to open the **Edit VRM Configuration** window.
3. Change the directory path in the *VRM Data* Directory entry box in the **General** tab.
4. Select **OK** to save the change.

At this point, the VRM GUI does not know for sure what the user is trying to do. On the one hand, the user may be intentionally pointing the *VRM Data* directory elsewhere because of the following:

- Made significant changes to either the design or the configuration settings and wants to start over with a blank slate.
- Plans to execute some experimental runs and then change the *VRM Data* directory back to its former value.

On the other hand, the user could intend to simply change the name (or location) of the *VRM Data* directory while keeping the history intact. If the intention is to point at a new *VRM Data* directory and start from scratch, and if there are no other configurations pointing to the former *VRM Data* directory, the history and other collateral files stored in the former *VRM Data* directory will no longer be accessible from the GUI until another configuration is made to point to that directory.

Rather than guess and make the wrong choice in some cases, the VRM GUI displays a dialog explaining the confusion and noting that the former *VRM Data* directory may become inaccessible. The following three choices are offered:

- Use the new *VRM Data* directory setting and leave the former *VRM Data* directory as is (possibly losing GUI access to the contents thereof).
- Rename the former *VRM Data* directory to reflect the new name/location.
- Return to the **Edit VRM Configuration** window (in the event the change was accidental).

If the first option (use the new *VRM Data* directory setting) is selected and no directory exists at the new location, then the user is prompted to create the new directory just as if a new configuration had been created.

## Launch Multiple Regression Runs in a Single VRM Data Directory

The VRM GUI does not enforce a one-regression-run-per-configuration rule, nor does it enforce a one-to-one correspondence between configurations and *VRM Data* directories. Instead, each batch-mode regression establishes an advisory lock on the *VRM Data* directory prior to launching any Actions (this occurs for regression runs launched from the GUI and for regression runs launched manually from outside any GUI session). In the case where a batch-mode regression detects an active lock on the *VRMData* directory, a message is emitted to the output of the *vrun* process and the process waits 30 seconds before launching any Actions to give the user time to abort the process if the overlap was unintentional.

In the case where a regression run launched from within a GUI session detects an active lock on the *VRM Data* directory, a dialog is displayed that lists all the active locks on the directory. In either case, an “active” lock is defined as a lock generated by a batch-mode *vrun* process that is known to be running (the lock file contains the *port@host* of the status port for the *vrun* process and a “ping” is issued to verify whether the associated regression is still running; lock files for which no running *vrun* process can be located are silently deleted).

The user has three choices of how to handle the situation as follows:

- Elect to ignore the lock files and start the new regression run anyway. This is used when two or more configurations point to the same *VRM Data* directory but also contain non-overlapping Runnable selections so concurrent execution cannot result in file collisions.
- Elect to delete selected lock files (and terminate the regression runs associated with those files) and start the new regression run. This is used when a previous regression run had not finished but the user wants to start a new regression run to replace the currently running regression (this could be accomplished by manually terminating the prior regression run but, in some cases, there may not be a **VRM Results** window open for the currently running regression in this GUI session so using the lock file to terminate the currently running regression may end up being a more direct solution).
- Elect to abandon the new regression run and let the currently running regression(s) continue to completion.

The warning dialog lists the contents of all active lock files, including the *port@host* of the *vrun* process’ status socket, the user ID of the user who initiated the run, the pid of the *vrun* process, and the date and time at which the regression run was started. By default, all the active lock files are selected. However, if the user elects to terminate one or more of the currently running regressions, the lock file selection may be changed. In this case, locks that remain selected will be deleted and the associated regression run terminated while locks that the user deselects before dismissing the dialog are ignored and the associated regression runs are allowed to continue unmolested. This gives the user the flexibility to deal with unusual corner cases while still providing a simple 3-way selection for the common case.

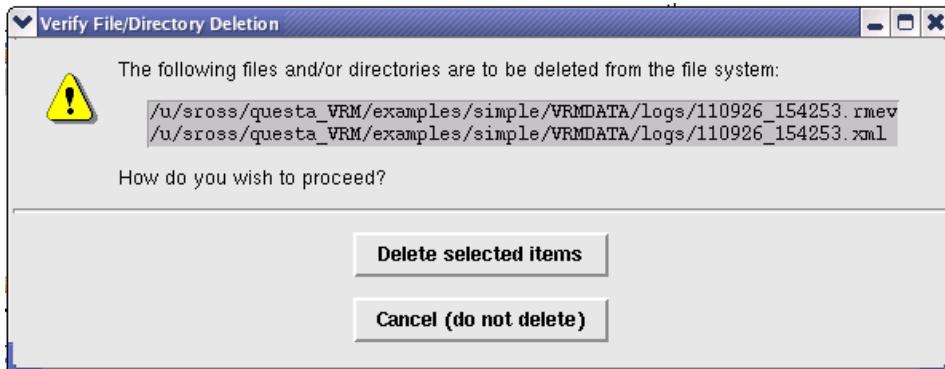
## Delete File and/or Status Event Log from the VRM Data Directory

Status Event Logs and collateral files under a *VRMData* directory can be deleted in the following manner:

1. Select the file(s) and/or directory(s) to be deleted.
2. Use the **Delete Selected Items...** context-menu item to initiate the deletion process.

This invokes the **Verify File/Directory Deletion** window as shown in [Figure 4-26](#).

**Figure 4-26. Verify File/Directory Deletion**

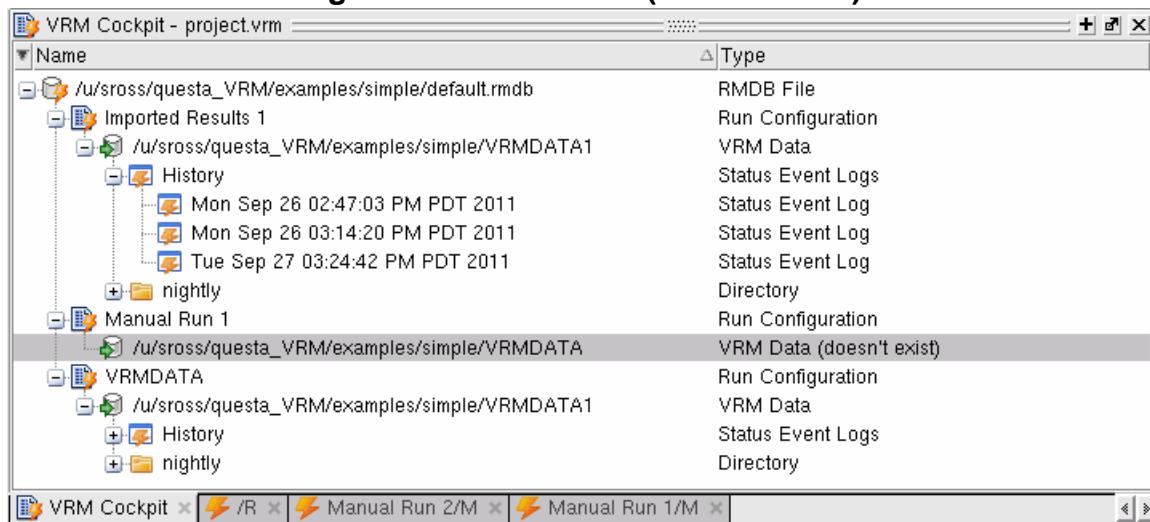


A confirmation is presented listing the set of files and/or directories to be deleted. This is also a selectable list that gives the user the choice of the following:

- Confirm the deletion of the entire selected set of items.
- Abandoning the deletion process (thereby not deleting anything).
- Deselect one or more items and delete only those items that remain selected.

Deletion of a directory results in the deletion of all files and/or subdirectories under that directory. Deleted items are removed from the **VRM Cockpit** display. In the case where the *VRM Data* directory itself is deleted, the *VRM Data* directory row is not removed from the **VRM Cockpit** display, but the *Type* column shows the directory as *VRM Data (doesn't exist)* as shown in [Figure 4-27](#). The next time a regression is executed in that *VRM Data* directory, the directory is recreated.

**Figure 4-27. VRM Data (doesn't exist)**



# Analyze Results and Rerun Tests Based on the Analysis

---

This section explains how current results inform future testing.

<b>Execute the Initial Regression Run .....</b>	<b>203</b>
<b>Analyze the Results and Rerun the Tests .....</b>	<b>203</b>

## Execute the Initial Regression Run

Typically, the user defines a configuration in which all the active tests in the regression suite are selected and the command-line options are chosen for optimum performance. It is likely the run is setup to automatically rerun failed tests with a debug parameter set to gather additional information to help assess the cause of the failure. One such tool is called **Results Analysis**. Assume the RMDB is setup to add the messages from every failed test into the triage database (TDB) file.

Once the RMDB and configuration are defined in a project file and the project file is loaded into the **VRM Cockpit**, the user selects the “run all tests” configuration in the **VRM Cockpit** window and uses the context menu to select **Run** to launch the regression run. A **VRM Results** window opens to show the regression results as the run progresses.

## Analyze the Results and Rerun the Tests

Once the regression has completed, the user selects a failing test in the **VRM Results** window and uses the context menu to select **File-> Open -> Results Analysis Database** to open the TDB file for that regression in the **Verification Results Analysis** window. This window shows all the messages generated by the failing tests. After selecting one or more errors and fixing them in the design, the user selects those error messages in the **Results Analysis** window and uses the **Run Again** context menu item to rerun the tests associated with those messages. This sub-menu opens to show a list of configurations available in the **VRM Cockpit** window.

Typically, the user defines a configuration called “rerun tests” in which the “Clear merge file before auto-merge” and “Clear TDB before auto-triage” check boxes are unchecked. This ensures that the merge file and TDB are not overwritten by the regression rerun. From the **Results Analysis** window “Rerun in Run Manager >” sub-menu, the user clicks the “rerun tests” configuration to rerun the tests associated with the selected messages using the modified command-line options.

The previous steps can be repeated as many times as necessary to clear the regression failures. When a rerun is launched from the **Results Analysis** window, the previous results remain in the **VRM Results** window so it is fairly easy to see when all the failures have been cleared.

## Counter Control in the VRM GUI

You can manipulate the names and contents of the VRM Cockpit columns and the counters in the status bar of the VRM Results window.

Note that the Coverage and Testplan counters will always be displayed, regardless of your customized settings.

<b>Changing the Counter Configuration .....</b>	<b>204</b>
<b>Configuration File for vrmcounter Commands .....</b>	<b>204</b>

## Changing the Counter Configuration

Modify the counter configuration of the VRM Cockpit columns or VRM Results window status bar by using the vrmcounter commands and altering a configuration file.

### Restrictions and Limitations

- It is assumed that you will want these changes to be persistent, so this procedure is written from the point of view of setting up your environment.

### Procedure

1. Open the VRM GUI  
**vrun -gui**
2. Save the current configuration as a template with the vrmcounter save command.  
**vrmcounter save configuration.txt**
3. Use a text editor to modify the saved file. Refer to “[Configuration File for vrmcounter Commands](#)” for a description of the syntax
4. Load the modified text file into the GUI with the vrmcounter load command.  
**vrmcounter load configuration.txt**
5. Close the VRM GUI
6. Re-open the VRM GUI to use the environment as you defined.

### Results

Your configuration is stored in your GUI preferences and all subsequent GUI sessions will reflect this change.

## Configuration File for vrmcounter Commands

The configuration file used with the vrmcounter commands contains mapping information that identifies the counter names and the related status keywords that contribute to the counter.

## vrmcounter Configuration File Syntax

```
<counter_name>: <status_keyword>...
```

- <counter\_name>

A text string that identifies the name to be shown within the GUI. It is followed immediately by a colon (:).

- <status\_keyword>

A space-separated list of keywords that identify the status values to use when totaling up the value to be associated with <counter\_name>.

dropped — The Action was unintentionally dropped by VRM.

eligible — The Action is eligible to be launched but has not yet been launched.

empty — The Action contained no script commands.

failed — The Action has completed but reported a failure.

killed — The Action was killed (by manual intervention).

launched — The Action has been launched but has not yet started.

noexec — The Action was not run because the -noexec option was specified.

passed — The Action has successfully finished.

pending — The Action is waiting for a dependency to clear.

resumed — The Action was suspended but has resumed and is currently running.

running — The Action has started and is currently running.

skipped — The Action has been skipped due to previous errors.

suspended — The Action has been suspended.

timeout — The Action has failed due to a timeout.

## Syntax Rules

- A comment starts from the first pound sign (#) on the line and runs to the end of that line
- The counter names must be unique and cannot contain embedded spaces or colons.
- To define a counter whose title contains a space, use an underscore in the counter name and it will be converted to a space when the name is used as a title in the VRM GUI.
- Whitespace before the counter name, immediately before the colon, or anywhere in the status keyword list is ignored, this includes properly escaped newline characters.
- If you use the string “Total” as your <counter\_name>, the value of all associated status\_keywords will be used to compute the percentage values for the other status keywords, otherwise the sum of all the counted Actions will be used to compute the percentage values.

- <status\_keyword> can be a reference to a <counter\_name> by adding an at symbol (@) in front of the reference, for example:

```
Total: @passed @failed @running @waiting @other
```

- You can alter the color used to display the text by specifying the color, in parentheses, after the <counter\_name> and before the colon. Valid color names are defined by the TCL/Tk toolkit, and some examples are: green4, red3, blue, black, and purple4, which all work well with the current color scheme. For example:

```
Passed (green4): passed  
Failed (red3): failed timeout
```

## vrmcounter Configuration File Examples

- Default Settings:

```
Pending: pending eligible  
Queued: launched  
Running: running resumed  
Suspended: suspended  
Passed: passed  
Failed: failed  
Timeout: timeout  
Killed: killed  
Skipped: skipped  
Dropped: dropped
```

- Customized Settings:

```
Passed: passed  
Failed: failed timeout  
Running: running resumed  
Waiting: pending eligible launched suspended  
Other: killed skipped dropped
```

# Effect of Projects and Configurations on Batch-mode VRM Regressions

---

The batch-mode *vrun* application understands projects and configurations. If the *-project* command-line option points to a valid VRM Project file and the *-config* command-line option matches the name of a configuration inside that project file, then the batch-mode regression process uses the Runnable selections and command-line options specified by that project to run the regression. Any additional command-line options listed on the original *vrun* command line will override command-line options specified in the configuration.

For example, if the *myconfig* configuration (in the project file *myproject.vrm*) specifies the following Runnable selections and command-line options:

```
vrun -j 10 -html -mintimeout  
100 nightly
```

and the batch-mode *vrun* process is invoked with the following command line:

```
vrun -project myproject.vrm  
-config myconfig -j 3 -noautomerge
```

the regression runs as if the original command line had been:

```
vrun -j 3 -noautomerge  
-html -mintimeout 100 nightly
```

Note that only command-line options from the original command line will override the configuration contents. The Runnable selections are always taken from the configuration (there are issues that preclude merging the Runnable selections from both the configuration and the original *vrun* command line).

A useful side effect of this function allows one to query a configuration to see which Runnables are selected without loading the project into a GUI window. Simply add the *-show* command-line option to the command line used to initiate a configuration-driven regression as follows:

```
vrun -project myproject.vrm  
-config myconfig -show
```

The *-project* and *-config* command-line options are only queried in run mode. If any other mode option is specified on the *vrun* command line (such as *-gui*, *-send*, *-kill*, and so on), then the *-project* option is ignored and the *-config* command-line options is ignored unless said option is assumed to convey the path to the RMDB file.

<b>Batch-mode Regression .....</b>	<b>208</b>
<b>Interaction with the Legacy Mode -config Command-line Option .....</b>	<b>208</b>

## Batch-mode Regression

Under VRM, a batch-mode regression run is launched with the *vrun* command. This command generally selects one or more Runnables from an RMDB file and causes them to be executed according to a set of command-line arguments provided by the user. For example, the following command:

```
vrun -rmdb my.rmdb  
-vrldata ./VRMDATA -j 1 nightly
```

cause the *nightly* Runnable from the RMDB file *my.rmdb* to be executed, one at a time (*-j 1*) with the results being deposited into the directory *./VRMDATA*. This is known as a batch-mode regression run. In the simplest GUI use model, the user simply adds the *-gui* option to the *vrun* command line as follows:

```
vrun -rmdb my.rmdb -vrldata  
./VRMDATA -j 1 nightly -gui
```

This invokes the VRM GUI which captures the remaining command-line arguments and uses them to launch a batch-mode *vrun* process to run the actual regression. The reason that the regression is managed by a separate batch-mode process is that it must be possible for the user to end the GUI session without terminating ongoing (and possibly long-running) regression runs. Moreover, a single GUI can manage multiple batch-mode regressions, even if those regressions were launched from outside the GUI or in a previous GUI session. In other words, if a given regression run is expected to run overnight, the GUI session can be closed when the user leaves for the day and re-opened the following day to verify the results without killing the running regressions in the meantime. In order to initiate a VRM GUI session in idle mode (that is, without also launching a regression run), the following command can be used:

```
vrun -gui [-project <project-file>]
```

The *-project* command-line option directs the GUI session to load the indicated project file. If a project file is not specified, then a dialog is opened to prompt the user for a project file.

## Interaction with the Legacy Mode *-config* Command-line Option

Prior to the Questa Verification Run Manager 10.1 release, the command-line option used to specify the RMDB file was *-config* as opposed to *-rmdb*. For back-compatibility purposes, the *-config* command-line option can still be used to specify the RMDB file but only if the *-project* command-line option is not present on the command line. In that case, *vrun* assumes the *-config* was meant to specify the RMDB file but a deprecation warning will appear in the *vrun* output. If the *-config* command-line option is used in conjunction with the *-project* command-line option, the *vrun* process assumes the value of the *-config* option was meant to specify the name of a particular configuration in the specified project file.

## Create RMDB File from Existing UCDB File

The following is not a VRM function but can be useful for users just starting out with VRM. A VRM database can be generated from one or more existing UCDB, Merge, or Rank files. This database generation function is only available from the Questa GUI.

In the context menu of the **Verification Browser**, there is a **Generate RMDB File** menu item. Under this menu item is a sub-menu with the following two selections:

**Table 4-2. Generate RMDB File Menu Picks**

Menu Item	Function
Save All Tests	Create a VRM database from all UCDB/Merge/Rank files loaded in the Browser.
Save Selected Tests	Create a VRM database from the set of UCDB/MergeRank files currently selected in the Browser.

All tests included in the generated RMDB database are assumed to belong to a single group named *all*. The algorithm uses the *TESTNAME* and *VSIMARGS* from each test to generate the database. It also sets the *ucdbfile* parameter for each test to point to the original UCDB file for that test (as stored in *ORIGFILENAME*).

Action Scripts .....	209
Compilation and Post-processing .....	211

## Action Scripts

In order for the tests to run as is under VRM, a *cd* command (to the directory stored in *RUNCWD* for that test) is added to each Action script in order to run each simulation in the directory in which it was initially launched. In order to prevent file collisions between concurrent simulations, the sequential flag on the *all* Group is set to “yes”, meaning VRM will run each simulation one-by-one rather than concurrently.

In order to enable concurrency, the following steps should be taken depending on if the simulations are run in the same directory or in separate working directories.

Same Directory:

If the simulations are to be run in the same directory, the user’s script must be changed to give explicit non-overlapping names to the various files generated by the simulation (such as WLF files, UCDB files, transcript or log files, and so on). Then the *sequential="yes"* attribute can be removed from the “*all*” *runnable* element to enable concurrent execution of all the tests.

### Separate Working Directories:

If the simulations are to be run in separate working directories, the *cd* command must be removed and the relative file paths in the generated *vsim* command must be changed to allow the simulation to locate the necessary files. The path to the original working directory is saved in the RMDB database file as the parameter *runcwd* for convenience. All relative file names used on the *vsim* command or in any *do* script should be prefaced with a parameter reference to either *VRUNDIR* (that is, the directory from which *vrun* was launched) or *RUNCWD* (the directory in which the test was originally launched). In addition, a *-lib* option should be added to the *vsim* command line to locate the work library.

For example, if the following *execScript* Action script is auto-generated:

```
<execScript>
    <command>cd (%runcwd%)</command>
    <command>vsim -do {set test test2; do runtest.do} -c -sv_seed 2
        -vopt top</command>
</execScript>
```

The modified script should look something like the following:

```
<execScript>
    <command>vsim -lib (%runcwd%)/work -do {set test test2; do
        (%runcwd%)/runtest.do} -c -sv_seed 2 -vopt top</command>
</execScript>
```

This command points to the common work library in the original simulation directory and to the *do* script in that same directory. Because the simulation runs in its own working directory under the scratch directory tree, the *cd* command is no longer needed. Any result checking or coverage merging scripts may have to be modified to look for the WLF/UCDB files generated by the simulation in their new location (under the *VRM Data* directory tree). Finally, the *sequential="yes"* attribute can be removed from the "*all*" *Runnable* element to enable concurrent execution of all the tests.

If the *vsim* command calls out a *do \_dofile\_* command in its *do* command-line option, the contents of the *dofile* may also have to be modified to account for the difference in directories when the *dofile* itself is executed. It may, in fact, be easier to copy the commands from the *dofile* directly into the *execScript* element where they can also make use of parameter references.

If there were other TCL commands executed prior to or following the *vsim* command when the simulation was initially launched (as might be the case if the original *vsim* command had been part of a larger script), the other commands would not have been recorded in the resulting UCDB file. In this case, it may be necessary to add these additional commands to the *execScript* element either before or after the auto-generated *vsim* command.

No attempt is made to parameterize the *execScript* commands. Common paths, options, settings, and other command elements may be pulled out into parameters and these parameters may be moved to the *all* Group Runnable if all the Tasks share the same value for a given parameter. The auto-generated database is intended for demonstration and bootstrap purposes and it is expected that the file would undergo significant change on its way to becoming a complete regression suite.

## Compilation and Post-processing

The UCDB files do not contain information regarding either compilation or post-processing steps. The compilation command(s) can be added to the *preScript* Action script under the *all* Group Runnable. If the *execScript* Action scripts have been modified to support concurrent execution, it might be a good idea to add a *vopt* command to the *preScript* Action. This will cause the design database to be optimized once before any of the simulations are launched. Without this, the first simulation to be launched starts an optimization step and the other simulations (launched at the same time) will complain that the work library is locked. Pre-optimizing avoids these messages.

The user may also wish to add post-processing steps (such as coverage merging) into the *postScript* Action script under the *all* Group Runnable.



# Chapter 5

## VRM Regression Author

---

The Regression Author is a web-based interface that allows you to easily create a simple RMDB file for your VRM-based regressions.

<b>VRM Regression Author Introduction .....</b>	<b>213</b>
<b>Invocation Workflow - Server Only.....</b>	<b>213</b>
<b>Invocation Workflow - Combined Server and Browser.....</b>	<b>214</b>
<b>vrun -author .....</b>	<b>216</b>
<b>VRM Regression Author GUI .....</b>	<b>217</b>

## VRM Regression Author Introduction

You can use the web-based VRM Regression Author interface to create RMDB files for your VRM-based regression runs.

The VRM Regression Author provides you with a correct-by-construction interface for creating RMDB files to control your VRM regression environment.

By filling out the various sections of the interface you are able to turn your existing scripts into a VRM-based RMDB file without having to edit any XML files.

The options available through the interface are applicable to many situations.

### Platform Requirements

Use of the VRM Regression Author requires you to install the following on your machine:

- Python Programming Language — Version 2.6.7.
- Web Browser — Mozilla Firefox version 42, Google Chrome version 47, or Internet Explorer version 11.

## Invocation Workflow - Server Only

Use this task to start a Regression Author server to connect to from a different machine.

### Prerequisites

- Set up your machine to match the [Platform Requirements](#).

## Procedure

1. Navigate to the directory where you want to keep your Regression Author data.
2. Execute the command:

**vrun -author -server**

---

### Note

 The first time you run this in a directory, you will be prompted to create a suites/ directory in the current working directory.

---

This starts the web server for the Regression Author.

3. Take note of the port number created in the output of the command, which appears similar to:

Starting HTTP server on port <port\_number>

4. Open a web browser on any machine, Linux or Windows, and navigate to the address:

<machine\_name>:<port\_number>

## Invocation Workflow - Combined Server and Browser

Use this task to start a Regression Author server that also invokes a web browser on the same machine.

## Prerequisites

- Set up your machine to match the [Platform Requirements](#).

## Procedure

1. Navigate to the directory where you want to keep your Regression Author data.
2. (Optional) Explicitly specify a browser.

Set the BROWSER environment variable to the path of your browser executable.

The Regression Author will then automatically use this browser

3. Execute the command:

**vrun -author**

---

### Note

 The first time you run this in a directory, you will be prompted to create a suites/ directory in the current working directory.

---

This starts the web server for the Regression Author, assigns a port number, and opens your default, or specified (step 2) browser to the web interface.

## vrun -author

Starts the VRM Regression Author environment.

### Usage

```
vrun -author [-authorlog <path>] [-cwd] [-server] [-sport <portname>] [-suites <path>]
```

### Arguments

- **-authorlog [<path>]**

Specify a path to save a log file containing server messages. By default the file is named *author\_<number>.log*, where <number> is a random number assigned to the filename.

The default location is the current working directory.

- **-cwd**

Instructs the command to look for content files in the current working directory. By default, the content files are included in your product install directory.

Useful for users who are writing local templates. If you do create a custom set of content files, the directory structure must match that of the *vm\_src/vauthor*/directory in your installation directory.

- **-server**

Run in server-only mode, which suppresses invocation of a browser.

Useful when you are using a browser on a machine other than from where you invoked the Regression Author server.

- **-sport <port\_name>**

Specify a listening port.

Useful when you want to define a specific port rather than allow the command to select one for you.

- **-suites [<path>]**

Specify a path to the location in which to save the configuration files. The string “global” is restricted, you cannot use this as the suite name.

By default, the suites/ directory is saved in the current working directory

# VRM Regression Author GUI

Introduction to the web interface of the Regression Author.

## Description

The web interface of the Regression Author allows you to define your regression run and create an RMDB file for use with VRM.

## Objects

- Navigation Menu
  - Manage Regressions — Loads a list of existing regression suites in the right-hand pane that allows you to create, edit, or remove suites.
  - Global Settings — Loads a set of data-entry panels in the main window.
  - Regression List — Links to regression suites you have created, where you can update any settings.
- Main Window
  - Regression Suites — Visible when you select Manage Regressions from the Navigation Menu. Use this window to manage your existing regressions, including creating, editing, deleting, or generating your RMDB file.
  - Global Settings — Visible when you select Global Settings from the Navigation menu. Use this window to define global settings that apply to all regression suites.
  - Regression Suite Editing Window — Visible when you select a Regression suite from the navigation menu or from the Regression suite list. This window has four main tabs for defining your regression suite:
    - Regression Settings — General settings for your suites.
    - Compile Settings — Parameter and compile script definitions.
    - Simulation Settings — Parameter and simulation script definitions.
    - Reporting Settings — Parameter definitions and choices for reporting of results.



# Chapter 6

## RMDB Database Topology

---

The RMDB database contains the information required to expand one or more selected Tasks or Groups of tasks, invoked by name on the command line, into one or more individual Actions that are to be executed.

<b>RMDB Database Requirements .....</b>	<b>219</b>
<b>Database Content and Naming Conventions.....</b>	<b>220</b>
<b>Database Structure .....</b>	<b>222</b>
<b>Execution Model .....</b>	<b>225</b>
<b>Actions and Commands.....</b>	<b>227</b>
<b>Repeating a Task or a Group .....</b>	<b>229</b>
<b>Dynamically Repeating Runnables .....</b>	<b>233</b>
<b>Selective Membership in a Group .....</b>	<b>243</b>
<b>Basic Regression Execution Algorithm .....</b>	<b>245</b>
<b>Test-centric Reporting and Control.....</b>	<b>249</b>

## RMDB Database Requirements

The RMDB database design is based on the following requirements:

- It must be able to support both individual Tasks and Groups of tasks.
- It must support nested Group hierarchy to a theoretically infinite depth.
- It must be able to parameterize a single shared Task description.
- It must be able to accommodate existing user scripts easily.
- It must be able to accommodate scripts in various scripting languages (that is, Perl, Ruby, TCL, and so on).
- It must support prerequisite Actions (for example, design compilation) that must be run prior to other Actions.
- The storage format should be human-readable so that users can easily generate, maintain, and debug their regression setup or convert to/from other formats.
- The syntax should not depend on the position or sequence of the data elements, so as to maintain back-compatibility as the data structure changes.

- This database is relatively static (few writes) with a relatively low volume of information.

The term “database” is used in a loose sense here. The current implementation uses a single XML file known as a Run Manager DataBase (RMDB) file. A C-API (callable from TCL) encapsulates the database and provides semantic access functions independent of the physical format used to store the data.

There are multiple ways to create a RMDB database file. The obvious way is with a text editor, editing the XML format directly. The second is via an experimental context-menu option on the UCDB Browser. The **GenerateRMDB File** context menu option generates a rudimentary RMDB file from one or more UCDB files loaded into the **UCDB Browser** (of course, some information such as source code location and compile options are not available from the UCDB files so some manual editing of the resulting database file is to be expected). A prototype Edit GUI is also available as part of the Questa GUI.

Details on the VRM RMDB file contents and the API used to access the data can be found in the Appendix titled “[VRM Database \(RMDB\) API](#)” on page 573.

## Database Content and Naming Conventions

The RMDB database contains user-supplied content. For the most part, this content follows common-sense guidelines. For example, the content of commands provided in the database must conform to the syntax of the script language in which it is written. This section outlines the exceptions to the obvious guidelines.

All content in the database must conform to XML syntax rules. This basically means that the meta-characters used for XML markup cannot be used in user content. These include the angle brackets (< >) and the ampersand (&). In addition, quote characters (" or ') matching the quotes around an attribute value cannot be used within the attribute value itself. In place of these characters, the user must use the XML internal entity associated with the character in question, as shown in [Table 6-1](#). Note that this limitation only applies when the XML file is edited directly. The database API (and, presumably, any GUI database edit tool based on this API) does the conversion automatically.

**Table 6-1. XML Internal Entities for Meta-characters**

Meta-character	Entity
&	&amp;
<	&lt;
>	&gt;
'	&apos;
"	&quot;

Identifiers (including names of Runnables, methods, parameters, and so on) must be single-token strings consisting of some combination of alphanumeric characters, underscore (\_) characters, and hyphens (-). The initial character of an identifier must be alphabetic. All identifiers are case sensitive. There are other characters that are acceptable within identifiers but VRM reserves these for future use (and to avoid issues with internal algorithms). For example, since the name of a repeat-expanded Runnable is made up of the original name of the Runnable and a loop index, separated by a tilde (~) character, as in *test~1*, the tilde character should not be used in a Runnable name.

Text content (including commands, parameter values, user TCL code fragments, and so on) can contain any combination of characters, save only that any XML meta-characters must be replaced with the appropriate internal entity.

Whitespace is ignored between XML elements. Whitespace within an element is significant if the semantics of that element permit text content (that is, commands, parameter values, and so on). However, within an Action script or execution method *command* element, any newline characters are converted to spaces and runs of multiple whitespace characters are compressed to a single space each. This allows the user to break a command element in the XML file into multiple lines for readability even though each command will still be presented to the selected command interpreter as a single command. Newlines within the TCL content of a *usertcl* element are passed to the interpreter intact.

The names of the predefined parameters (see [page 266](#)) provided to the parameter expansion algorithm are all upper-case. This is to allow the user to easily differentiate between parameter values supplied by VRM and those which have been defined by the user in the database. In order for this differentiation to be effective, it is recommended that user-defined parameters be given lower-case names. However, this is up to the user and is not a requirement of VRM.

All keywords, attribute values, and element content stored in the XML database are stored in string format. The VRM API assumes all strings read from or written to the XML database are encoded using UTF-8. For strings whose content is limited to the standard ASCII set (the 128 characters whose 8th bit, or MSB, is zero), no conversion is needed. ASCII is a proper subset of UTF-8. If strings containing characters outside of the standard ASCII set must be stored in the database, they should be converted to UTF-8 before being passed to the VRM API. However, since the VRM interfaces with other programs and libraries (such as shell programs, the TCL interpreter, and so on), no guarantee is made as to whether the entire range of UTF-8 encodable strings is supported. Much of that depends on the underlying operating system, and the various libraries and executables in the user's environment.

XML syntax requires that the values of element attributes be quoted (in contrast to the more lenient HTML rules that allowed the quotes to be omitted in many cases).

The XML parser used in VRM conforms to the XML 1.0 specification. For more information on the XML syntax in general, refer to the XML specification on the World Wide Web Consortium website.

## Database Structure

---

One of the requirements of the XML format is that there be a single “root” XML element that contains the entire contents of the XML file. In the case of an RMDB file, that element is the *rmdb* element that essentially defines and contains the entire database. This *rmdb* element is also called the “document element” in XML terminology, since that single element and its contents form the entire definition of the “document” represented by the XML file (the XML format was originally designed to represent document markup, similar to HTML).

The top-level *rmdb* element supports a *toprunnables* attribute. This attribute can contain a list of one or more Runnables that will run by default if no Runnables are named on the command line.

The immediate children of the *rmdb* element are called top-level elements. There are three kinds of top-level elements that can be defined within the *rmdb* element: *Runnable*, *method*, and *usertcl*.

The *Runnable* element corresponds to an instance of a conceptual object referred to as a Runnable, which is a generic term used to refer to the nodes (that is, Groups and Tasks) that make up the hierarchical regression suite tree. A *Runnable* element always has a name and a type and must be defined as a top-level element under the document (*rmdb*) node. See [Runnables and the Regression Suite Hierarchy](#) below for details with Runnable objects and the *Runnable* elements that define them.

The *method* element defines an execution method used to launch Action scripts. A *method* element can have an optional name and can be defined either as a top-level element under the document (*rmdb*) element or as a direct child of the *Runnable* element to which it applies. See [“Execution Method Elements”](#) on page 224 for additional information.

The *usertcl* element defines a fragment of user-specified TCL code that is to be loaded into VRM under certain conditions. A *usertcl* element always has a name and must be defined as a top-level element under the document (*rmdb*) node. See [“User-defined \(usertcl\) TCL Code Fragments”](#) on page 224 for additional information.

The namespaces for *Runnable*, *method*, and *usertcl* elements are independent. In other words, it is not an error for a *method* to exist with the same name as a *Runnable* in the same RMDB file. In every case where an element is referred to by name, the type of element required is already well-known. Therefore, there is no need for the different element types to share a common namespace.

<b>Runnables and the Regression Suite Hierarchy</b> .....	<b>223</b>
<b>Execution Method Elements</b> .....	<b>224</b>
<b>User-defined (usertcl) TCL Code Fragments</b> .....	<b>224</b>

## Runnables and the Regression Suite Hierarchy

VRM views the entire regression suite as a tree whose leaf-level nodes represent individual Tasks to be executed. Non-leaf nodes, known as Groups, can contain zero or more leaf-level Tasks or other Groups as members. The depth to which Groups can be nested is unlimited. This hierarchical relationship is a virtual one, reflected only in the membership lists of the Group nodes. In the database itself, the Group and Task definitions are stored as a flat list of named and typed *Runnable* elements that share a single namespace.

Any given Group or Task can be a member of any number of Groups at the same time. This can be compared to the way hierarchy is implemented in Verilog (or VHDL). Verilog *module* constructs are stored as a flat list in the HDL source and each one can be instanced in one or more other modules. Each instance of a given module is unique, even though the definition of what that module represents is common to all instances of that module (save for some limited parameterization capability).

Likewise, in the VRM database, each Group “instances” one or more Tasks (or other Groups) by name. The definition of each Runnable item in the hierarchy is stored as a top-level *Runnable* element in the database. As in the Verilog language, these Runnable items are highly configurable. Each time a given Runnable item is instanced, it can draw context data from the instancing Group which will, in turn, affect the context of any Groups, Tasks, or Action scripts instanced by that Runnable. Because of this, it is often important to understand the “parentage” of a given Group or Task with reference to a given context. As the list of members of each Group is iterated, a history is maintained as to how each member Runnable came to be invoked (that is, instanced). This “calling chain” is used as a search path for Action scripts and parameter values. It is also used to report the status of any Action scripts. For example, if a Group called *nightly* contained a member Group called *random* which, in turn, contained a member Task called *test1* that contained an *execScript* Action, that Action is referred to as *nightly/random/test1/execScript* in any log messages relevant to that Action.

Groups can be sequential or non-sequential. A sequential Group executes its members in order, launching each as the previous member completes. A non-sequential Group (the default if not otherwise specified) executes its members in parallel whenever possible. The difference is that the Actions in sequential groups execute in a known order whereas the Actions in non-sequential groups can be executed in any order. A Group is considered as “sequential” if the *sequential* attribute of the *Runnable* element corresponding to the Group in question has a “yes” value. Any other value, or if such an attribute is not attached to the *Runnable* element at all, results in the Group being treated as a non-sequential Group.

Runnable (Task and Group) names can consist of any number of alphanumeric characters, underscore (\_) characters, or hyphens (-); except a Runnable name must start with an alphabetic character or an underscore (not a digit). All other punctuation is specifically disallowed. These names can be used as arguments on TCL and shell command lines, in addition to any scripting language the user may be using for external scripts (Perl, Ruby, and so on) so the lexical repertoire is intentionally constrained.

## Execution Method Elements

The *method* element is used to define an execution method for Action scripts. An execution method is essentially a single shell command that is used to launch the job corresponding to a given Action script.

By default, each Action script is launched as a background job on the local machine. Under the control of a *method* element, this same Action script can be launched on a remote server or queued for execution on a server grid.

Multiple conditional *method* elements can be defined under any *runnable* element. As with most elements supporting inheritance, named *method* elements at the top-level of the database (children of the document (*rmdb*) element) can be used as base methods.

As each Action becomes ready to launch, a search is made for a *method* element matching the current conditions and, if found, the command defined under that element is used to launch the script (see “[Execution Methods](#)” on page 315 for details).

## User-defined (*usertcl*) TCL Code Fragments

The *usertcl* element is used to define custom implementations for one or more of the user-definable procedures.

Refer to “[Catalog of User-Definable and Utility Procedures](#)” on page 499 for a complete list of user-definable procedures.

These procedures are called by the VRM at specific points during the execution process to implement behaviors such as 3rd-party notification of the completion of an Action script, pass/fail processing, deletion of the *VRMDATA* directory upon completion, and other granular chunks of behavior that the user may need to customize in order to fit VRM into their existing design flow. These behaviors are encapsulated into TCL procedures and VRM supplies a default implementation for each such procedure. The way these behaviors are customized is to load one or more TCL procedure definitions into the interpreter that essentially “replace” the existing default implementations.

The *usertcl* element can also provide definitions for TCL procedures that can be called from any of the TCL-capable attributes (such as the *foreach* attribute on the *runnable* element), from the value expression of a TCL-type parameter definition (see “[Including TCL Code in a Parameter Definition](#)” on page 267 for details) or can be used to more tightly bind VRM into the user's existing regression flow. In other words, anything that can be done with TCL code from within VRM can also be coded into a *usertcl* element.

A *usertcl* element contains only TCL code. When a *usertcl* element is “loaded” (see “[Loading User-supplied TCL Code Using usertcl into VRM](#)” on page 415 for details), the entire text content of the element is evaluated in a predefined namespace in VRM context. Any procedures defined therein are then available to be called as commands and any TCL commands outside of

a procedure definition are executed at the time the *usertcl* element is loaded. Consider, for example, the following RMDB database fragments:

```
<rmdb>
  <usertcl name="mystuff">
    proc ActionCompleted {userdata} {
      upvar $userdata data
      echo "Action $data(ACTION) has finished"
    }
    echo "Usertcl element 'mystuff' loaded..."
  </usertcl>
</rmdb>
```

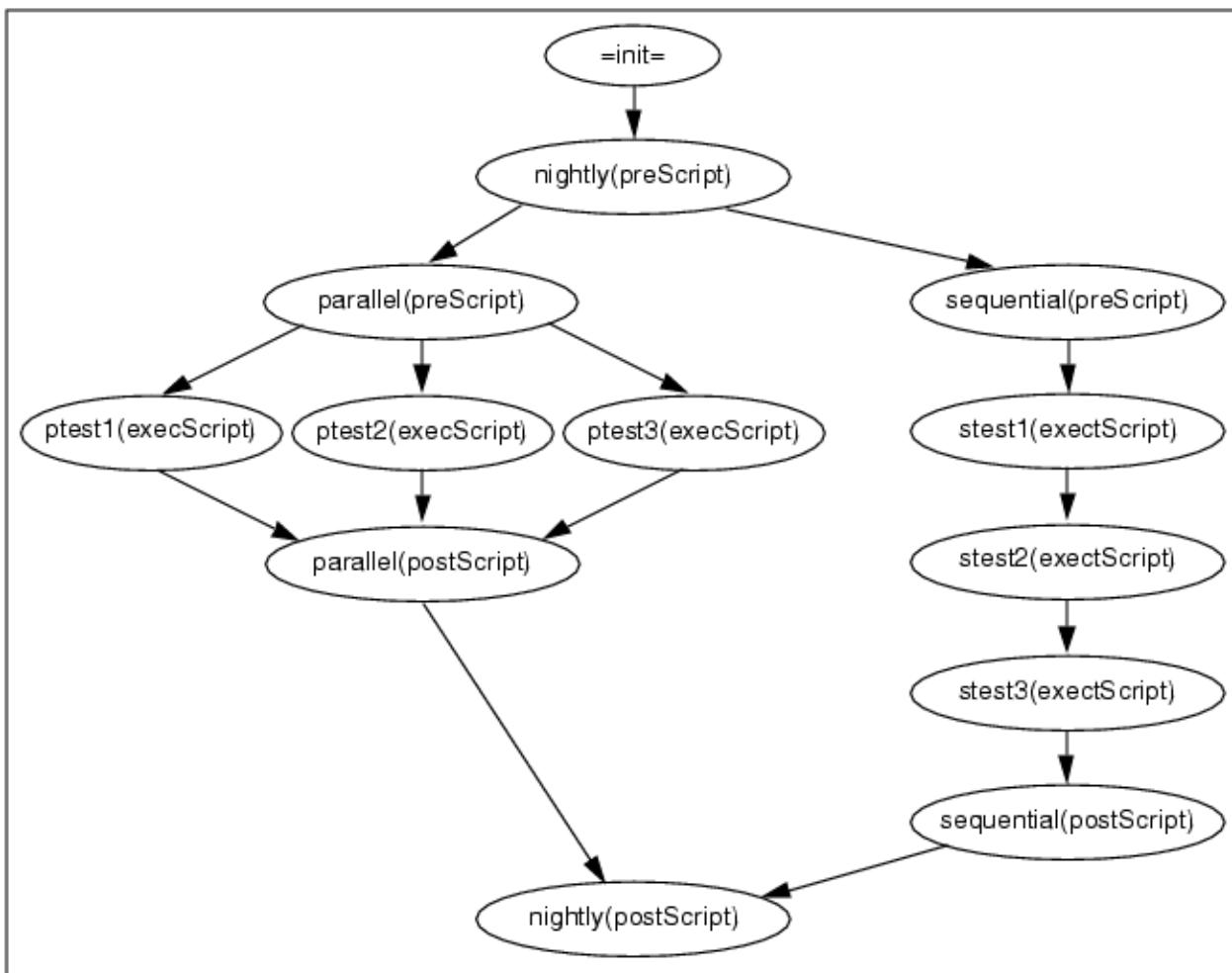
In this example, once the “mystuff” *usertcl* element is loaded, a completion message is emitted to the log output every time an Action script completes. In addition, a “loaded” message is emitted to the log output at the time the *usertcl* element is loaded (the *echo* command is defined by the VRM TCL code to emit the string argument to the transcript output).

All *usertcl* elements are defined at the top-level of the database, under the *rmdb* element. They are identified by a *name* attribute and loaded either automatically or from a command-line option.

## Execution Model

This section illustrates the VRM execution model as it relates to sequential and non-sequential Group Runnables.

Figure 6-1. VRM Execution Model



As illustrated in Figure 6-1, each main Group (named *parallel* and *sequential*) consists of three tests. These two Groups, themselves, are members of a larger Group called *nightly*. The *parallel* Group executes its three Tasks in parallel (hence, the name). The *sequential* Group executes its Tasks one-by-one in sequence. Each Group executes a *preScript* Action prior to iterating its member list and a *postScript* Action after the member Tasks are complete. Each leaf-level Task executes an *execScript* action. The top-level *nightly* Group also executes *preScript* and *postScript* Actions before and after the two main Groups execute.

The *=init=* node represents a pseudo-Action upon which all top-level Group Actions depend. A “done” event on this pseudo-Action queues the initial Actions to run. Thereafter, each time an Action is completed, it serves as a trigger to queue subsequent Actions that depend upon the Action just completed.

## Actions and Commands

There are three kinds of Actions defined in the database: *preScript*, *execScript*, and *postScript*. A leaf-level Task can define an *execScript* that contains the commands necessary to carry out the Task in question. A Group can define a *preScript* and/or a *postScript*. The *preScript* consists of the commands that are run prior to invoking the Tasks and/or Groups listed as members in the Group in question. The *postScript* consists of the commands that are run after all the Tasks and/or Groups contained in said Group have completed. Each of these Actions is associated with a script defined in the RMDB database. Each script is made up of zero or more commands. If an Action script is not defined for a particular Test or Group, or if the Action script contains no commands, then the Action is skipped and nothing is executed at that stage in the run.

When a given Action is to be executed, the commands for that Action are copied to a script file in the working directory for that Action. By default, the commands in each Action are assumed to be TCL commands.

A *launch* attribute can be added to the element corresponding to any Action to indicate how the commands in the corresponding script are to be launched. If the *launch* attribute is set to *exec*, then the script file is launched from *vish* using the TCL *exec* command.

When you set the launch attribute to *vsim*, or if you do not specify the launch attribute, the action executes the script file from within *vsim* using the **do** command. When you set the launch attribute to *vsim* you can also specify any *vsim* arguments to affect the way in which the *vsim* shell opens, prior to executing the script file. For example:

```
<execScript launch="vsim -lic_vlog -lic_noqueue">
  ...
</execScript>
```

which instructs *vsim* to perform the specified licensing tasks before opening the *vsim* shell.

If the launch attribute is set to any other value, then the setting is used as the path to the “shell” to be launched in order to interpret the script commands. This shell command, if specified, is launched from *vish* using the following TCL command:

```
exec $launch $script
```

Rather than defining commands, a given Action definition can point to a file using the *file* attribute. The *file* attribute contains the absolute or relative path to a file containing the Action script. If the path is not absolute, it is assumed to be relative to the directory from which the database was read. When the *file* attribute is given, the commands that make up the Action are read, line-by-line, from the file in question. If the file does not exist or is not readable, then an error is issued and the Action is abandoned. Each command in the file is treated as if it is a command appearing in the database for that Action and is subject to parameter expansion (see “[Parameters and Parameter Expansion](#)” on page 263).

By default, the commands in the file are assumed to be TCL commands. However, the *launch* attribute can be used to control the launching of the Action script in the same manner as for commands defined in the RMDB database.

The *preScript* and *postScript* Actions for any given Group must be defined within that Group (if not, the Actions are skipped). When resolving the *execScript* for a leaf-level Task, however, the entire calling chain is consulted in reverse order until an *execScript* is found (see “[Inheritance, Overrides, and Search Order](#)” on page 271 for details). That allows the user to define a common *execScript* for all the Tasks in a Group by defining the *execScript* Action in the Group itself. When VRM tries to resolve the *execScript* for a given Task and none is found, it checks each calling (parent) Group on up the calling chain looking for a common *execScript* definition. If none is found, then the Action is skipped and a warning message is generated.

Note that if the Runnable in question is a Task and no *execScript* element can be found for that Task, then VRM issues a warning. However, if the Runnable is a Group and either the *preScript* or *postScript* is undefined, then no message is emitted. The theory is that while Groups can be used for simple grouping (in which case, there cannot be any *preScript* or *postScript* Actions to be performed), a Task without an *execScript* is powerless and thus, less likely to be the result of an intentional omission.

## Repeating a Task or a Group

In some cases, such as when constrained random techniques are used to generate stimulus for the device under test (DUT), it can be useful to be able to repeat a single Task (that is, simulation) or Group of Tasks multiple times as part of the same regression run. In the RMDB database itself, this can be as simple as adding a *repeat* attribute to the Runnable representing the Task or Group that is to be executed multiple times. The value of this attribute determines the number of times the Task or Group repeats. If the value of the *repeat* attribute is non-numeric, or if it is set to a number less than one, the effective value is assumed to be one and the Runnable will not repeat.

The repeat-count expansion occurs when the list of selected Runnables is constructed. Actions resulting from the repeating Task or Group are duplicated and added to the graph multiple times, according to the integer number specified in the *repeat* attribute for that Task or Group. The dependencies of each of these Actions is the same as if the name of the Runnable from which they are derived is listed multiple times in the membership list of the calling parent Runnable (or specified multiple times on the command line).

If the parent is a non-sequential Group, then the repeating Runnables can be scheduled to execute in parallel. If the parent is a sequential Group, then the repeating Runnables is scheduled to execute in sequence, each repetition waiting for the previous repetition to complete (without failure) before executing. Each instance of the repeating Task or Group is also executed in its own working directory, just as if it is listed separately in the membership list of its calling parent Runnable.

The name used for these duplicate graph nodes is generated by joining the value of the *name* attribute of the repeating *runnable* element with the one-based index of the repeat count (that is, from one to the number specified in the *repeat* attribute). This index is separated from the name of the runnable element by a tilde (~) character. Although use of the tilde (~) character in the name of a Runnable (in the *name* attribute of a *runnable* element) is not illegal, it should be avoided in order to avoid confusion with repeating Runnables. The (%INSTANCE%) built-in parameter contains the entire expanded name of the Task or Group under execution (that is, *test~1*) and the (%ITERATION%) built-in parameter contains the repeat index of the current repetition of the Task or Group (that is, the number after the tilde in the expanded name).

Like all element attributes, the *repeat* attribute does not inherit. That is to say that if you add a greater-than-one *repeat* attribute to a *runnable* element of type “group”, that Group will repeat but, within each instance of the repeated Group, the Groups and/or Tasks listed as members of that Group will not repeat (unless one or more of the members also has a greater-than-one *repeat* attribute).

The *repeat* attribute can be parameterized. The value of the *repeat* attribute in any given Runnable can be a parameter reference, as in the following:

```
<runnable name="test" type="task" repeat="(%numtests%)">
  ...
</runnable>
```

The parameter reference can be satisfied by a parameter in a base Runnable or a group ancestor Runnable. An abbreviated set of predefined parameters are also available (see “[Predefined Parameter Usage](#)” on page 266 for details) but, as the predefined parameters are all non-numeric, they may not be useful for parameterizing a numeric value. The parameter(s) referenced by the *repeat* attribute can also include a default value and can be overridden from the command line (using the *-g/-G* options) like any other parameter.

**Controlling the Repeat Index (foreach Functionality)..... 230**

## Controlling the Repeat Index (foreach Functionality)

Unless otherwise specified, the indices of the repeating Runnables run from one (1) to the number given in the *repeat* attribute (N). The list of indices can be controlled by the *foreach* attribute. The *foreach* attribute contains a list of one or more string tokens, separated by whitespace, commas, or a combination of both. A Runnable containing a properly formatted *foreach* attribute (one whose value contains at least one token) repeats once for each token in the value of the *foreach* attribute, irrespective of whether there is a *repeat* attribute specified. The indices used for each instance of the Runnable are derived from the token strings found in the *foreach* attribute.

The *foreach* attribute of any Runnable can contain a token that begins with an asterisk (\*). This signals the Runnable to repeat dynamically at run-time based on the return value of a user-definable procedure.

Note that the tokens in the *foreach* attribute need not be numeric.

For example, assume you have the following Task and Group in the RMDB database:

```
<runnable name="test" type="task" foreach="93 111 156">
  ...
</runnable>
<runnable name="allseeds" type="group">
  <members>
    <member>test</member>
  </members>
  ...
</runnable>
```

When the “allseeds” Group executes, it executes three Tasks, one for each value in the *foreach* attribute of the Task. The Task instances are called *test~93*, *test~111*, and *test~156*.

The *foreach* attribute can be parameterized. The example above could have been modified as follows:

```
<runnable name="allseeds" type="group">
    ...
    <parameters>
        <parameter name="seeds">93 111 156</parameter>
    </parameters>
    <members>
        <member>test1</member>
    </members>
    ...
</runnable>
<runnable name="test1" type="task" foreach="(%seeds%) ">
    ...
</runnable>
```

Parameterization allows the list of tokens to be overridden from the command line at runtime (using the *-g/-G* options). As in the case of the *repeat* attribute, the parameter can contain a default value and can be resolved by a parameter definition located anywhere along the base or group inheritance chains. The same abbreviated set of predefined parameters available for expansion of the *repeat* attribute are also available for the *foreach* expansion (see “[Predefined Parameter Usage](#)” on page 266 for details).

The *foreach* functionality makes no assumption as to what the specified indices represent. The *(%ITERATION%)* predefined parameter is set to the corresponding token string for each instance in turn, just as it is for a Runnable with a *repeat* attribute. However, the primary reason for using *foreach* is to specify a list of random seeds to be used in executing a constrained random test. If this is the case, the index variable can be renamed for convenience. For example,

```
<parameter name="seed">(%ITERATION%)</parameter>
```

The value of the *(%ITERATION%)* predefined parameter for a non-repeating Runnable is always “1”.

The *(%ITERATION%)* predefined parameter applies only to the Runnable on whose behalf the parameter reference is being expanded. In the case of nested repeating Runnables, it is sometimes necessary to reference the instance index of a Runnable further up in the chain. To accommodate these cases, each repeating Runnable defines an additional predefined parameter that is the name of the Runnable with the string *.ITERATION* appended (for example, *testA.ITERATION*). In the example above, this additional parameter is referenced as *(%test1.ITERATION%)*. In addition, the user may define a parameter to contain the iteration index for a given Runnable by attaching the *index* attribute to the *runnable* element for the repeating Runnable in question. The value of the *index* attribute is used as the name of a

parameter whose value is the iteration index for that Runnable. Even when an *index* attribute is specified, the other predefined parameters are still available for use.

Repeating Runnables (that is, those with *repeat* or *foreach* attributes) automatically define a built-in parameter whose value is the current repeat index for that runnable, regardless of where said parameter is referenced.

The reason both whitespace **and** commas are accepted is because spaces look more natural in an XML file but they are annoying when they appear on a shell command-line. For example, one could override, at runtime, the whitespace-separated list of seeds in the example above with a list separated by commas as follows:

```
vrun ... -Gseeds=2,4,6,8
```

The tokens (strings separated by whitespace and/or commas) can contain any characters, except whitespace or commas. As always, XML meta-characters must be converted to the appropriate external entity. The tokens are used in their specified sequence to construct repeating instances of the Runnable to which they are attached. If there are no tokens in the list (that is, if the *foreach* attribute is empty), then the *repeat* attribute is consulted instead. Note that this implies that a Runnable that defines both a non-empty *foreach* list and a greater-than-one *repeat* value will **always** repeat unless both attributes are somehow overridden.

If either the *foreach* or the *repeat* attribute is overridden by the *-g/-G* option, the same rules apply as given in “[Override Parameter Values from Command Line](#)” on page 132. However, in order for these override options to work with the *foreach* attribute, the value of the attribute to be overridden must be specified as a parameter reference.

There is no *foreach* override option equivalent to the *-R* option for *repeat* (which can force a *repeat* value onto a Runnable even if the Runnable did not originally specify a *repeat* attribute). For any given Runnable, if the *-R* option is specified, that takes precedence. Otherwise, if a *foreach* attribute is specified and its value contains at least one valid numeric token, the tokens found therein are used as the repeat indices. Otherwise, the *repeat* count found in the RMDB database (subject to the standard override/default rules) is used. In the absence of any of the foregoing conditions, the Runnable will execute exactly once.

Note that it is also possible to limit execution of a repeating Runnables to some subset of the indices defined in the RMDB database (or to directly specify repeat indices for non-repeating Runnables) by specifying the Runnable with a repeat index on the command line. See “[Specify Selected Runnables](#)” on page 98 for details.

## Dynamically Repeating Runnables

VRM supports repeating an Action (that is, test, and so on) or a group of Actions multiple times from a single *Runnable* definition. There are two mechanisms to accomplish this as follows:

- A simple repeat count, where a Runnable can be configured to run some number of times in succession. The index value of each repeating instance (called the “iteration” of the Runnable and represented by the *ITERATION* built-in parameter) is the value of a simple integer counter (that is, 1, 2, 3, and so on).
- A flexible mechanism whereby the user can specify a set of iteration values within the RMDB file. This mechanism is most commonly used to specify a set of seeds and/or configurations that are applied to multiple instances of that Runnable in succession. This mechanism, if specified, overrides the simple repeat count. Simple numeric repeating Runnables simply continue to repeat numerically.

Runtime-controlled repeat loops provides a simple way for the user to specify a function by which an arbitrary sequence of iteration values can be provided, this list and its ultimate length to be determined as the regression run progresses rather than before it starts. The uses for this functionality need to use the generation of a stream of random seeds, the length of which is determined by one or more run-time criteria (such as the total elapsed time for the regression, the aggregate coverage achieved by the regression suite up to that point, or any other data that may be visible from within the *vrun* application).

Refer to “[Concurrency and Multiple Repeating Runnable Instances](#)” on page 239 for additional information.

<b>Use Model .....</b>	<b>234</b>
<b>Repeating Runnable Example .....</b>	<b>240</b>

## Use Model

For runtime-controlled repeat loops, the iteration values are referred to as *seeds*, since *seeds* is the most common use for this functionality. However, nothing related to this functionality is tied to seed management, specifically; the same functionality can be used to dynamically iterate configurations, run-time options, test names, or any other user-configurable setting that can be reduced to a list of tokens passed to the *foreach* attribute of the *runnable* element with which they are associated.

<b>Activating the Feature .....</b>	<b>234</b>
<b>Controlling the Feature .....</b>	<b>236</b>
<b>Lexical Requirements for foreach Attribute Values and Tokens .....</b>	<b>237</b>
<b>GetNextIteration Procedure .....</b>	<b>238</b>
<b>Concurrency and Multiple Repeating Runnable Instances.....</b>	<b>239</b>

## Activating the Feature

When a user wants a given Runnable to repeat with specific seeds (as opposed to a simple integer counter), the user attaches a list of the seeds to the *runnable* element via the *foreach* attribute as follows:

```
<runnable name="test" type="task" foreach="123 456 789">
  ...
</runnable>
```

The *foreach* attribute is one of the few RMDB attributes that is parameter-expanded before use. The expansion takes place from the point of view of the pre-iterated Runnable instance. Thus, the seeds may also be encoded in a parameter (where they can be overridden from the command-line if need be) as follows:

```
<runnable name="test" type="task" foreach="(%seeds%)">
  <parameters>
    <parameter name="seeds">123 456 789</parameter>
    ...
  </parameters>
  ...
</runnable>
```

Also, by using a TCL-based parameter, the seeds can be generated by a user-supplied TCL procedure and/or read from a file as follows:

```

<runnable name="test" type="task" foreach="(%seeds%)">
  <parameters>
    <parameter name="seeds" type="tcl">[getSeeds]</parameter>
    ...
  </parameters>
  ...
</runnable>
...
<usertcl>
proc getSeeds {} {return [list 123 456 789]}
</usertcl>
```

Note that TCL-based parameters are not evaluated. They simply undergo variable and command substitution (via the TCL *subst* command). In order to actually call a TCL procedure, the square brackets are required. The return value of the procedure is then substituted in place of the sub-string containing the bracketed command.

The value of the *foreach* parameter is interpreted as a TCL list (that is, a string containing one or more tokens separated by spaces). The *vrun* application considers spaces and commas to both be valid delimiters, so the following *runnable* elements are expanded identically as follows:

```

<runnable name="test" type="task" foreach="123 456 789">...</runnable>
<runnable name="test" type="task" foreach="123,456,789">...</runnable>
<runnable name="test" type="task" foreach="123, 456, 789">...</runnable>
```

In order to enable the dynamic repeat function for a given Runnable (see the note below), the user includes one or more special pseudo-tokens, token strings that begin with an asterisk (\*), in the token list passed to the *foreach* attribute of the *runnable* element in question as follows:

```

<runnable name="test" type="task" foreach="123 456 789 *">
  ...
</runnable>
```

Note that the \* token can be mixed in with other, statically defined, seeds. Moreover, just because the seed list is shown hard-coded into the *runnable* element, the other mechanisms (such as TCL-based parameters) can still be used to pass the token list, including the \* token. For example, the \* token can be appended onto the end of a token list read by a user-supplied TCL procedure and returned to the *foreach* attribute. The user can also resort to mixed paradigms as shown in the example below:

```
<runnable name="test" type="task" foreach="(%seeds%) * ">
```

The important thing to remember is that the value of the *foreach* attribute is interpreted as a string containing space/comma separated tokens, that string can include one or more parameter

references, and each parameter reference can be defined as a plain string, a TCL variable reference, a TCL function call, or any combination thereof. If, after all the manipulation and expansion, the string returned to *vrun* for the *foreach* attribute contains one or more \* tokens, then the dynamic repeat function is activated.

For explanation purposes, the dynamic repeat pseudo-token is referred to as “the \* token”, but any string beginning with an asterisk (such as \*1, \*star, and so on) triggers a dynamic repeat loop. The asterisk, if included in the token string, must appear as the first character of the token. Asterisks are not allowed in *foreach* attribute token strings except to trigger this dynamic repeating mechanism and the \* tokens are never included as part of any real Action context string.

## Controlling the Feature

Once *vrun* detects one or more \* tokens in the *foreach* attribute of a given *Runnable* element, it assumes that the \* token is a stand-in for one or more real seeds that is read and/or generated by a user-supplied TCL code as the regression progresses. In order to fetch these seeds, a user-definable procedure, called *GetNextIteration*, is passed the usual *userdata* array containing data variables relevant to the context in which the Runnable instance in question is executed. The *GetNextIteration* procedure, in response, returns one or more tokens separated by spaces and/or commas, which are used as the iteration tokens for the next repeating iteration. Each token must meet the same criteria applied to tokens supplied directly via the *foreach* attribute (see the next section for details).

For each token supplied by the *GetNextIteration* procedure, a new instance of the Runnable in question is spawned. This instance appears, for all practical purposes, as if the value obtained from the *GetNextIteration* procedure had been present in the *foreach* attribute of the *Runnable* element at the time the regression was invoked. In other words, this provides a way to defer the full expansion of the *Runnable* element in question to run-time instead of start-up time. The resulting Runnable instance, once spawned, has no special characteristics to set it apart from the Runnable instances expanded from any hard-coded tokens listed in the same *foreach* attribute. This implies that Runnable instances generated at run-time will exhibit the same behavior as any other Runnable with respect to execution, pass/fail analysis, failure rerun, logging, or any other downstream processing.

As long as the *GetNextIteration* procedure continues to supply valid tokens, the repeating Runnable runs forever (in theory). In order to persuade the \* enabled Runnable to stop repeating, the *GetNextIteration* procedure should return a blank token (that is, an empty string). Since an iteration value with no characters is not useful for an actual Runnable instance (because of the confusion it would cause downstream), this value makes a good signal for the repeating Runnable to stop repeating. At that point, the execution graph node associated with the \* enabled Runnable will simply be deleted from the execution graph. The Action nodes associated with the \* enabled Runnable serve only as a template for potential run-time generated Action nodes and are never launched as is. These nodes can, however, be reported in the “unrun Runnable instances” report emitted by *vrun* in the event of premature termination

(either due to a bug, as the result of a user interrupt, or due to the user returning a true value from the *StopRunning* user-definable procedure). This is consistent, since these nodes can result in additional launched Actions had the run not terminated prematurely. But the number and names of the Actions not executed as a result of the termination are unknown because the seed values were being generated on an as-needed basis.

To assist in generating the token stream, several data items pertaining to the repeating Runnable's context are provided to the *GetNextIteration* procedure. One of these, the *ITERATION* value, is a copy of the exact \* token used to enable the dynamic repeat mechanism. By using different \* tokens in different places in the regression hierarchy (that is, \*1, \*2, and so on), the RMDB author can signal the *GetNextIteration* procedure as to which token stream should be used to generate the return token. Another data item supplied to the *GetNextIteration* procedure is the attempt count, a 1-based integer that increments with each new request for an iteration token. This integer, known as *ATTEMPT*, can be used to validate a maximum loop count, index an array, compute a mathematical series, and so on.

The iteration tokens returned from the *GetNextIteration* procedure are validated against two criteria: the token must meet the lexical requirements for *foreach* tokens (see the next section), and the context path resulting from the expansion of the repeating Runnables must be unique (that is, the same token cannot be used twice at any given level of the regression hierarchy). If these criteria are not met, the invalid token is discarded (with the appropriate warning message) and the *GetNextIteration* procedure is called again (this process will repeat a maximum of 93 times, at which time *vrun* will assume that an empty string was returned, thus terminating the repeating Runnable's loop). If the *GetNextIteration* procedure is called because of an invalid or unusable token, then the value of the *ATTEMPT* data item will be the same as when the invalid token was generated. Thus, retries will not count against any maximum loop count that can be implemented in the user-supplied procedure.

## Lexical Requirements for *foreach* Attribute Values and Tokens

The value of the *foreach* attribute must conform to the following pattern:

```
(<NMOKEN>) ? (<SPACE-OR-COMMA> <NMOKEN>) *
```

where *SPACE-OR-COMMA* is any combination of whitespace and/or a single comma, and *NMOKEN* conforms to the following pattern:

```
('*'?)? (<NameChar>) *
```

where *NameChar* conforms to the following pattern:

```
<Letter> | <Digit> | '.' | '-' | '_' | ':'
```

where *Letter* implies a character in the range A-to-Z or a-to-z, and *Digit* implies a character in the range 0-to-9. Tokens not conforming to this specification are ignored and a warning to that effect is generated by *vrun* (either at start-up time if the token is included in the actual *foreach* attribute, directly or indirectly, or at run-time if the token was generated by the *GetNextIteration* procedure).

Note that similar lexical requirements were always assumed but never enforced by the *vrun* application. Since *foreach* tokens are inserted directly into the context paths of launched Actions, and since mysterious behavior and strange errors can result from allowing invalid tokens to be used for real context paths, *vrun* validates every *foreach* token prior to using it (this includes both tokens supplied directly to the *foreach* attribute in the RMDB file and tokens generated at run-time by the *GetNextIteration* procedure).

## GetNextIteration Procedure

The *GetNextIteration* procedure, like all other user-definable procedures, is passed a reference to a TCL array containing various context-specific data items

Refer to the section “[GetNextIteration](#)” on page 523 for additional information.

To override this procedure, the user writes a TCL procedure, generally following the example template shown below, and include it either in an auto-loaded *usertcl* element or in an external file loaded with the *-loadtcl* command-line option. This procedure performs whatever computation is necessary to determine: (a) whether or not the Runnable in question should stop repeating at this time, and (b) the next iteration value in the event the Runnable continues to repeat. The procedure then returns the result of that computation. The procedure follows the usual user-defined procedure override template as follows:

```
proc GetNextIteration {userdata} {
    upvar $userdata data
    ... (some computation resulting in a value for nextIteration) ...
    return $nextIteration
}
```

By default, this procedure returns a series of incrementing integer tokens from 1 up to a maximum loop count determined by expanding the *repcount* parameter from the point of view of the repeating Runnable. This makes it easy to run “N” simulations without supplying a *GetNextIteration* procedure. Once the maximum number of tokens have been generated, or if the *repcount* parameter cannot be expanded, then the procedure returns an empty string.

Each time the *GetNextIteration* procedure is called, it can return one or more iteration tokens separated by any combination of spaces and/or commas. If multiple tokens are returned in any one call to this procedure, then those tokens are used to create that many concurrent Runnable instances.

Since the *GetNextIteration* procedure is called at the time that a \* enabled Action becomes ready-to-run, the context-specific values passed to this procedure will match those passed to the *ActionEligible* procedure.

Note that it is recommended that the loop count not be used directly as the simulation random seed. Nor should the user-supplied *GetNextIteration* procedure attempt to generate a random seed itself, since the TCL random number generator is weak. A better solution is to pass the option *-sv\_seed random* to the simulation and let *vsim* generate the random seed. This seed can then be captured from the UCDB file and stored in a list of “good seeds” if that be the intention.

It is also important that the *GetNextIteration* procedure is implemented such that if it is called twice with the same data item values, it will not return the same token both times. In particular, the *GetNextIteration* procedure should not be implemented to simply return the value of the *ATTEMPT* data item or any combination involving only data item values passed to the procedure. The reason is that in the event of an invalid token value, the *GetNextIteration* procedure is called again with the exact same data item values as when it generated the invalid token. In this case, returning a new generated token reduces the possibility of an infinite loop that cannot be resolved.

## Concurrency and Multiple Repeating Runnable Instances

The primary purpose is to provide a way to run one or more Actions until some arbitrary limit is reached when that limit may not be known until run-time (for example, running tests until a given level of coverage is reached).

For this reason, dynamically repeating Runnables always run sequentially (see “[Dynamically Repeating Runnables](#)” on page 233). That is, a new copy of the \* repeating instance will not be launched until the previous copy has finished. If the parent group of the \* instance has other non-\* members or if the \* token is mixed in with other non-\* tokens in the *foreach* attribute of a given *Runnable* element, the \* enabled Actions is launched either sequentially or concurrently with respect to the non-\* instances, depending on whether the parent group is marked as sequential or non-sequential. However, the Runnable instances generated by a single \* repeating instance will always run sequentially with respect to each other.

If the intention of a given dynamically repeating Runnable is, for instance, the execution of a simulation with some arbitrary number of dynamically generated seeds, and if the environment would support (for example) three simultaneously running instances of this particular simulation, the *foreach* attribute on the *Runnable* element associated with that simulation may contain three \* tokens. When the execution graph is first expanded, three separate nodes (or sub-graphs, in the case of a group Runnable) will be created for these three \* instances. Each of these nodes, when eligible, will cause the first of a sequence of dynamically generated nodes (or sub-graphs) to be created and launched (each running sequentially with respect to each other but all three working in parallel with respect to the regression as a whole). The same *GetNextInstance* procedure is called by all three repeating Runnables to generate the sequence of seeds but as long as the *GetNextInstance* procedure continues to return valid seeds, there

should always be three concurrent instances of the dynamically repeating Runnable running at any given time.

If multiple \* tokens are used in the *foreach* attribute and multiple dynamic tokens are returned from the *GetNextIteration*, then the result is N times M (that is, the product of the two multiples) repeating Runnables.

## Repeating Runnable Example

This is an example of how this functionality can be used. The assumption is that the design has a number of different configuration and/or mode settings, each of which has several possible settings such that it is not possible to simulate every combination in one overnight regression run. The basic premise of the solution is that the user can simulate some random number of these configurations each evening, as time allows, and that over time a significant number of the possible combinations are covered.

The RMDB file below consists of three nested Group Runnables representing each of three configurable parameters (Opt1, Opt2, and Opt3) with a single Task Runnable at the bottom of the stack to represent the test. Each of the dynamic *foreach* expansions uses a unique token so that the *GetNextIteration* procedure knows for which level it is being asked to generate an iteration token.

For the three Group Runnables, this example has encoded the set of possible configuration values in an array in the *User* namespace (all *usertcl* elements are evaluated in the *User* namespace). For demonstration purposes, each Group defines a *repcount* parameter that limits the number of iterations of each Runnable to a fixed maximum. In its current form, this regression could be implemented by picking “N-of-M” seeds at start-up time using a TCL-based parameter in place of the dynamic *foreach* attribute value. However, the user may wish to continue running simulations until some specific time of day (like just before the developers return in the morning) or the user may prefer to terminate the regression run if a given coverage goal is reached (in order to conserve grid resources). Because the seeds are generated at run-time rather than at start-up time, the direction and duration of the regression run can be changed while the regression is still running.

```

<?xml version="1.0" ?>
<rmdb version="1.1" loadtcl="mytcl">
    <runnable name="nightly" type="group">
        <members>
            <member>level1</member>
        </members>
        <execScript launch="vsim">
            <command>runmgr::rm_message "Options: (%Opt1%) (%Opt2%)
                (%Opt3%), seed (%seed%)</command>
        </execScript>
    </runnable>
    <runnable name="level1" type="group" sequential="yes" foreach="*1"
        index="Opt1">
        <parameters>
            <parameter name="repcount">4</parameter>
        </parameters>
        <members>
            <member>level2</member>
        </members>
    </runnable>
    <runnable name="level2" type="group" sequential="yes" foreach="*2"
        index="Opt2">
        <parameters>
            <parameter name="repcount">2</parameter>
        </parameters>
        <members>
            <member>level3</member>
        </members>
    </runnable>
    <runnable name="level3" type="group" sequential="yes" foreach="*3"
        index="Opt3">
        <parameters>
            <parameter name="repcount">2</parameter>
        </parameters>
        <members>
            <member>task4</member>
        </members>
    </runnable>
    <runnable name="task4" type="task" foreach="*4" index="seed">
        <parameters>
            <parameter name="repcount">4</parameter>
        </parameters>
    </runnable>
    <usertcl name="mytcl">

        # large number of options from which to pick a smaller subset of
        # combinations

        set optionArray(*1) [list Opt1_A Opt1_B Opt1_C Opt1_D Opt1_E Opt1_F
            Opt1_G]
        set optionArray(*2) [list Opt2_A Opt2_B Opt2_C Opt2_D Opt2_E Opt2_F
            Opt2_G]
        set optionArray(*3) [list Opt3_A Opt3_B Opt3_C Opt3_D Opt3_E Opt3_F
            Opt3_G]

        proc GetNextIteration {userdata} {
            upvar $userdata data

```

```
variable optionArray

variable iterationCache
variable iterationValue

set action      $data(ACTION)
set iteration  $data(ITERATION)

# check the loop count and either bail out if you are done...

if {$data(ATTEMPT) > [ExpandRmdbParameters $data(ACTION)
                      (%repcount%)]} {
    return {} ;# done
}

# ...or, compute the next loop token based on the original "foreach"
# token

switch -exact -- $iteration {

    *1 - *2 - *3 {

        # the 1st time we see this repeating instance, all options are
        # available...

        if {! [info exists iterationCache($action)]} {
            set iterationCache($action) $optionArray($iteration)
        }

        # ...and we either pick an option from among the available
        # choices...

        if {[llength $iterationCache($action)] > 0} {
            set pick [expr {int(rand() * [llength
                                         $iterationCache($action)])}]
        }

        # use the newly chosen token as the next iteration value...

        set iterationValue($action) [lindex
                                     $iterationCache($action) $pick]

        # ...and delete the token from the list so it cannot be
        # used again

        set iterationCache($action) [lreplace
                                     $iterationCache($action) $pick $pick]

        # ...or, we run out of options (at which point we might as
        # well quit)

        } else {
            set iterationValue($action) {}
        }
    }

    default {

        # for regression stability, generate sequential "seeds"
    }
}
```

```

        # between 11 and "N"

        if {![$info exists iterationValue($action)]} {
            set iterationValue($action) 10 ;# 1st loop token minus
                one
        } else {
            incr iterationValue($action) ;# sequential seeds --
                could be random
        }
    }

    # return the next loop token (option and/or seed)

    return $iterationValue($action)
}
</usertcl>
</rmdb>

```

Following are the important points to notice:

- The star-based *ITERATION* string is used to determine what kind of token is returned (one of the configuration settings or a random seed).
- The *ACTION* context string is used as an index into the *iterationCache* array and, in the case of the seed variable, the *iterationValue* array. This allows the user to track the state of multiple token streams at once.
- The option value list for each Action is copied and values removed as they are returned. This prevents the *GetNextIteration* procedure from returning the same value more than once for any given Action.
- The value of *ATTEMPT* is used to determine when the token string should be terminated. As mentioned before, it is normal to check some other user-defined criteria to determine when to terminate each loop.
- The above notwithstanding, if all the available values for any given Action are exhausted, then the repeating Runnable's loop for that Action is terminated (by returning an empty string as the next token).

The random seeds returned for the leaf-level task are, in this example, sequential integers starting with 11. This was done in order to make the output predictable. In reality, the user would probably return a random seed from the *GetNextIteration* procedure or use the *-sv\_seedrandom* option on the simulation command line to instruct *vsim* to generate its own random seed. The index parameter used to hold the seed value is called *seed* so that the value is captured in the Status Event Log and used in subsequent reruns of the same test.

## Selective Membership in a Group

It is possible to prevent the execution of a specified Runnable based on an attribute that can be parameterized. If the *if* or *unless* attributes are specified in a *Runnable* element, that Runnable's

membership in a Group can be disabled based on the value of either attribute. This might be useful if a particular Group contains some Tasks that are only applicable in certain modes or when run under certain conditions.

The *if* and *unless* attributes are evaluated as TCL code when the execution graph is expanded.

The *if* attribute is a positive selector. If the value of an *if* attribute evaluates to true, then the Runnable to which that attribute is attached is added to the execution graph and executed at its proper time in the regression run.

The *unless* attribute is a negative selector. The Runnable to which this attribute is attached is added to the execution graph only if the value of the *unless* attribute evaluates to false.

If both attributes are specified in the same *Runnable* element, then both must “pass” (that is, the *if* attribute must evaluate to true and the *unless* attribute to false) before the associated Runnable is added to the execution graph.

Both the *if* and *unless* attributes can be parameterized. The same abbreviated set of predefined parameters available for expansion of the *repeat* and *foreach* attributes are available for the expansion of the *if* and *unless* attributes (see “[Predefined Parameter Usage](#)” on page 266). It is often useful for one of these attributes to contain a TCL expression that contains parameter references as operands, as in the following example:

```
<runnable name="basicmode" type="group">
  ...
  <parameters>
    <parameter name="mode">basic</parameter>
  </parameters>
  <members>
    <member>test1</member>
  </members>
  ...
</runnable>
<runnable name="test1" type="task" if=""(%mode%) == "
  &quot;basic&quot;">
  ...
</runnable>
```

In this case, the *test1* Runnable is considered for execution because the TCL evaluation of the *if* attribute passes (the *mode* parameter is replaced by the string *basic* from the parameter definition in the parent Group). Note the use of quote (") to encode the double quote characters, which are not allowed within an XML attribute value. Of course, this example is contrived; if conditional membership is based on a parameter that is constant for each Group, it would be better to simply omit the *test1* Runnable from the membership lists of the Groups to which it does not apply.

## Basic Regression Execution Algorithm

The generic VRM use model is relatively simple. The user defines a regression suite containing an arbitrary number of tests. For convenience, these tests can be collected into groups, where each group can perform some action before and after the group's members are executed. The complete regression suite hierarchy is encoded into a RMDB database. A single call to VRM can then be used to execute all or part of the regression suite under specific conditions. Because VRM is expected to work under various existing regression flows, there are many opportunities for customizing and/or parameterization.

In the overall VRM flow, the user can discern four distinct levels of functionality as follows:

- At the very top-level, is the *vrun* command-line, with all its options, including a recursive “macro” approach that allows a *vrun* command line of arbitrary complexity to be stored inside the RMDB database and invoked via a much shorter *vrun* command. A user shell script that calls the *vrun* utility falls into this top-most level.
- The second level of functionality is the infrastructure of the VRM application. This includes the algorithm which, based on the contents of the *vrun* command line and the RMDB database, decides which Actions should run and when, maintains the *VRMDATA* directory, generates parameter-expanded script files, and launches jobs. This level is the most complex and, fortunately, the least likely to vary from one user environment to the next.
- At the very lowest level are the basic atomic functions. For example, given a UCDB file name, return the pass/fail status from the testdata record; or given a parameter name and a context stack, determine the defined value of the parameter. These things are basic and most users will not benefit from being able to alter them.
- In between the reinforced infrastructure and the atomic functions, there exists a whole range of behaviors that vary from user to user. Functions like: “tell me whether a given test passed or failed” or “should I delete the *VRMDATA* directory for a given test upon completion.” For these functions, a plug-in like architecture is used where the users can write their own code to implement these specific behaviors, overriding the default behavior built into VRM.

The VRM implementation consists of the following major functional elements:

- RMDB database.

The RMDB database describes the hierarchical tree of tasks and groups of tasks. The grouping can be nested to any depth, allowing the user to subdivide the regression suite as necessary. The hierarchy of the tree determines the inter-dependence of the various tasks and the nodes of the tree define the Action scripts that will become the “jobs” executed as part of the regression suite. The content of the database is completely parameterizable and can be configured to invoke the user's existing run scripts in addition to defining run commands from scratch. Refer to “[RMDB Database Topology](#)” on page 219 for detailed information.

- Parameter expansion engine.

The parameter expansion engine supports the parameterization of Action scripts and certain attributes in the RMDB database. Parameter values can also contain nested references to other parameters, making the entire database more flexible. It is common for many Tasks (tests) in a typical regression suite to share the same basic command set with some modifications unique to each Task (such as a *plusarg* argument, a stimulus file name, or a seed value). The parameterization functionality of VRM allows the common parts of the commands to be defined once with Task-specific parameters defined where needed. Refer to “[Parameters and Parameter Expansion](#)” on page 263 for detailed information.

- Working store management.

For any given regression run, each Task or Group of Tasks is assigned a unique working directory and the working directories for the member Groups and/or Tasks within any given Group are created directly under that Group's working directory in a manner reflecting the membership hierarchy defined by the RMDB database. Refer to “[Working Directory and Script Execution](#)” on page 287 for detailed information.

- Task launch algorithm.

The task launch algorithm and job progress monitoring manage the execution of the Actions defined within the regression suite. An execution graph outlining the Actions to be executed and their interrelationship is built from the RMDB database.

- Job progress monitoring.

The task launch algorithm and job progress monitoring manage the execution of the Actions defined within the regression suite. Internal queues are maintained for those Actions that are eligible to run and those that are currently running. VRM tracks and reports the status of the Actions for each run. Refer to [Job Progress Monitoring](#) below.

**Run (and Rerun) Algorithm .....** **247**

## Run (and Rerun) Algorithm

This section describes the Run and Rerun algorithms.

<b>Notification of Regression Start and/or Completion .....</b>	<b>247</b>
<b>Failed Test Rerun.....</b>	<b>247</b>
<b>Job Progress Monitoring.....</b>	<b>247</b>

### Notification of Regression Start and/or Completion

A user-definable procedure, *RegressionStarting*, is called immediately before VRM begins to launch Action scripts. Another user-definable procedure, *RegressionCompleted*, is called after the “done” message for the last Action to be executed has been received (actually, the *RegressionCompleted* procedure is called when the *vrun* application determines, as a result of the processing of the final “done” message, that there are no more Actions eligible to run and none currently running).

The *RegressionStarting* and *RegressionCompleted* procedures return nothing. Also, since *RegressionCompleted* is called after the idle processing loop has exited, there is no penalty for blocking within this procedure. In other words, commands that do not exit for an arbitrarily long time can be called from within the *RegressionCompleted* procedure with impunity (this is definitely not true for the other user-defined procedures).

### Failed Test Rerun

VRM allows a single rerun to be performed on selected Actions. The *-rerun* option is given a new set of options, at least one of which must select one or more Runnables for the rerun phase, and used the options supplied to rerun a subset of the Actions launched on the first pass.

Details of this functionality can be found in “[Failure Rerun Functionality](#)” on page 346.

### Job Progress Monitoring

The VRM algorithm can be divided into two basic functions. The first, somewhat trivial function is to build a list of Actions that need to be executed. This happens once when VRM is first executed. The second, more complex function is to listen for job completions and kick off subsequent rounds of Actions as they become eligible to run. This second function is mainly triggered in response to a “done” message being received from an existing job (exceptions are made for the initial round of eligible Actions that are triggered by a synthetic “done” message on the pseudo-Action *=init=* and for timeouts, which are handled as if a “done” message had occurred but in response to the lapse of a preset time interval).

In order to receive these “done” messages and use them to launch Actions in the proper sequence, the initiating VRM process must remain alive for the duration of each regression run. This process maintains an open TCP/IP socket via which running jobs, possibly executing on

servers other than that on which VRM was invoked, can communicate their “done” message upon completion. Specifically, each job is provided with a server address and the TCP port number opened by the VRM process that kicked off the job. This information is actually written into the wrapper script for the Action. When a job completes, it passes a unique identifier and the *port@host* string to a newly invoked VRM instance (in notification mode) that then connects to the open socket on the initiating VRM to communicate the “done” message. The initiating VRM process will then update its internal data structures and launch subsequent Actions as needed. VRM tracks the yet-to-be-completed jobs (running Actions) and exits when the last of the Actions resulting from the user-specified regression run has been executed.

This communication process is independent of any global status monitor (that is, JobSpy or the monitor software supplied with the grid system). The user can continue to use any global monitor tools with VRM without fear of interfering with its normal operation.

# Test-centric Reporting and Control

This section describes how you can create your regression suite such that you can track actions by test name, rather than action strings.

A RMDB-based regression suite is defined in terms of runnables, which own action scripts defining child processes, or jobs, to be launched by the vrun process during a regression run. There are two basic forms of runnables:

- Task runnables — leaf-level runnables that own action scripts.
- Group runnables — collections of task runnables.

The RMDB infrastructure and vrun command-line allow you to create your regression suite in a way such that you can track actions by test name, rather than action string, which may better suit your current regression environment.

<b>Test-centric Limitations .....</b>	<b>249</b>
<b>Test-centric Reporting.....</b>	<b>250</b>
<b>Defining Test Names in the RMDB File.....</b>	<b>250</b>
<b>Enabling a Test-centric Regression .....</b>	<b>252</b>
<b>Controlling Action Scripts by Test Name .....</b>	<b>252</b>

## Test-centric Limitations

This section describes several restrictions to test-centric regression suites, due to the fact that test-centric control is not native to the VRM workflow.

- You may apply a test name to both task runnables and group runnables.
- You should define the testname parameter in task runnables if the intention is to flag only the leaf-level execScript actions as tests.
- You may only refer to a task runnable or group runnable by a test name when its testname parameter expands to a non-blank string.
- You may only refer to a task runnable or group runnable by a test name when its testname parameter is unique across all selected runnables. If you assign the same testname parameter to multiple runnables, the execScript action associated with those runnables will be treated as if it had no test name.
- When executing a control operation (suspend, resume, or terminate) for a specific test name, the operation applies only to the execScript action of the associated runnable.
- When reporting status by test name, the reports will include only the status of execScript action of the associated runnable.

- For group runnables, any mergeScript or triageScript actions inherit the testname under which it runs and the preScript and postScript actions of a group runnable, which defines a non-blank testname parameter, are considered part of the same test.

## Test-centric Reporting

Status reports generated with the **vrun -status** command will default to the mode, test-centric or action-centric, used for the regression run being reported on.

You can force the report to be in test-centric mode by using the **-testname** option or action-centric mode by using the **-notestname** option.

The test-centric report varies in the following ways:

- Contains only **execScript** actions that define a valid **testname** parameter.
- Replaces the **Action** column with **Test** column, which contains the test name. Note that you can force the appearance of the Action column using the **-columns** option
- The **-testname** option overrides the **-hierarchical** option for HTML reports. This is because there is no hierarchy to test names.
- For HTML output, both the **Action** and **Script** columns are replaced by the **Test** column, where the test name is a link to a page with more information about the action.

### Related Topics

[Regression Run Status Report](#)

## Defining Test Names in the RMDB File

This task describes how you define test names in your RMDB file.

### Restrictions and Limitations

Definition of the **testname** parameter must follow these rules:

- It must be unique. Duplicate names will be treated as if they were not named.
- XML-specific characters (“<”, “>”, and “&”) must be escaped.
- It may contain embedded parameter references. For example

```
<parameter name="testname">random_(%SEED%)</parameter>
```

You can define nested parameter references in a group runnable so that leaf-level task runnables inherit the expanded task names.

## Procedure

1. Determine the test name for a task runnable or group runnable, following the restrictions and limitations previously defined.

Use the following suggestions for determining the test name:

- It is up to you to choose a **testname** parameter for use in the RMDB and to set the **TESTNAME** attribute in your UCDB file. It is suggested that you use the **testname** parameter when setting the **TESTNAME** attribute since that guarantees the names will be the same.
- Avoid characters that may need to be parsed by reporting processes, such as HTML elements or newlines.

2. Create the parameter entry in the RMDB file by adding the **testname** parameter to the **parameters** section of the runnable,

## Examples

This example executes three tests and the names of those tests would expand to **random\_123**, **random\_456**, and **random\_789**.

```
<runnable name="randomTests" type="group">
  ...
  <parameters>
    ...
    <parameter name="testname">random_(%SEED%)</parameter>
    ...
  </parameters>
  ...
  <members>
    ...
    <member>randomTest</member>
    ...
  </members>
  ...
</runnable>
<runnable name="randomTest" type="task" foreach="123, 456, 789">
  ...
  <parameters>
    ...
    <parameter name="SEED">(%ITERATION%)</parameter>
    ...
  </parameters>
  ...
</runnable>
```

## Related Topics

[Test-centric Limitations](#)

[Enabling a Test-centric Regression](#)

## Enabling a Test-centric Regression

By default, the **vrun** command analyzes the specified RMDB file and will enable the action-centric functionality. However, you can manually force the command into test-centric mode by following this procedure.

### Procedure

1. [Defining Test Names in the RMDB File](#)
2. Add the **-testname** option to either A) the **vrun** command line or B) the **options** attribute of the **rmdb** element of your RMDB file.

### Results

As the regression progresses, **vrun** emits messages to the standard output that contain information about the action context string of the script to which they pertain. However, if you specify **-testname**, messages pertaining specifically to **execScript** actions will contain the test name instead of the action context string.

The format of the status event log will include the test name at the end of the eligible status change event. An empty string is used if a test name is not defined, is not unique, or is blank. This information is produced regardless of the use of the **-testname** option.

### Related Topics

[Defining Test Names in the RMDB File](#)

[Test-centric Limitations](#)

## Controlling Action Scripts by Test Name

While your regression suite is running, you can perform three control operations (suspend, resume, or terminate) on an execScript action associated with a task runnable.

### Procedure

1. Begin your regression suite.
2. As needed, use the **vrun -suspend**, **vrun -resume**, or **vrun - kill** commands to affect the desired runnable.

By default, each of these commands will search for the correct runnable whether you specify an action string or testname as the argument, therefore you do not have to specify **-testname** when in test-centric mode. However, you cannot combine test names and action context strings on the same command line.

You can force the control operations to work in test-centric mode by adding the **-testname** option to the command line. For example:

```
vrun -kill -testname myFirstTest
```

Similarly, you can force them to work in action-centric mode by adding the -**notestname** option.

## Examples

```
vrun -suspend [<action> | <testname>] ...
vrun -resume [<action> | <testname>] ...
vrun -kill [<action> | <testname>] ...
```

## Related Topics

[Test-centric Limitations](#)

[Enabling a Test-centric Regression](#)

[Defining Test Names in the RMDB File](#)



# Chapter 7

## Execution Graph and Advanced Topology

---

At the highest level, VRM takes one or more regression suite names given on the command line, expands them (using information in the RMDB database) into a list of Actions that must be executed in some sequence, executes each Action in turn, and then reports the outcome of the entire regression run. The sequence in which the Actions are executed depends on their grouping in the RMDB database. In order to discern the proper execution sequence, the selected Runnables are expanded into Actions using a directed acyclic graph (referred to as the “execution graph”). The nodes of the graph represent Actions (that is, *preScript*, *execScript*, and *postScript*) to be executed. The vertices of the graph encode the dependencies between these Actions (that is, a *postScript* Action depends on the completion of the various *execScript* Actions required for the members of the group to which it belongs).

For each Action executed, VRM manages the working directory, creates any necessary execution scripts, and launches the Actions. It is informed of the completion of each Action and, in the case of *execScript* Actions that represent simulations, performs some preliminary triage (pass/fail analysis). It also handles timeouts and determines (based on an optional heuristic) whether the regression run should be allowed to progress. The sections below discuss the details involved.

A description of the algorithms used to execute and analyze each Action script can be found in the chapters on “[Script Execution and Timeouts](#)” on page 296 and “[Post-execution Analysis](#)” on page 361.

<b>Execution Graph Expansion . . . . .</b>	<b>256</b>
<b>Handling Errors from Action Scripts . . . . .</b>	<b>260</b>

## Execution Graph Expansion

When VRM is invoked, the user specifies one or more Runnable names on the command line. The RMDB database entry for each of these Runnable names is retrieved from the database and one of the following actions takes place:

- In the case of a leaf-level Task entry, an *execScript* Action node is placed into the execution graph.
- In the case of a non-sequential Group, a *preScript* Action node is placed into the execution graph. Then the list of Group members is enumerated and each is handled in turn (recursively). The lead Action of each member (either *preScript* or *execScript*) is marked as depending on the *preScript* of the Group under consideration. Finally, a *postScript* Action node is placed into the graph and marked as depending on the trailing Action (similarly, either an *execScript* or another *postScript*) of each member of the Group.
- In the case of a sequential Group, a *preScript* Action node is placed into the execution graph. As the list of Group members is enumerated, the lead Action of each is marked as depending on the trailing Action of the previous member (the lead Action of the first member is marked as being dependent on the sequential Group's *preScript* node). A *postScript* Action is then placed into the graph and marked as being dependent on the trailing Action of the last member of the Group.

By nesting this handling behavior recursively for each member of any Groups that are encountered, the resulting graph contains a complete list of Actions to be executed and the vertices of the graph are set up to ensure that the *preScript/execScript/postScript* dependencies of both sequential and non-sequential Groups are honored. The leading Action of each Runnable named on the command line is marked as depending on a special predefined node called `=init=`.

For example, suppose your regression run consisted of a single non-sequential Group called *nightly* with three Task members, *test1*, *test2*, and *test3*. A special pseudo-node called `=init=` represents the beginning of the regression run (see [Figure 7-1](#) on page 258). After expansion, a single vertex from `=init=` would lead to the *preScript* Action for the *nightly* Group. From there, three vertices would lead to the *execScript* Actions corresponding to each of the three member Tasks. From each of these *execScript* Actions, a single vertex each (three in all) would lead to the *postScript* of the *nightly* Group. What this implies is that the three *execScript* Actions can only start after the *nightly* Group's *preScript* Action has completed and that the *nightly* Group's *postScript* Action can only start after all three of the *execScript* Actions associated with the member Tasks have completed. The three *execScript* Actions, however, can be run in any sequence and can be concurrently queued to a server farm as parallel jobs (since there is no dependency relationship among the three *execScript* nodes). The *preScript* will most likely contain compile or other preparatory steps necessary to prepare the simulations to be run. The *execScript* Actions will most likely represent three simulations. The final *postScript* Action will probably perform any wrap-up activities (like coverage merging). By further subdividing a

regression suite into nested Groups, one can emulate any reasonable set of dependency requirements.

All of this processing depends on the repeat, foreach, and conditional membership algorithms. As the execution graph is constructed, each Runnable is examined to see if: (a) it can repeat, or (b) it can be disabled.

In the case of a repeating Task or Group (*repeat/foreach*), the handling normally afforded said Task or Group is repeated to satisfy the repeat count or the *foreach* token list, as though the Runnable in question had been included that number of times in the membership list if its parent Group (or on the command-line, in the case of a top-level Runnable). How to specify repeating Runnables is discussed in “[RMDB Database Topology](#)” on page 219.

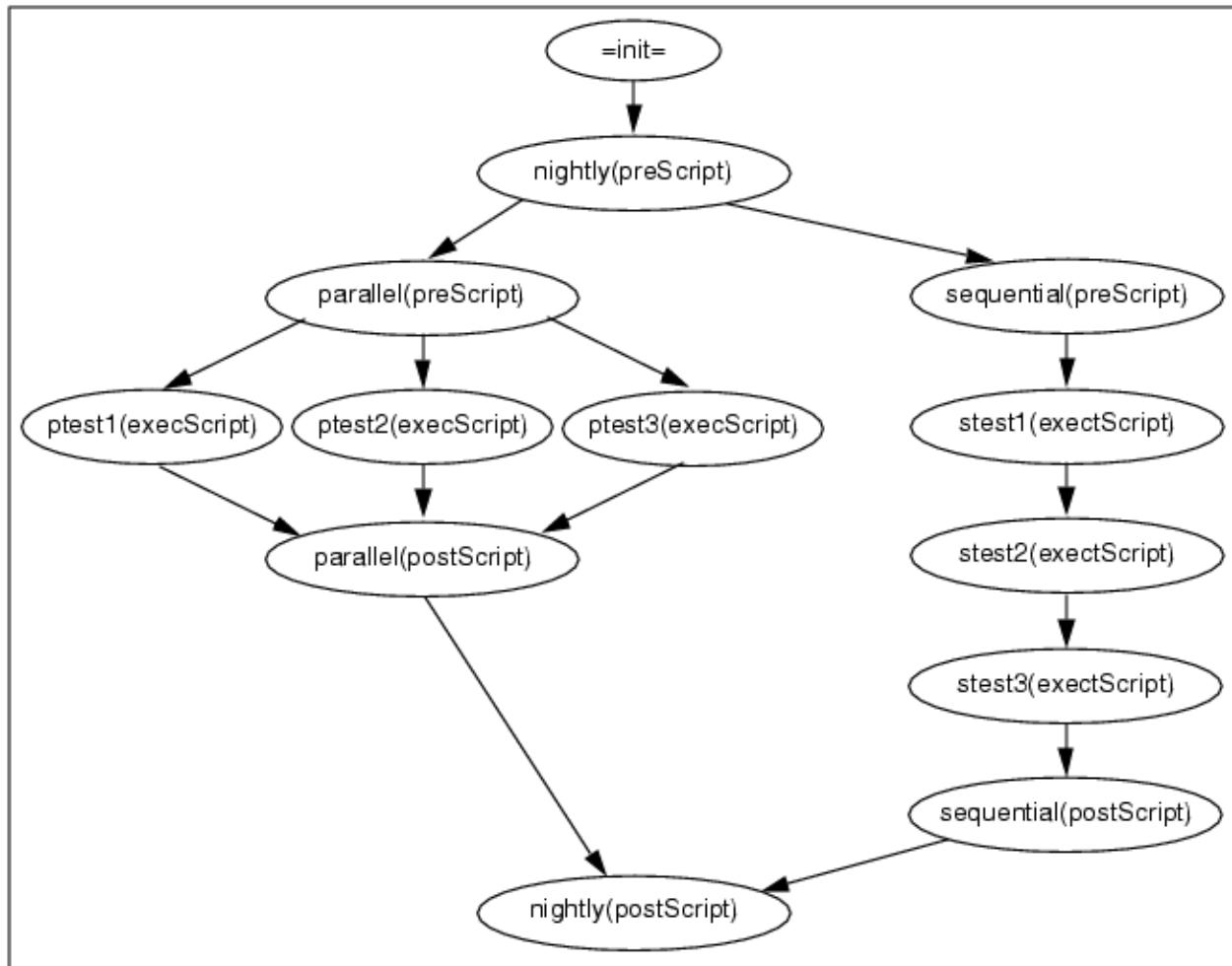
If a Runnable specifies conditional membership (that is, *if/unless*), the Runnable is either processed as above if the conditional test passes, or ignored if the conditional test fails. How to specify conditional Runnables is discussed in “[RMDB Database Topology](#)” on page 219.

<b>Execution Graph Example .....</b>	<b>258</b>
<b>Restrictions on Group Membership.....</b>	<b>259</b>

## Execution Graph Example

This section illustrates how individual Tasks and Groups may be grouped together:

**Figure 7-1. Grouping of Tasks and Groups Diagram**



Each main Group (*parallel* and *sequential*) consists of three tests. These two Groups, themselves, are members of a larger Group called *nightly*. The *parallel* Group executes its three Tasks in parallel (hence, the name). The *sequential* Group executes its Tasks one-by-one in sequence. Each Group executes a *preScript* Action prior to iterating its member list and a *postScript* Action after the member Tasks are complete. Each leaf-level Task executes an *execScript* action. The top-level *nightly* Group also executes *preScript* and *postScript* Actions before and after the two main Groups execute.

The `=init=` node represents a pseudo-Action upon which all top-level Group Actions depend. A “done” event on this pseudo-Action queues the initial Actions to run. Thereafter, each time an

Action is completed, it serves as a trigger to queue subsequent Actions that depend upon the Action just completed.

## Restrictions on Group Membership

In general, VRM assumes that the Runnables listed as members under a given Group are unique (that is, no single Runnable is listed twice in the membership list of any single Group). This restriction is enforced when *vrun* builds the execution graph. The handling depends on whether the Group in question is sequential or non-sequential.

In a non-sequential Group, there is no legitimate reason to include any Runnable in the Group's membership list twice, since the members run concurrently and they will all resolve their parameter references the same way each time. If a duplicate member is detected in a non-sequential Group, then a warning message is reported to the *vrun* log output and the duplicate member(s) are ignored. It is suggested that the duplicate members be removed from the RMDB database.

In a sequential Group, member duplication can have a legitimate purpose, as the conditions under which each instance of the duplicated Runnable executes can be different (as a consequence of the execution of intermediate members in the sequence). However, duplicate members in a sequential Group are not currently supported and their presence in the RMDB database will cause loops in the execution graph which will cause the VRM execution algorithm to exhibit strange behavior. For this reason, duplicate members in a sequential Group will cause the *vrun* application to report a fatal error. The duplication must be removed before the regression can be run.

# Handling Errors from Action Scripts

Some Action nodes are marked for special handling in the event of failure. A failure can be detected by a non-zero return status from the user script for that Action, or it can be the result of the post-execution pass/fail analysis performed after a simulation. In either case, a failure can (depending on the circumstances) cause other Actions in the execution graph to be skipped.

The most obvious case is the failure of a *preScript* Action. Since a *preScript* Action usually contains compilation or other preparatory steps required for the successful execution of the Action scripts associated with the members of the Group with which the *preScript* is associated, failure of a *preScript* Action will cause all “downstream” Actions (that is, all Actions associated with members of the Group or their dependents) to be skipped. The Action's failure is reported to the *vrun* log output and, in verbose reporting mode, the skipped Actions will list the failing *preScript* Action as the cause of their non-execution.

In the case of a sequential Group, it is also assumed that successful completion of each member of the Group is necessary to ensure the successful execution of further members of that same sequential Group. For this reason, a failing member of a sequential Group will cause all Actions associated with (or dependent upon) subsequent members of that same Group to be skipped. As with a *preScript* failure, the failing Action is reported to the *vrun* log output and, in verbose reporting mode, the skipped Actions will list the failing Action that was the cause of their non-execution.

If any member of a sequential Group happens to be another Group (sequential or non-sequential), the “failure” of the member Group is determined by the pass/fail status of the *postScript* Action at the end of that Group. The rationale for this is that a non-sequential Group can have one or more failures among its members and still be able to successfully complete its mission (which might be the merging of the coverage results of its members, including the pass/fail status of each member). In any Group, it is up to the *postScript* Action of that Group to determine whether the failure of one or more of that Group's members constitutes failure of the Group as a whole. In the case where there is no *postScript* Action defined, a sequential Group is considered to have failed if any of its members (or its *preScript*) failed whereas a non-sequential Group will only be considered to have failed if its *preScript* Action failed.

## **Passing Exit Codes From Action Scripts to vrun . . . . . 260**

## Passing Exit Codes From Action Scripts to vrun

You can pass codes to the controlling vrun process of an action script by adding the following Questa command/argument set:

**quit -code <value>**

You can use post-regression run analysis to find instances where an action script ended in a predefined manner.

The controlling vrun process does allow for a built-in reaction to a specific exit code:

**quit -code 93**

When vrun encounters an action script that returns code 93 it will return the script to the execution queue to try the script again without error. The vrun process will attempt this “try again” behavior a maximum of ten times before reporting the script as having failed.



# Chapter 8

## Parameters and Parameter Expansion

---

Each Runnable definition (that is, each *runnable* element) can include a list of zero or more *parameter* element definitions. A parameter is nothing more than a name/value pair that allows multiple Tasks to share the same execution script (or multiple Groups to share the same Task) by providing unique per-Group or per-Task values to be used in the scripts.

Parameters can be attached to any Runnable. In the case of a Group instance, the Parameters defined by the Group are also visible from all child Runnable items invoked as part of that Group (recursively, unless overridden by a more local Parameter with the same name). This allows the user to define name/value pairs that apply globally across the entire regression suite (such as compile options or the location of HDL source files) or across a specific Group of tasks (such as the name and location of the UCDB merge file to be used for a given subset of simulations). For details about parameter visibility see “[Inheritance, Overrides, and Search Order](#)” on page 271.

Parameter references consist of an identifier surrounded by parentheses and percent signs, for example (%*identifier*%). The parameter reference syntax was chosen to reduce the possibility that legitimate syntax in the user's scripting language is not mistakenly interpreted as a parameter reference. The contents of every *command* element, whether it be a method command or part of an Action script, is parameter-expanded prior to use. In addition, if an Action script points to an external file, then each line of that file is parameter-expanded before being copied to the corresponding Action script in the working directory. Each command string can contain zero or more parameter references.

Some element attributes, such as the *repeat* attribute and the *ucdbfile* attribute are also parameter-expanded before use. However, other element attributes are not expanded. See “[Parameterized Element Attributes](#)” on page 495” for a list of expanded attributes).

Parameter “expansion” consists of replacing each of the parameter references found in a given string with a corresponding string value based on a search of the contents of the combined list of defined parameters found in the Runnable itself, its hierarchical ancestors, any base Runnables defined by the Runnable itself or its ancestors, and the command-line options specified by the user when VRM is invoked (not necessarily in that order).

The parameter expansion engine ignores extraneous whitespace within the parameter reference string. For example, “(% fred %)” is the same as “(%fred%). The non-whitespace content of the parameter reference must be a single token, hopefully matching the *name* attribute of a *parameter* defined within the visible scope of the referring Runnable. Embedded whitespace should not be used in the parameter name itself. When a parameter definition is found whose name matches the identifier in the parameter reference, the parameter value is used to replace

the parameter reference. This replacement is a simple lexical substitution operation that is completely ignorant of command syntax.

Parameter expansion is recursive. That is, parameter values (that supply the strings used to expand the parameter references) can, themselves, contain references to other parameters. For example,

```
<parameter name="ucdbfile">../test(% seed %).ucdb</parameter>
```

or the user can embed parameter references within other parameter references in the command itself as follows:

```
<command>This is an (% embedded(% parameter %) %)</command>
```

Expansion continues recursively from the inside out<sup>1</sup> until all the parameter references embedded in a command (or in the parameter values used to replace said references) are resolved. Parameter expansion treats each *command* element in the RMDB database as a separate physical line in the script. Newline characters are converted to whitespace and any extraneous whitespace characters are removed from each command after parameter expansion.

Parameters in the VRM are unlike parameters in Verilog. Verilog parameters are declared and have values assigned. VRM parameters are simply defined. They are key/value string pairs much like the `define compiler directives in Verilog. VRM parameters can be defined in a number of different ways. A *parameter* element within a *runnable* element defines a parameter whose name is given by the *name* attribute of the element. The -g and -G command-line options each define parameter values at one or more places in the database topology. And a specific set of predefined parameters are defined directly within the VRM algorithm itself. When a parameter is defined, it automatically gets a value (even if that value is an empty string, as would occur for a parameter element with a name but no contents, or if the -Gfred= option were given on the command line). An empty string is considered a valid value for a parameter.

Action script and execution method commands, other parameter values, and certain element attributes can contain parameter references. When a parameter reference is expanded, a search is initiated for a parameter definition whose name matches the keyword given in the parameter reference. The parameter reference can also carry a value of its own that is used if no other value can be found for that parameter. Without this fall back value, the only options on finding no definition for the parameter is to fail miserably or swap in an empty string and hope for the best. This fall back value (that is sometimes called a “reference-specific default”) gives the user the opportunity to control what happens in the case where the referenced parameter is not

---

1. “inside out” means that inner (nested) parameter references are resolved first. Therefore, if the name of the parameter to be used to expand a parameter is, itself, the value of some other parameter, then the string “(% (%paramname%) %)” will first resolve the inner parameter reference using the value of the *paramname* parameter, and then will attempt to resolve the outer parameter reference using a parameter of the name determined by the value of the *paramname* parameter. If VRM detects a loop in the parameter reference chain, then it reports an error and the parameter reference resolves to an empty string.

defined. In some cases, as in the case an optional argument being supplied to a command, an empty string is both correct and benign. In other cases, a reference-specific default can prevent a missing parameter definition from turning an otherwise legal command into gibberish.

Note that if a reference-specific default is supplied, then the absence of a parameter definition matching the reference keyword is not an error. The reference-specific default is silently used in place of the parameter reference in the parameter-expanded result. If no reference-specific default is supplied, then an error is reported but the parameter reference is replaced with an empty string and processing continues. If an empty string is the correct result for a particular parameter reference (as when writing a command to test whether a given parameter is or is not defined), then an empty string can be supplied as the reference-specific default.

The reference-specific default value is considered to be part of the reference itself. It says: “for this expansion of this parameter, and no other, use the following value if you are unable to find a matching parameter definition.” It does not result in a “definition” for the parameter in question, nor does it affect any other place from which the same parameter might have been referenced, even from the same Runnable (unlike the `-g` default that has a more global effect).

For example, suppose you have a `covtypes` parameter that compiles your source with the various coverage types. Then suppose you have several `vlog` commands, two of which are as follows:

```
<command>vlog +cover= (%covtypes%) top.sv</command>  
<command>vlog +cover= (%covtypes%) +define+DUT dut.sv</command>
```

In this case, if the `covtypes` parameter are not defined with one or more appropriate coverage types (such as "bces..."), the resulting empty string would cause the following two commands to be executed:

```
<command>vlog +cover= top.sv</command>  
<command>vlog +cover= +define+DUT dut.sv</command>
```

This script would most likely fail. In this case, the author of these two commands can prevent a gibberish command from being generated by supplying reference-specific defaults with the parameter references as follows:

```
<command>vlog +cover= (%covtypes:bsce%) top.sv</command>  
<command>vlog +cover= (%covtypes:bsce%) +define+DUT dut.sv</command>
```

That is certainly simple enough. But suppose you decide that, unless stated otherwise, `top.sv` should only be compiled with statement coverage but the DUT should be compiled with the whole gamut of code coverage (except toggle). You could (and probably should) use separate parameters for DUT and/or testcase coverage. But defining a separate parameter to hold the coverage types for each file can get pretty messy if you have lots of files to compile. To

establish a default pattern that you can still override if necessary, you could add the following reference-specific default values to the parameter references:

```
<command>vlog +cover=(%covtypes:s%) top.sv</command>  
<command>vlog +cover=(%covtypes:bscef%) +define+DUT dut.sv</command>
```

Given that these commands are not only in the same Runnable but are, in fact, in the same Action script (this is something you cannot do with the `-g=/=-G` mechanism). Again, one can argue that this is best accomplished by defining separate parameters to allow independent control of how each file is compiled, but there are doubtless other scenarios where this kind of fall back value can be useful. The important thing to remember is that this fall back value belongs to the reference and does not constitute a definition of the parameter.

In some cases, it is desirable for a parameter reference for which there is no corresponding parameter definition to use the default empty string but without declaring an error. In this case, an empty reference-specific default can be used. For example,

```
<command>echo "Compiling with coverage types '(%covtypes:%)'</command>
```

The colon (:) indicates the start of a parameter-specific default but no default value is supplied. In this case, the parameter is silently replaced with the empty string when no other definition for `covtypes` is found.

<b>Predefined Parameter Usage .....</b>	<b>266</b>
<b>Including TCL Code in a Parameter Definition .....</b>	<b>267</b>
<b>Referring to Source Files Via a Parameter .....</b>	<b>269</b>
<b>Nested Parameter Expansion and TCL Parameters .....</b>	<b>270</b>
<b>Inheritance, Overrides, and Search Order .....</b>	<b>271</b>
<b>Parameterizing the Regression Suite for Debug .....</b>	<b>283</b>
<b>Exporting Parameters to Action Scripts .....</b>	<b>284</b>

## Predefined Parameter Usage

A small number of parameters are predefined with values equal to certain global paths or names relevant to VRM algorithm.

Refer to the section “[Predefined Parameters](#)” for a complete listing.

These predefined parameters can be used in parameter references where explicit path names or unique file names are required. For example, the HDL source code for a design might be located in a `src` subdirectory within the same directory where the RMDB database is stored. In this case, the `RMDBDIR` predefined parameter can be used to locate the source files, as in

(%RMDBDIR%)/src. This removes any dependency between the nesting depth of the *preScript* Action containing the compile commands and the relative location of the design source files. It is best to avoid such dependencies, as the Group nesting is designed to make it easy to re-arrange the regression hierarchy as often as necessary.

Note that by convention, the names of parameters defined internally by VRM are capitalized to avoid namespace collisions with user-defined parameters. In order for this convention to work, it is recommended that user-defined parameters use lower-case names. This also makes it easier to determine, when looking at an Action script, which parameters are supplied by VRM (and are, therefore, dependent on the context of the Action being run) and which are supplied by the user (and, therefore, are defined elsewhere in the database or on the *vrun* command line).

Also note that predefined parameters can be overridden by the *-G* command-line option. In most cases, the override only affects the value used to expand a parameter reference that refers to the predefined parameter in question, as VRM maintains its own copies of the relevant values. However, caution should be exercised when overriding predefined parameters, as doing so can adversely affect the VRM algorithms.

## Including TCL Code in a Parameter Definition

By default, parameter values are used as is, exactly as they appear in the parameter definition (after external entities (like >) are converted back to the characters they represent). It is possible to define a parameter value as a TCL expression that undergoes the usual variable/command substitution within the TCL interpreter each time the parameter is expanded. Note that this is **TCL substitution, not evaluation**. In order to execute TCL code as part of the parameter expansion, the code needs to be defined as a procedure and command substitution used to execute the procedure.

In order to specify that TCL substitution should be performed on a parameter value, a *type* attribute whose value is *tcl* must be attached to the parameter element. For example, this functionality can be used to initialize a parameter value with the value of an environment variable as follows:

```
<parameter type="tcl">$::env(USER) </parameter>
```

The TCL expansion of this parameter results in a value that equals the value in the *USER* environment variable (on most Posix-like systems, that is the username of the user who invoked VRM). This expansion takes place in the same namespace as that used for user-defined procedures. Therefore, if a procedure is needed to compute the value of a parameter, that procedure can be defined in a *usertcl* method in the RMDB database or loaded from the

command line via the `-loadtcl` option. For example, the following computes the random seeds based on the time of day:

```
<rmdb loadtcl="getseeds">
  <usertcl name="getseeds">
    proc getSeeds {} {
      if {[clock format [clock seconds] -format %H] > 20} {
        return [list 10 20 30 40 50 60]
      } else {
        return [list 10 20 30]
      }
    }
  </usertcl>
  ...
  <runnable name="nightly" type="group">
    <parameters>
      <parameter name="seeds" type="tcl">[getSeeds]</parameter>
    </parameters>
    <members>
      <member>test</member>
    </members>
    ...
  </runnable>
  <runnable name="test" type="task" foreach="(%seeds%)">
    ...
  </runnable>
  ...
</rmdb>
```

The `foreach` attribute contains a reference to the `seeds` parameter. This parameter, when expanded, resolves to the string `[getSeeds]`. The TCL substitution causes the `getSeeds` command (which can be found in the auto-loaded `getseeds usertcl` element) to be executed which, in this case, picks up one of two lists of seeds selected based on the time of day (presumably the user does not want to waste compute resources during the working day by running all their favorite seeds).

Caution is advised when using TCL code in parameters (or in the `if`/`=unless` attributes). The parameter value string, after expanding any nested parameter references, is passed to the TCL `subst` command for expansion. If any errors occur as the TCL interpreter parses the string, an error message is emitted and the parameter behaves as if it were undefined (at that point, any applicable inheritance search path is still followed).

In the case of the `if` and `unless` attributes, the parameter-expanded attribute value string is passed to the TCL `eval` command. If any errors occur as the TCL interpreter parses and attempts to evaluate the value of the string, an error message is emitted and the attribute behaves as if it had not been specified (that is, an `if` attribute behaves as if the errant expression evaluated to true and an `unless` attribute behaves as if the errant expression evaluated to false).

Using TCL-expanded parameters alters the expansion algorithm slightly. Rather than substituting the parameter value (as is) in place of the parameter reference and iteratively

searching for new parameter references that may have appeared as a result of the substitution, a TCL-expanded parameter is completely parameter-expanded as a unit before being substituted into the original string. What was an iterative process is now recursive. For most non-complex applications, this makes no difference in the apparent behavior of the resolution process. For complex cases, it can make a difference.

## Referring to Source Files Via a Parameter

One often needs to refer to source files (external scripts, HDL source, *vsim do* files, and so on) as part of a complete regression run. One of the best places to store these source files is in the same directory as, or a subdirectory of, the directory from which the *RMDB* file is read. The reason is that the user can refer to the *RMDB* file in the *-rmdb* command-line option from any directory and the relevant source files references are still correct (one can solve the problem by using absolute paths as well, but the use of absolute paths can cause additional problems if the source files are ever moved to some other directory in the file system).

Most file-reference attributes are already relative to the *RMDBDIR* directory (that is, the directory from which the *RMDB* file is read). In order to provide the same relative file reference functionality for parameters, a *file* special parameter type is supported. When the parameter type is *file*, the contents of the parameter are combined with the *RMDBDIR* path and the combined path is normalized based on the platform on which *vrun* is executed.

The following example assumes the *runall.do* file is located in the same directory as the *RMDB* file:

```
<runnable name="test1" type="task">
  <parameters>
    <parameter name="dofile" type="file">runall.do</parameter>
  </parameters>
  <execScript>
    <command>vsim -do (%dofile%) ...</command>
  </execScript>
</runnable>
```

The TCL code executed to resolve this parameter is as follows:

```
[file nativename [file normalize [eval file join (%RMDBDIR%)
  <parameter-value>]]]
```

The value defined for the parameter can consist of multiple directory components. In this case, either the slash character or the space character can be used and the resulting file name should be correctly converted to a normalized path.

Note that the expansion occurs within VRM as the parameterized script file is generated. In the case where the *vrun* command is executed on one platform but the Action is passed off to a different platform for execution, this shortcut may not work. If the script can be executed on a platform other than that on which *vrun* is running, then the TCL *file normalize* and/or *file*

*nativename* command(s) should be executed as part of the Action script to ensure that the path is in the appropriate form for the platform on which the Action script is executed.

## Nested Parameter Expansion and TCL Parameters

The examples in this section are complete and will run as is under VRM; however, they are too simple to be real “examples” on their own. In any event, use the code herein and add to it when appropriate.

The following example illustrates nested parameters and a TCL-evaluated parameter:

```
<?xml version="1.0" ?>
<rmdb>
    <runnable name="top" type="group">
        <parameters>
            <parameter name="go">GO</parameter>
            <parameter name="goal" type="tcl">[join [list (%go%) (%al%)] { }]</parameter>
        </parameters>
        <members>
            <member>leaf</member>
        </members>
    </runnable>
    <runnable name="leaf" type="task">
        <parameters>
            <parameter name="al">AL</parameter>
        </parameters>
        <execScript launch="exec">
            <command>echo I have reached my (%goal%)</command>
        </execScript>
    </runnable>
</rmdb>
```

The command in the *execScript* element in the *leaf* Runnable refers to the *goal* parameter. The parameter *goal* is defined as a TCL-expanded parameter under the parent *top* Group Runnable. The *goal* parameter value first expands the two nested parameters *go* and *al*, from the point of view of the *leaf* Runnable on whose behalf the *execScript* Action is being run. The *go* parameter is defined in the *top* Runnable while the *al* parameter is defined in the *leaf* Runnable. Together they form a TCL expression that evaluates to the word *GOAL* which is then substituted into the command.

When this Action is executes (in context) with the following command:

**vrun top/leaf**

the following line should appear in the *execScript.log* output file:

I have reached my GOAL

# Inheritance, Overrides, and Search Order

When considering the configuration of most regression suites, there is usually some degree of repetition. If, for example, 100 tests all require the same simulation commands with some minor variation, it would be a poor system that forced the user to specify the redundant information 100 times, once per test.

Inheritance, combined with parameterization, seeks to reduce the need for redundant data entry to zero (or as close to zero as possible). In VRM, the following two forms of inheritance are supported:

- One is explicitly called out by the user in the RMDB database.
- One is implicit in the membership hierarchy of Groups and Tasks.

Both forms combine to provide the user with the power to specify relatively complex regression suites with a minimum of verbiage.

The explicit form of inheritance is called “base inheritance,” in that its role in the topology is similar to base classes in an object-oriented language. Base inheritance is supported for all three top-level elements: *Runnable*, *Method*, and *UserTcl* elements. To specify a base inheritance relationship, a *base* attribute is added to the element that seeks to inherit content from some other element. The value of this *base* attribute is a list of names of elements of the same type, separated by spaces and/or commas. The attribute is called *base* to indicate that the target of that relationship link serves as the “base” on which the referring element is built. For example, consider the following XML fragment:

```
<Runnable name="bob">
    <!-- bob's content -->
</Runnable>
<Runnable name="fred" base="bob">
    <!-- fred's content -->
</Runnable>
```

In this example, the *Runnable* element named *fred* refers to the *Runnable* element named *bob* via the *base* attribute. The effect of this reference is that anything defined inside of the *Runnable* element named *bob* (including parameters, Action scripts, member lists, and so on) is treated as if it were also defined inside of the *Runnable* element named *fred*, unless the *fred Runnable* element already defines the construct in question. That is, if *bob* defines an *execScript* but *fred* does not, then any search for an *execScript* within *fred* is satisfied by the definition found within its base *Runnable* element (*bob*).

The implicit form of inheritance is called “group inheritance,” in that it derives naturally from the arrangements of Runnables into Groups. While base inheritance is a static property of the configuration of the regression suite, group inheritance can only be resolved at runtime, when the complete calling graph of Groups and Tasks has been constructed. As mentioned above, a Task can be a member of multiple Groups. When it is executed as a member of one Group, its group inheritance chain is different from that of the same Task when executed as a member of

some other Group. Group inheritance derives from membership in a Group, as seen in the following example:

```
<runnable name="parent" type="group">
  <members>
    <member>child</member>
    <!-- other members -->
  </members>
  <!-- other content -->
</runnable>
<runnable name="child" type="task">
  <!-- content -->
</runnable>
```

When the *child* Task is executed as part of the *parent* Group, its immediate group ancestor is the *parent* Runnable.

Since Groups can be members of other Groups, the complete graph of a typical regression suite can be rather complex, even if the configuration of that suite is not. The complete sequence of events from the initial Group named on the VRM command line to the leaf-level Task under consideration is called the “calling chain” of the Task. At the very end of that calling chain are the specific Action scripts invoked to carry out the regression subtasks. In log messages and many other places within VRM, these calling chains are displayed in a manner similar to paths in a directory structure (in fact, they correspond directly to working directory paths within the *VRMDATA* directory tree). For instance, the string:

```
nightly/random/randtest1/execScript
```

refers to the *execScript* Action script of the *randtest1* leaf-level Task, that is called as a member of the *random* Group, which itself is called as a member of the *nightly* Group, which itself is invoked from the VRM command line. The Group that is the immediate group ancestor of a given Task or Group is sometimes called the “parent” Runnable of that Task or Group.

In the case of base inheritance, the *base* attribute of the referring element can specify multiple base elements. This could be looked upon as “multiple inheritance” except without most of the baggage associated with that term in other programming languages. The base element names can be separated by either whitespace or commas (or both). Each base element name is made up of one or more non-space/non-comma characters and is assumed to be the name of another element of the same type as the referring element that is defined at the top-level of the database.

Following is an example that defines two Tasks, each of which refers to two base Runnables:

```
<runnable name="base0" type="base">...</runnable>
<runnable name="base1" type="base">...</runnable>
<runnable name="base2" type="base">...</runnable>
<runnable name="task10" type="task" base="base1, base0">...</runnable>
<runnable name="task20" type="task" base="base2, base0">...</runnable>
```

In this example, both Task Runnables (*task10* and *task20*) inherit the contents of the base Runnable *base0* (to the extent there is no overlapping content already defined in the *task10* and

*task20* Runnables themselves), whereas *task10* also inherits the contents of *base1* and *task20* also inherits the contents of *base2*. The *base1* and *base2* Runnables can serve as a sort of “categorization” scheme, each imparting some content to other Runnables in their respective categories while the *base0* Runnable lends content that is common across all (or at least a significant number of) Runnables in the regression suite. This same functionality applies to *method* and *usertcl* elements as well.

Where multiple base elements are specified via the *base* attribute, the order in which the base elements are specified determines the priority in the event similar content is defined in both base elements. The search rules are as follows:

1. Since VRM treats the contents of any base element as though the contents were defined directly in the referring element (except when overlapping local content exists), VRM always follow the base inheritance chain before the group inheritance chain (for *Runnable* elements, that is the *method* and *usertcl* elements do not support group inheritance).
2. If there are multiple base elements, then VRM completely analyzes the first base element **before** continuing on to analyze the second. The order in which the element names appear in the value of the *base* attribute of the referring element is the determining factor.
3. In the case of *Runnable* elements, if a search is not satisfied by any base Runnables on the base inheritance chain, then the search continues with the immediate parent Runnable on the group inheritance chain.

Note that this “multiple inheritance” only applies to base inheritance. This is not an arbitrary rule but the result of the definition of group inheritance. Unlike human lineage, it only takes one Group Runnable to act as the “parent” in kicking off the execution of a “child” Runnable from its member list. Therefore, with respect to group inheritance, a Runnable only ever has **one** immediate parent (but a potentially unlimited group inheritance chain of ancestors from which to draw, of course).

Both reconvergence and cyclicity (that is, loops) in the base inheritance topology are allowed and handled gracefully. Consider the famous “diamond” inheritance topology represented by the following collection of Runnables:

```
<Runnable type="task" name="top" base="left right"/>
<Runnable type="base" name="left" base="bottom"/>
<Runnable type="base" name="right" base="bottom"/>
<Runnable type="base" name="bottom"/>
```

In this case, the search order would be *top*, *left*, *bottom*, then *right*. As odd as that may seem at first, the principle is that anything defined in *bottom* is treated as if it were also defined in both *left* and *right* (per the definition of base inheritance). So, when you search *left* you are implicitly searching *bottom* as well. The search order *top*, *left*, *right*, then *bottom* requires a special exception to be made in the case of multiple base elements.

Both types of inheritance are recursive. Group inheritance is implicitly so, tracing backward along the calling chain leading to the Task or Action script in question. Base inheritance is explicitly recursive by virtue of the fact that a “base” Runnable can, itself, define another *base* Runnable via its own *base* attribute. When searching for a parameter or Action script definition or for a Group membership list, the search follows the chain of base inheritance pointers until the desired object is located or a *Runnable* element without a *base* attribute is encountered.

Cyclicity (loops) in the group inheritance chain is illegal, as one cannot be one's own ancestor/descendant. The base inheritance chain, however, ignores cyclicity. It is perfectly legal for two elements to refer to each other as base elements as follows:

```
<Runnable type="task" name="left" base="right"/>
<Runnable type="task" name="right" base="left"/>
```

According to the definition of base inheritance, anything defined in the base Runnable is considered to have been defined in the referring Runnable. Therefore, barring any overlapping content, what this does is define two Tasks which, for all practical purposes, contain identical content. This is a contrived example to illustrate a point and not something one might do intentionally. But the “diamond” configuration above can occur in a complex configuration either intentionally or inadvertently and is not an error in either case. A simple cyclicity check, however, would be dumbfounded by the “diamond” topology and the check would thus be very difficult to get right. The base inheritance walking algorithm quietly terminates its traversal when it detects any duplication with elements that have already been examined (due to either cycles or reconvergent fanout in the base inheritance topology) on the assumption that a second search of that part of the inheritance graph will be as unfruitful the second time around as it was the first. On the other hand, a cycle in the group inheritance chain is squawked as an error and search traversal terminates at the point where the cycle is detected in order to avoid getting into an infinite search loop.

It is also **not** an error for a group ancestor to refer to a descendant or one of the descendant's base Runnables (or vice-versa) via base inheritance. Any content search will still terminate when it detects the duplication. In the case of a *parent* and *child* both referring to the same base Runnable, the given base Runnable (and further base Runnables along the base chain, if any) would have already been searched as part of searching at the *child* level so there is no point searching again when we are at the *parent* level (the content of the Runnables is static once the RMDB database is loaded). One possible application of this would be a base Runnable defining a parameter that applies to all Groups starting with the letter *a*. Since one could easily have a Group Runnable whose name starts with an *a* and which contains a member Runnable whose name also starts with *a*, these seemingly odd topological twists can easily occur in a complex regression suite and are generally harmless.

Note that a unique situation exists in the case of *Runnable* elements, since they support both base and group inheritance. Taking parameter definitions as an example, the list of parameters “visible” from any given Runnable can contain parameters defined in either base elements or group ancestors. The question can arise as to which type of inheritance should be used in what circumstances. Since a parent is shared by all Runnables that are listed as members of that

parent (which itself must be a Group in order to be a parent), parameter definitions intended to be visible to all members of a given Group (and any descendants thereof) should be defined in the common Group shared by all the Runnables who should have access to that parameter definition.

On the other hand, parameter definitions that should apply to multiple Runnables that do not share a common ancestor (that is, Runnables that are scattered throughout the regression suite hierarchy) should be defined in base Runnables and referred to via the base inheritance mechanism. Parameters that should apply to all Runnables can be referred to via a common base Runnable but it would result in less configuration hassle to simply add that parameter to the top-most Runnable of the regression suite topology (which, in theory, should be visible from any Runnable in the suite, barring more local definitions overshadowing the top-level definition). The determination should be made on a case-by-case basis according to the topology of the regression suite and the purpose of the parameter. There is no fixed rule.

Only the “content” of an element can be inherited (that is, text content and/or other elements defined within the element in question). Attributes of a given element (for example, the *type* attribute of a runnable element) cannot be inherited by other elements. In other words, if a Runnable of type *task* points to a Runnable of type *group* as its base, then the referring Runnable remains a *task* Runnable. However, any parameter or Action script elements defined within the base *group* Runnable are treated as if they also occurred within the referring *task* Runnable.

Base inheritance includes all “content” data in the base Runnable. Action scripts, parameters, and membership lists can all be inherited from a base Runnable. However, only parameter definitions or the *execScript* definition can be inherited via the group inheritance chain. The reasons for this become clear after considering the execution model. Every parent Group along the calling chain can potentially define its own *preScript* and/or *postScript*, the absence of which must indicate the absence of any required action at that juncture. The *execScript*, on the other hand, **must** be defined for each leaf-level Task and **cannot** exist for Groups under the current execution model. Therefore, it is safe to allow *execScript* definitions to be inherited from parent Groups while considering *preScript* and *postScript* definitions to belong exclusively to the Group in which they are defined. As for Group membership lists, there is no benefit to allowing group inheritance of Group membership lists because in order for a Group to even have a parent, it must be listed in the membership list of its parent Group. Yet it cannot be listed as a member of itself without introducing a cycle into the graph. Therefore, membership lists cannot be inherited via the group inheritance chain.

When searching for a parameter or an *execScript* definition, both dimensions of inheritance come into play. The chain of base Runnables is queried first on the grounds that the contents of a base Runnable are always treated as if those contents had been directly specified in the referring Runnable. If the search remains unsatisfied after walking the base inheritance chain(s), the search proceeds to the immediate parent Runnable and continues from there (searching, as always, any base Runnables of said parent before ascending to the next parent).

So one can see that even though the inheritance mechanism is similar, the purpose of the two types of inheritance differ in substantial ways. Base inheritance provides a static way to collect the common semantic elements of multiple Groups or Tasks into one place in order to save typing. The contents of a base Runnable are considered part of the contents of the referring Runnable, except only when the referring Runnable already defines that which is sought after. Group inheritance provides a dynamic way to define common behavior among the members (or descendants) of a particular Group when (and only when) invoked within the context of that Group. It allows a given Task or Group of Tasks to be run multiple times with slight variations in behavior each time (for example, debug vs. non-debug runs).

As an example of how base inheritance differs from group inheritance and why the latter is not sufficient in itself, consider the case of several Groups, each representing a different configuration of the design under verification. Let us also say that you have defined a number of Tasks that each hold nothing but a randomization seed. The list of seeds can be applied against each of the design configurations simply by naming each of the Tasks as a member of each of the Groups. But that would require replicating the complete member list in each Group. Adding a new Group or Task to this structure would require changes in several places. Using base inheritance, the user can move the list of Tasks (which, in fact, represents a list of seed values) to a common Runnable definition and refer to it from each of the Groups. Adding a new Group or Task now involves only one edit/addition to the RMDB database, and then of course that new Group or Task will presumably be referred to in one or more other Runnables.

In the specific case of parameters, the search also involves any *-g* or *-G* options that may have been specified on the command line. The semantics of the *-g* and *-G* options are detailed in [“Override Parameter Values from Command Line”](#) on page 132. These options can provide parameter values on specific Runnables (or specific calling chains) that were not specified in the original RMDB database. In this case, the search takes these into consideration at each Runnable it encounters.

It might be useful at this point to give a more explicit definition of what it means to say something is “defined” in a given Runnable:

- A Group Runnable defines a member list when a non-empty *members* element is present within the runnable element.
- A Runnable defines an Action script when an element corresponding to that script is present within the *Runnable* element (even if that element is empty).
- A Runnable defines a parameter of a given name when there exists a *parameter* element within the *Runnable* element whose name matches that given name.

Note that an “empty” member list (that is, a *members* element containing no child *member* elements) is not considered to define members. The empty *members* element will not cause the search for a membership list to terminate. Normally, there is no legitimate reason for a Group to define a base Runnable with members but to intentionally have no members itself.

To create a group with no members, ignoring any membership list defined as part of a base Runnable, simply define an empty Task and make it a member of the Group that should behave as if it has no members. Said Group will no longer be a “zero-member” Group (it will be a Group with one member that does nothing, resulting in pretty much the same effect). Also, it might be necessary to add an empty *execScript* element to the empty Task Runnable in order to avoid the “no *execScript*” warning.

Note also that the search algorithm is implemented in VRM itself and not in the VRM Database (RMDB) API. The API returns results only for information defined directly within the Runnable named as an argument of the API command.

A special case exists when group inheritance is applied to an *usertcl* element. Other elements (besides *usertcl*) are searched, not loaded. Suppose, for example, that you are looking for a definition of the parameter *fred* in a base chain of *runnable* elements. If you do not find a matching parameter element in the first Runnable searched, then you look at the base *runnable* element(s). If you do not find a definition for *fred* there, then you look for any base elements of those elements, and so on. Once you find the data you are looking for (that is, a *parameter* element by the name of *fred*), the search stops and further base elements are not consulted.

The *usertcl* element, on the other hand, gets loaded and not searched. So even though there can be real TCL code defined in the first element you load, that does not terminate the chain. Once the first element is loaded, all the base elements on the base chain starting from that element are consulted (and TCL code therein loaded) no matter what.

The principle is that anything defined in the base element is considered to have been defined in the referring element. Therefore, if a *usertcl* element containing a *base* attribute is loaded, the base inheritance chain is followed in the same order as with any other group inheritance chain and the TCL contents of the base *usertcl* elements are also loaded. If, for example, a number of *usertcl* elements are to be selected for loading via a command-line option, then those elements can be referred to as the *base* elements of a collecting *usertcl* element which is then, itself, called out on the command line. For example, if you have the following *usertcl* element in the RMDB file:

```
<usertcl name="loadall" base="tcl1, tcl2, tcl3"/>
```

then the command line:

```
vrun -loadtcl @loadall ...
```

would cause the TCL contents of the *usertcl* elements *tcl1*, *tcl2*, and *tcl3* to all be loaded. This would occur despite the fact that the *usertcl* element *loadall* was, itself, empty.

**Table 8-1** summarizes the types of data (and meta-data) items found in the RMDB database and which modes of inheritance apply to each.

**Table 8-1. RMDB Database Data Item Types**

Element Base	Data Item Group	Inheritance from Base	Inheritance from Group
runnable	parameters	Yes	Yes
runnable	execScript	Yes	Yes
runnable	preScript/postScript	Yes	No
runnable	members list	Yes	No
method	method command	Yes	No
usertcl	TCL content	Yes	No
(all)	attributes (note that attribute meta-data is always specific to the XML element to which it is attached)	No	No

<b>Parameter Inheritance Example .....</b>	<b>278</b>
<b>Runnable Types and Inheritance .....</b>	<b>280</b>
<b>Search Order When Expanding a Parameter Reference.....</b>	<b>281</b>

## Parameter Inheritance Example

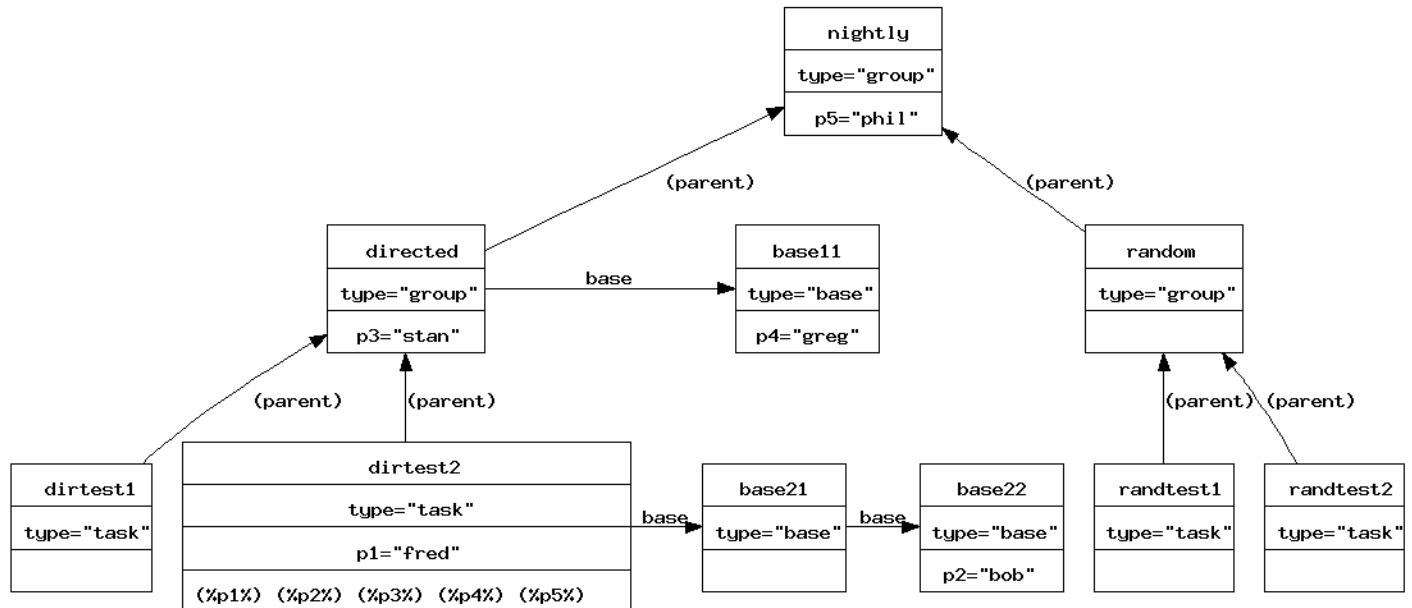
The following shows an example of base/group inheritance in the context of parameter expansion.

The example regression suite consists of two Groups (*directed* and *random*), each with two Tasks. The two Groups are members of a single top-level Group (*nightly*) that can be used to run the entire regression suite in one shot. Assuming, while processing the *nightly/directed/dirtest2/execScript* Action script, you encounter the following Action script command (shown at the bottom of the *dirtest2* runnable node in [Figure 8-1](#)):

```
<command> (%p1%) (%p2%) (%p3%) (%p4%) (%p5%) <command>
```

Figure 8-1 illustrates how those five parameters are resolved:

**Figure 8-1. Parameter Inheritance Diagram**



Parameter *p1* is defined in the *dirtest2* Runnable itself so that value is used to expand that parameter reference. Parameter *p2* is not defined in the *dirtest2* Runnable but it is defined in the *base22* Runnable that is on the base inheritance chain. Parameters defined in base Runnables on the base inheritance chain are treated as if they were defined in the Runnable itself so, in this case, the value in the *base22* Runnable is used to expand the *p2* parameter reference.

In the case of the *p3* parameter reference, there is no *p3* parameter defined in the *dirtest2* Runnable or in any of its base Runnables. Therefore, we defer the search to the Group that invoked the *dirtest2* Task (in this case, the *directed* Group). This Runnable does define a *p3* parameter so that value is used to expand the *p3* parameter reference. Note that if the *dirtest2* Runnable is included as a member of multiple Groups, the group (parental) inheritance chain leads us to a different Group each time the *dirtest2* Runnable is invoked. This is how parameterized Runnables can be reused in different Groups with different results.

The *p4* parameter is not defined in the *dirtest2* Runnable, its base Runnables, or in its immediate group ancestor Runnable. It is, however, defined in the *base11* Runnable that is a base of the *directed* Runnable. Therefore, we satisfy the *p4* parameter reference using this parameter definition. To expand the *p5* parameter reference, it is necessary to follow the group inheritance chain to the top-level *nightly* Group.

Not shown in this diagram is the effect of default values and command-line overrides. These are fairly straight-forward and are discussed in “[Override Parameter Values from Command Line](#)” on page 132.

## Runnable Types and Inheritance

From the point of view of base inheritance, all Runnable nodes are equivalent. A Group can refer to a Task as a base Runnable or vice-versa (since base inheritance is concerned only with content, not behavior). Data defined in the base Runnable is treated as if it is defined in the referring Runnable itself. While there is nothing preventing a base Runnable from participating in the execution graph of the regression suite, it is more common for Runnables used as a base for other Runnables to not be involved in the execution graph themselves. In such cases, it can be desirable to enforce this non-involvement so as to prevent inadvertent cycles in the graph. To mark a Runnable as non-participatory, set the value of the *type* attribute to *base* instead of *task* or *group*. If a *base* type Runnable is included in the membership list of a Group, then VRM outputs an error and ignore the *base* type Runnable. Note that this is similar to the “pure virtual base class” concept in C++ (a base that cannot, by itself, be instanced).

# Search Order When Expanding a Parameter Reference

---

There is, of course, no reason to expect that there would be only one definition for any given parameter, or that the parameter in question is defined in the same Runnable from which it is referenced. VRM parameters (and other element content) can be inherited from other Runnables. There is a priority order (that is, search order) used to determine the best definition to use in expanding a given parameter on behalf of a parameter reference found in a given Runnable. These rules are outlined below.

<b>Inheritance Rules .....</b>	<b>281</b>
<b>Parameter Expansion Rules .....</b>	<b>281</b>

## Inheritance Rules

For any given element, a content element (for example, a parameter or Action script, in the case of a Runnable) is resolved as follows:

1. If the element in question defines one or more content elements that match the search criteria (that is, parameter name, in the case of a parameter), the first matching content element (in document order) is used to resolve the reference and the search is terminated.
2. If the element in question defines a *base* attribute, the value of that attribute is parsed as a comma/space separated list of tokens and each is interpreted as the name of a base element to be searched.
3. In the event there are multiple tokens in the *base* attribute, the elements to which those tokens refer are searched in the order they appear in the *base* attribute.
4. In the case of a chain of base elements (that is, the base element itself has a *base* attribute), each base element corresponding to a token in a list of element names in a single *base* attribute value is searched completely before the next token in the list of element names is considered.
5. If the base inheritance chain produces no matching content element, and if the element in question is a *runnable* element, then the search continues with the immediate parent of the element in question.

## Parameter Expansion Rules

The following points form the complete set of rules by which parameter references are expanded.

For a complete understanding of these rules, it may be necessary to first understand the section on “[Override Parameter Values from Command Line](#)” on page 132.

1. For each Runnable searched according to the inheritance rules given in “[Inheritance Rules](#)” on page 281:
  - a. If an override (-G) command-line option matches the Runnable instance under examination, that override value is used and the search is terminated. In the case of multiple matching override options, priority is given to the following conditions, in sequence:
    - i. A rooted context-specific override (for example, -G/a/b/c/p=v, where c is the name of the Runnable in question, /a/b is the calling context of that Runnable, and a is a top-level Runnable).
    - ii. A non-rooted context-specific override with context (for example, -Ga/b/c/p=v, where c is the name of the Runnable in question and a/b is the calling context of that Runnable). In the case of multiple matching override options in this category, priority is given to the one with the most specified levels of calling context (that is, the most specific).
    - iii. A Runnable-specific override (for example, -Gc/p=v, where c is the name of the Runnable in question).
    - iv. A generic override (for example, -Gp=v).
  - b. If the parameter in question is defined in the *Runnable* element under consideration and no override is found, then that value is used and the search is terminated.
  - c. If multiple override options for the same parameter with identical context are issued on the same command line, then the last such option to appear dominates, and all earlier such options are ignored. (For example, if the following two options appear on the same command line, -Ga/b/c/p=v1 -Ga/b/c/p=v2, then only the v2 option is considered, and the v1 option is ignored.)
2. If the parameter is not resolved from the above combination of parameter definitions and -G overrides, then the -g default options are considered. The same inheritance search rules are used to revisit each Runnable in turn and, at each Runnable, the same priority rules are used to select from among multiple matching default options as are used for the -G override options.
3. If a parameter remains undefined after inheritance searches and application of -G/-g options, then any reference-specific default is applied.
4. If, after all this searching, no parameter is found to match the parameter reference in question, the parameter reference is removed from the command and nothing is substituted in its place. In addition, an error is reported in the VRM log output.

# Parameterizing the Regression Suite for Debug

One requirement of most regression environments is the ability to run one or more tests in “debug mode” either early in the design process or in response to failures exposed in the nightly run. Assume that enabling debug in your regression suite requires the use of additional options on both the compile command line and the simulation command line. Assume that you can define the compile command options as a parameter called *cmoops* and the simulation command options as a parameter called *simopts*. There are at least two ways in which these options can be applied to the regression suite.

To make sure that the debug runs do not clobber the non-debug runs (especially the Questa *work* directory), use a separate top-level Group for the debug run. The top-level Group (which is run every night) is called *nightly*. From this, you can construct a “nightly-debug” group that runs the entire suite in debug mode as follows:

```
<runnable name="nightly" type="group">
  <parameters>
    <parameter name="cmoops"/>
    <parameter name="simopts"/>
  </parameters>
  <members>
    <member>test1</member>
  </members>
  <preScript>
    <command>vcom (%cmoops%) ...</command>
  </preScript>
  <!-- other normal non-debug contents -->
</runnable>
<runnable name="nightly-debug" type="group" base="nightly">
  <parameters>
    <parameter name="cmoops"><!-- compile-command
      options --></parameter>
    <parameter name="simopts"><!-- simulation-command
      options --></parameter>
  </parameters>
</runnable>
<!-- possibly many intervening Groups and Tasks -->
<runnable name="test1" type="task">
  <execScript>
    <command>vsim (%simopts%) ...</command>
  </execScript>
  <!-- other normal non-debug contents -->
</runnable>
```

When the “*nightly*” Runnable is executed, the *cmoops* and *simopts* parameters are empty and, therefore, contribute no additional information to the compile or simulation command lines. When the *nightly-debug* Runnable is executed, the same data as in the *nightly* Runnable is used with two exceptions: (a) the *cmoops* and *simopts* parameters have been given non-empty values, and (b) the working directory created for the debug run is called *nightly-debug* instead of *nightly*.

If the goal is to run the debug mode simulations in the **same** working directory as the non-debug simulations, the same *nightly* Runnable can be invoked and the compile and simulation command options can be passed in from the command line as follows:

```
vrun -rmdb mycfg.rmdb -Gcmopts='<compileoptions>' -Gsimopts='<simulation options>'
```

If the parameter override values contain embedded spaces and/or shell meta-characters, the parameter value strings must be appropriately quoted or the meta-characters escaped. How to do this depends on the shell being used to launch the *vrun* executable.

## Exporting Parameters to Action Scripts

In some cases, especially where an existing *do* file or shell script is used in conjunction with VRM, an Action script can make use of the value of a parameter as an environment variable. The most obvious way to do this is to add commands to the script (or to a wrapper around the script) that explicitly set environment variables based on parameter values.

For example, the following *execScript* sets two environment variables based on VRM parameter values before invoking a legacy script:

```
<parameters>
    <parameter name="vsimopts">....</parameter>
    <parameter name="ucdbfile">....</parameter>
    ...
</parameters>
...
<execScript launch="/bin/csh">
    <command>setenv vsimopts (%vsimopts%)</command>
    <command>setenv ucdbfile (%ucdbfile%)</command>
    <command>source myLegacyScript.csh</command>
</execScript>
```

While this works as expected, VRM provides a short-cut to achieve the same functionality. Any parameter that is marked with an *export* attribute whose value is “yes” is automatically exported to the environment of any Action script from which that parameter is visible.

For example, the following RMDB fragment has the same effect as that shown above:

```
<parameters>
    <parameter name="vsimopts" export="yes">....</parameter>
    <parameter name="ucdbfile" export="yes">....</parameter>
    ...
</parameters>
...
<execScript launch="/bin/csh" file="myLegacyScript.csh"/>
```

In this case, the *execScript* is read directly from the legacy file and the *export* attribute on the two parameters ensures that environment variables of the same name and value as the two parameters shown will exist in the Action script’s environment by the time the script assumes

control. The environment variable names will match the names of the parameters (including case).

In order to change the case of the exported parameter, a second (exported) parameter should be created as follows:

```
<parameters>
    <parameter name="vsimopts">...</parameter>
    <parameter name="VSIMOPTS" export="yes">(%vsimopts%)</parameter>
</parameters>
```

Note that exported parameters do not need to be defined in the same Runnable as the Action script. Even parameters inherited from parent Groups and/or base Runnables will be exported to the Action script's environment if they are marked with the appropriate *export* attribute.



# Chapter 9

## Working Directory and Script Execution

---

In order to prevent multiple Actions (that is, user scripts) from a single *vrun* invocation from overwriting or otherwise interfering with each other, each Action executed by VRM is launched from within a working directory unique to the Runnable on whose behalf the Action is launched. In order to prevent multiple invocations of VRM (by other users or by the same user) from interfering with each other, the root of the scratch hierarchy within which these per-Runnable directories are created is control by the user.

<b>Working Directory Overview .....</b>	<b>288</b>
<b>Script Execution and Timeouts .....</b>	<b>296</b>
<b>Execution Methods .....</b>	<b>315</b>
<b>Execution Queues and Job Aggregation .....</b>	<b>333</b>
<b>Failure Rerun Functionality .....</b>	<b>346</b>

## Working Directory Overview

For any given regression run, each Task or Group of Tasks is assigned a unique working directory and the working directories for the member Groups and/or Tasks within any given Group are created directly under that Group's working directory in a manner reflecting the membership hierarchy defined by the RMDB database. This allows the user scripts to refer to files maintained on behalf of their parent Group (that is, the coverage merge file or design library) via relative path names such as “..” (although that does not preclude the user from passing that information via parameters defined at the Group level). Any files generated by VRM in association with a given Action (that is, Action script files, .do files, log files, and so on) are written to the working directory associated with the Runnable on whose behalf the Action is executed.

The entire working directory structure is managed by VRM and is rooted in a top-level *VRMDATA* directory supplied by the user (via the *-vrmdata* command-line option). By default, VRM creates and uses as its *VRMDATA* directory a subdirectory called *VRMDATA* in the directory from which VRM was invoked (in other words (%*DATADIR*%) defaults to (%*VRUNDIR*%)*/VRMDATA*).

Compile (work) libraries and/or source files must be visible from the working store of each test. Several known directory paths are provided as predefined parameters (see [page 266](#)) for use in resolving command parameter references. The *VRUNDIR* parameter holds the path from which VRM is invoked. The *RMDBDIR* parameter holds the path from which the RMDB database file is read. Either of these can be used as references for locating design source or other mostly static data files. The *VRMDATA* parameter holds the root directory of the *VRMDATA* (working) directory tree, providing a convenient place to store compiled libraries common to the entire regression. The *TASKDIR* parameter holds the path to the working directory associated with the current Action that can be used as a reference to locate runtime-generated data or result files.

<b>Local File Generation .....</b>	<b>289</b>
<b>Disposition of Test Collateral .....</b>	<b>295</b>
<b>File Safety .....</b>	<b>295</b>

## Local File Generation

A Runnable can define one or more local files that are generated, copied, or linked into the working directory prior to the execution of any Action scripts in that directory. Each local file to be generated into the working directory is designated by a corresponding *localfile* element contained within the *Runnable* element with which it is associated. An optional *src* attribute specifies the file or directory from which the contents of the local file are to be taken. If the *src* attribute is not specified, then the contents of the *localfile* element itself are used to generate the local file.

If the *src* attribute is not specified, then the *name* attribute is required and specifies the name of the local file to be generated. In this case, the *localfile* element can contain one or more *command* elements, each of which corresponds to a line in the local file. For example, the following element generates a local command file called *compile* in the working directory associated with the “nightly” Runnable:

```
<Runnable type="group" name="nightly">
  ...
  <localfile name="compile" expand="yes">
    <command>vlib (%DATADIR%)/work</command>
    <command>vlog -work (%DATADIR%)/work -cover sbectf
      (%RMDBDIR%)/src/tb.sv</command>
    <command>vlog -work (%DATADIR%)/work -cover sbectf
      (%RMDBDIR%)/src/dut.sv</command>
    ...
  </localfile>
  ...
</Runnable>
```

If the *expand* attribute is set to “yes”, then each line in the generated file is scanned for parameter references and any that are found are expanded. If the *expand* attribute is missing or set to anything other than “yes”, then the lines in the *localfile* element (or the source file if the *src* attribute is specified) are copied verbatim.

If the *executable* attribute is set to “yes”, then the resulting target file is set to the executable mode. This is only effective on UNIX and/or Linux platforms (Windows files do not have an executable flag).

If the *append* attribute is set to “yes”, then the contents of the element will be appended to the specified file, if it exists.

If the *src* attribute is included in the *localfile* element, the contents of the *localfile* element are ignored and the contents of the generated file are taken from the file (or directory) located at the path indicated in the *src* attribute. This path can be an absolute path or, if relative, is interpreted as being relative to the *RMDBDIR* path (that is, the directory from which the RMDB database is read).

In VRM, the *src* attribute of *localfile* elements in the RMDB supports wildcard globing (according to the rules for the TCL *glob* command). Globing works under the following conditions:

- If the *name* attribute is blank or does not exist, then the files referred to by the *src* attribute are copied/linked into the working directory for the Runnable on whose behalf the *localfile* element is being expanded.
- If the *name* attribute is non-blank and refers to a directory, then the files referred to by the *src* attribute are copied/linked into the directory referred to by the *name* attribute.

If the *src* attribute expands to more than one source file and the *name* attribute is non-blank and does not refer to a directory, an error is reported and the *localfile* element will not copy/link any files.

In the event a *src* attribute is specified, an optional *type* attribute can also be specified. If the value of the *type* attribute is “link,” a symbolic link is generated in the working directory (at the path specified by the *name* attribute) that points to the file or directory specified in the *src* attribute. If the *type* attribute is not specified or does not contain one of the two valid values (“copy” or “link”), then the file contents are copied by default. The *type* attribute is ignored if the *src* attribute is not specified.

If the value of the *type* attribute is “copy,” the behavior depends on the contents of the *src* and *name* attributes as follows:

- If the *type* attribute is “copy” and the *src* attribute points to a regular file, then the *name* attribute is optional and the single file specified in the *src* attribute is copied to the working directory. If the *name* attribute is specified, then the value of the *name* attribute is used as the name of the local file. If the *name* attribute is not specified, then the name used for the local file is the same as the basename of the source file.
- If the *type* attribute is “copy” and the *src* attribute points to a directory, then the *name* attribute can be a list of one or more filenames, each of which are copied to the working directory using the same name as in the source directory specified by the *src* attribute. If the *name* attribute is empty, then a warning is issued and nothing is copied.

Note that it is not possible to copy an entire directory using a “copy” type *localfile* element. This is to prevent inadvertent runaway copy operations. If an entire directory copy is required, it should be coded into the appropriate Action script.

In all cases (link mode, single copy mode, and multiple copy mode), the *name* attribute value(s) can contain a relative path. If a path is specified for a given value of the *name* attribute, then the directories leading to the path in question are created before the file is generated. In multiple copy mode, where the value of the *name* attribute is used along with a path to a directory from the *src* attribute, the path designated by the *name* attribute value(s) is combined with the directory in the *src* attribute to refer to the source file.

The target of the *localfile* element (path in the *name* attribute) must resolve to a path within either the current working directory (*\$cwd*) of the *vrun* process or the *VRMDATA* directory. Any attempt to copy files to a location outside those bounds results in an error message and the copy/link is not performed. The reason is to prevent inadvertent clobbering of existing files.

Note that in link mode, if a file or directory already exists at the target path, a warning is issued and the symbolic link is not created. In copy mode, if a file already exists at the target path, its timestamp is checked against that of the source file and the copy is silently ignored unless the source file is newer. If a directory already exists at the target path in copy mode, then an error is issued and the copy operation will not proceed.

Note that the use of file timestamps is intentional. The test is not whether or not the file has changed but whether or not the source file is newer (for example, *make*). If the user modifies the file in the working directory for any reason, then VRM must not copy over the modified file.

For those readers who find code easier to read than prose, the following pseudo-code describes VRM behavior under different conditions:

```
if (value(src) is empty) {
    if (value(name) is empty) {
        error; /* either value(src) or value(name) must be specified */
    } else /* value(name) is specified */ {
        if (dirname(value(name)) is NOT empty) {
            mkdir -p (%TASKDIR%)/dirname(value(name))
        }
        cp <commands-in-localfile-element> to (%TASKDIR%)/value(name)
    }
} else /* value(src) has been specified */ {
    if (isDirectory(value(src)) /* value(src) points to a directory */ {
        if (value(name) is empty) {
            if (value(type) == "link") {
                ln -s (%RMDBDIR%)/value(src) to (%TASKDIR%)/basename(value(src))
            } else /* value(type) == "copy" (by default) */ {
                error; /* whole-directory copy not supported */
            }
        } else /* value(name) is specified */ {
            if (value(type) == "link") {
                if (dirname(value(name)) != "") {
                    mkdir -p (%TASKDIR%)/dirname(value(name))
                }
                ln -s (%RMDBDIR%)/value(src) to (%TASKDIR%)/value(name)
            } else /* value(type) == "copy" (by default) */ {
                if (value(name) is empty) {
                    warning; /* nothing to be copied */
                } else /* split value(name) into multiple tokens */ {
                    foreach token in value(name) {
                        if (dirname(token) is NOT empty) {
                            mkdir -p (%TASKDIR%)/dirname(token)
                        }
                        cp (%RMDBDIR%)/value(src)/token to (%TASKDIR%)/token
                    }
                }
            }
        }
    } else /* value(src) points to a non-directory (ie: a regular file) */ {
        if (value(name) is empty) {
            if (value(type) == "link") {
                ln -s (%RMDBDIR%)/value(src) to (%TASKDIR%)/basename(value(src))
            } else /* value(type) == "copy" (by default) */ {
                cp (%RMDBDIR%)/value(src) to (%TASKDIR%)/basename(value(src))
            }
        } else /* value(name) is specified */ {
            if (dirname(value(name)) is NOT empty) {
                mkdir -p (%TASKDIR%)/dirname(value(name))
            }
            if (value(type) == "link") {
                ln -s (%RMDBDIR%)/value(src) to (%TASKDIR%)/value(name)
            } else /* value(type) == "copy" by default */ {
                cp (%RMDBDIR%)/value(src) to (%TASKDIR%)/value(name)
            }
        }
    }
}
```

}

In the case of a discrepancy between the pseudo-code above and the prose describing this feature, the prose description has precedence.

**Code Examples.....** ..... **294**

## Code Examples

Following are some examples of the *localfile* element in various configurations (assume (%RMDBDIR%)/srcdir is a directory):

```
<runnable name="..." type="task">
  ...
  <!-- create a symbolic link from directory (%TASKDIR%)/srcdir to
      (%RMDBDIR%)/srcdir -->
  <localfile type="link" src="srcdir"/>

  <!-- create a symbolic link from (%TASKDIR%)/fred.do to
      (%RMDBDIR%)/fred.do -->
  <localfile type="link" src="fred.do"/>

  <!-- copy (%RMDBDIR%)/fred.do to (%TASKDIR%)/fred.do -->
  <localfile type="copy" src="fred.do"/>

  <!-- copy (%RMDBDIR%)/src/fred.do to (%TASKDIR%)/fred.do -->
  <localfile type="copy" src="src/fred.do"/>

  <!-- create a symbolic link from (%TASKDIR%)/dofred.tcl to
      (%RMDBDIR%)/fred.do -->
  <localfile type="link" src="fred.do" name="dofred.tcl"/>

  <!-- copy (%RMDBDIR%)/fred.do to (%TASKDIR%)/bob.do -->
  <localfile type="copy" src="fred.do" name="bob.do"/>

  <!-- copy (%RMDBDIR%)/fred.do to (%TASKDIR%)/src/fred.do (creating
      "%TASKDIR%/src" if necessary) -->
  <localfile type="copy" src="fred.do" name="src/fred.do"/>

  <!-- copy (%RMDBDIR%)/srcdir/fred.do to (%TASKDIR%)/src/bob.do
  (creating
      "%TASKDIR%/src" if necessary) -->
  <localfile type="copy" src="srcdir/fred.do" name="src/bob.do"/>

  <!-- copy (%RMDBDIR%)/srcdir/fred.do to (%TASKDIR%)/fred.do -->
  <localfile type="copy" src="srcdir" name="fred.do"/>

  <!-- copy (%RMDBDIR%)/srcdir/fred.do and (%RMDBDIR%)/srcdir/bob.do to
      (%TASKDIR%) (retaining original names) -->
  <localfile type="copy" src="srcdir" name="fred.do bob.do"/>

  <!-- copy (%RMDBDIR%)/srcdir/sv/fred.do to (%TASKDIR%)/sv/fred.do
      (creating "%TASKDIR%/sv" if necessary) -->
  <localfile type="copy" src="srcdir" name="sv/fred.do"/>

  ...
</runnable>
```

Note that these same copy/link operations can be accomplished by adding *cp* or *mv* commands to one of the Action scripts defined in the Runnable. The *localfile* element provides the following advantages: (a) make-like timestamp checks, (b) automatic directory creation, (c) portability, and (d) robust error checking. The value-added capabilities should be worth the additional syntax and minor complexity of specifying the *localfile* element.

## Disposition of Test Collateral

Each Task or Group of Tasks must define what is meant by “test collateral.” That is, what files must be saved in order to do post-analysis even after the working store has been deleted.

By default, nothing is saved outside of the VRMDATA directory tree (including simulation status). Also, by default, nothing within the VRMDATA directory tree is deleted after use.

Variations from this default behavior must be implemented via the user-defined procedures (see “[User-definable Procedures and Loading of Arbitrary TCL Code](#)” on page 412) called during the post-execution analysis phase (see “[Post-execution Analysis](#)” on page 361) of each Action executed.

There are a number of possible “disposition policies” that can be implemented. One might be “delete on success.” This policy helps minimize disk utilization by deleting the working store for any tests that pass (presumably a passing test does not have to be analyzed after the fact, other than for coverage that would normally be saved off as part of the collateral or merged with other test coverage at the Group level). The disposition policy implemented by the user-defined procedures can also be controlled by parameters defined in the RMDB database itself.

## File Safety

When a file or directory is created or written to by *vrun*, a check is made to ensure that the file falls into one of two subtrees on the file system as follows:

- The directory from which the *vrun* instance is launched (that is, *vrun \$cwd*).
- The VRMDATA directory specified by the *-vrldata* command-line option (./VRMDATA by default).

This ensures that an incorrect RMDB file cannot result in damage to source files and/or the operating system (the latter of which is not possible when using a secure operating system). The exception to this rule is the *-vrldata* option itself, that defines the location of the VRMDATA directory tree. The VRMDATA directory can be placed anywhere the user has write access.

## Script Execution and Timeouts

---

Once the RMDB database has been read and the requested Runnables have been expanded to form the execution graph, a completion event is posted to the special `=init=` Action. This causes one or more Actions to become eligible for execution (that is, their dependencies having been completed). The Actions directly dependent on the `=init=` Action are generally the *preScript* Actions of the top-most Group(s) selected via the command line. Actions that are eligible for execution are added to a run queue. An inner loop periodically removes Actions from this queue and executes them. A list of currently executing Actions is also maintained, in which the expected completion time (execution time plus the minimum timeout value for that Action) is recorded. As each Action reports completion, it is removed from the currently executing list and from the graph. The vertices of the graph beginning at the completed node are followed to determine what Actions, if any, can be eligible to run next. If a vertex from the completed Action happens to be the last such vertex into the node at the opposite end, that opposite-end node also becomes eligible to run and is added to the eligible Action run queue.

VRM continues to execute its inner loop as long as at least one Action remains in either the eligible queue or the currently running list. When all the Actions in the eligible queue have been launched, VRM scans the currently running list to see if any launched jobs have passed their timeout limit. If a job times out, an error is emitted and a completion event is posted for the Action associated with the timed-out job. A timeout is considered a failure.

Once there are no further Actions in the currently running list and no more eligible Actions in the eligible queue, the run is declared complete. If any nodes still exist in the run graph, they are squawked to the output with an error message (this should only happen if nodes were placed in the graph and not properly connected to other nodes or if the graph following algorithm is faulty). It is also possible that some nodes would remain in the graph and be squawked in the event the regression run is terminated prematurely due to errors. In this case, the graph node dump serves to notify the user that some Actions were never executed.

Action and Wrapper Script Generation .....	297
Using Existing Scripts with VRM.....	302
Script Return Values .....	306
Pipe-based Command Launching .....	309
Sending User-defined Messages from an Action Script.....	310
Spoofing Default Behavior of Action Scripts .....	311
Timeouts .....	311

## Action and Wrapper Script Generation

Each Action (*preScript*, *execScript*, or *postScript*) is defined either in terms of a sequence of one or more commands stored in the RMDB database or as a pointer to an existing script file (said pointer also being stored in the RMDB database).

The sequence of commands related to a given Action are copied to a script file of the same name (*preScript*, *execScript*, or *postScript*) with a *.do* extension, located within the working directory associated with the Action. This file is marked as being executable. As the commands are stored, any parameter references contained therein are expanded (see “[Parameters and Parameter Expansion](#)” on page 263 for details).

A TCL wrapper script is then constructed in the same working directory to execute this Action script. The purpose of this wrapper is to handle the details of the Action that are related to internal VRM algorithms (steps that would not normally be handled by the user script). For example, since the wrapper is normally executed from the working directory associated with that Action, the first order of business is to change directories to that working directory and to set up the transcript log. After the user script completes, the wrapper also notifies the master VRM instance that the Action has completed.

The exact form of this wrapper depends on how the user script is to be launched. Each Action has an optional *launch* attribute attached to the element corresponding to that Action in the RMDB database. If the commands represent TCL commands to be executed within a *vsim* shell, the *launch* attribute should be set to *vsim*. If the commands represent shell commands to be executed by the default command shell (for example, */bin/sh* on most POSIX-compliant systems), the *launch* attribute should be set to *exec*. Alternately, if the *launch* attribute is set to anything other than *vsim* or *exec*, the value of the *launch* attribute is treated as the name of the shell program under which the commands are to be interpreted. By default (that is, if no *launch* attribute or set or if the value is the empty string), the commands will be treated as TCL commands to be executed under a *vsim* shell.

The result is that the commands contained within the *preScript*, *execScript*, or *postScript* Actions are treated as a series of *vsim* commands by default, but the user can elect to treat them as shell commands to be executed (by setting *launch* to *exec*) or can pass them to some other command interpreter (for example, Perl, Ruby, and so on) by setting the *launch* attribute to the absolute path name of the external interpreter program.

Note that only the default (TCL under *vsim*) is considered portable across all Questa-supported platforms. For example, the TCL *exec* command expects Bourne shell commands on Unix/Linux platforms, but MS-DOS *.bat* file commands on Windows. The construction and interpretation of operating system commands can differ across platforms. In order to ensure maximum portability, it is recommended that *vsim* commands be used for all Action scripts.

On platforms where script execution is carried out by the Bourne shell (that is, Unix/Linux), the first command defined in an Action may be a *shebang* line (a line beginning with the literal characters #, !, and a path to an interpreter executable). When executed, this informs the Bourne

shell to pass the script to some other interpreter (for example, Perl, csh) for execution. This *shebang* technique does not work on the Windows platform.

Regardless of the type of wrapper used, the appropriate sequence of commands is stored in a file named for the Action in question, but with a *.wrap* suffix (for example, *execScript.wrap* for the *execScript* Action). Unless an alternate execution method is specified (described in “[Execution Methods](#)” on page 315), this wrapper is passed to either *vsim* or *vish* for execution.

In order to avoid potential collision with scripts, logs, and other files generated by VRM, the user is advised to avoid writing to files whose base name consists of *preScript*, *execScript*, or *postScript* regardless of extension.

<b>vsim Mode Wrapper .....</b>	<b>298</b>
<b>exec Mode Wrapper.....</b>	<b>300</b>

## **vsim Mode Wrapper**

The command sequence used in the wrapper for *vsim* launched commands is shown below.

```

namespace eval runmgr {
    set rm_gone 0
    set rm_stat 0
    set rm_mesg {}
    proc rm_status {stat {mesg {}}} {
        variable rm_stat
        variable rm_mesg
        set rm_stat $stat
        set rm_mesg $mesg
        return
    }
    proc rm_message {mesg {stat {0}}} {
        vrun -dest <port@host> -send user -code $stat -message $mesg
            <actionName>
    }
    proc rm_done {} {
        variable rm_gone
        variable rm_stat
        variable rm_mesg
        if {$rm_gone} {
            puts stderr "Blocking bogus 'done' (<actionName>)"
        } else {
            vrun -dest <port@host> -send done -code $rm_stat -message
                $rm_mesg <actionName>
        }
        set rm_gone 1
    }
}
rename quit runmgr::rm_quit
proc quit {args} {
    if {[lindex $args 0] ne {-sim}} {
        transcript file {}
        runmgr::rm_quit -sim
        cd <vrunDirectory>
        runmgr::rm_done
    }
    eval runmgr::rm_quit $args
    puts stderr "Blocking bogus 'done' (<actionName>)"
}
transcript on
onbreak {resume}
onerror {runmgr::rm_status 1 {Script error}}
quietly catch {
    cd <workingDirectory>
    vrun -dest <port@host> -send start <actionName>
    transcript file <script>.log
    do <script>.do
}
quit -f

```

In this case, the value of *actionName* is the full calling path to the Action (used to identify the specific Action instance in the graph), the value of *workingDirectory* is the absolute path to the working directory associated with this Action, and the value of *script* is the name of the Action (*preScript*, *execScript*, or *postScript*). Redirecting the *quit* command ensures that, except for catastrophic failure of the *vsim* shell, the “done” event is always passed to the master VRM

process (the value of *port@host* is set to the TCP/IP port and hostname on which a socket has been opened by the master VRM process to receive event communications).

When a *quit* command is received, the wrapper first quits any simulation that might be running and changes directories back to the directory from which VRM was launched (indicated by *vrunDirectory* in the script example). The reason for this is because VRM might need to delete the working directory for the Action under certain conditions. That cannot be done if the simulation is still running or if the VRM wrapper script process is still active in the working directory.

If, during the course of execution, the user script calls the *runmgr::rm\_status* procedure, the integer status value and optional message string are saved off to be returned when the script completes. In this way, errors and/or warnings occurring within the scripts can be conveyed to the invoking VRM to be analyzed and handled. For more details see “[Script Return Values](#)” on page 306.

## exec Mode Wrapper

The command sequence used in the wrapper for non-*vsim* (shell) commands is shown below.

```
package require RunManager
namespace eval runmgr {
    set rm_stat 0
    proc rm_status {stat} {
        variable rm_stat
        set rm_stat $stat
        return
    }
    proc rm_done {} {
        variable rm_stat
        RunManager::Main -dest <port@host> -send done -code $rm_stat
            <actionName>
    }
}
rename exit runmgr::rm_exit
proc exit {args} {
    cd <vrunDirectory>
    runmgr::rm_done
    runmgr::rm_exit $args
}
cd <workingDirectory>
set ::env(MTI_VRUN_SEND) {vrun -dest <port@host> -send user <actionName>}
if {[catch {<execCommand> >& <script>.log}] } {
    if {[lindex $errorCode 0] eq {CHILDSTATUS}} {
        runmgr::rm_status [lindex $errorCode 2]
    } else {
        runmgr::rm_status 1
    }
}
exit 0
```

In this case, the value of *actionName*, *workingDirectory*, *scriptName*, and *port@host* are the same as in the *vsim* case. The value of *execCommand* is either *exec <script>* or *exec <launch> <script>* depending on whether the value of the *launch* attribute for this Action is *exec* or some other string (presumably the name of the interpreter to be used to interpret the script). The *RunManager::Main* call is essentially the same as calling *vrun* from within *vsim* except that this script is passed off to *vish* instead of *vsim* (to save time and memory) and the *vrun* command is not known to *vish*. The TCL *catch* command ensures that errors occurring while launching the script do not cause the *vish* wrapper to abort without sending the “done” message.

Like the *vsim* based wrapper, this wrapper changes directories upon completion to the directory in which VRM itself is running in order to allow the working directory to be deleted upon completion. Since the script is running in a *vish* shell, any simulation that was run would have been terminated before control returns to the wrapper. However, caution should be exercised in not putting a simulation into the background while running under VRM. The reason is that VRM assumes all activity associated with a given Action has ceased by the time the “done” message for that Action is received.

If any command in the Action script fails with a non-zero status or if the Action script itself exits with a non-zero status, then the integer status value is passed to the invoking VRM with the “done” message. Unfortunately, TCL does not have access to the exact error message that caused the non-zero status so the message string passed to the invoking VRM is always blank.

## Using Existing Scripts with VRM

The Action scripts (*preScript*, *execScript*, and *postScript*) are parsed for parameter expansion and copied to the working directory. This includes scripts defined in the RMDB database and scripts pointed to by the file attribute of the script element. If pre-existing (user-defined) scripts are to be used to drive the behavior of the regression, then there are the following three ways in which the parameterizable information can be passed:

- Edit the pre-existing scripts to include VRM parameter references in place of hard-coded data values.
- Pass the parameterized values as positional arguments on a command line that invokes the pre-existing script.
- Set environment variables based on the parameterized values and access those environment variables from the script.

Also, another brute-force alternative is to create a unique script for every Action containing hard-coded values (in which case, there would be little value added by using VRM). Assuming, however, that you wish to parameterize the pre-existing scripts, the following subsections are examples of the latter two methods for passing data values.

<b>Passing Values Via Position-dependent Parameters .....</b>	<b>302</b>
<b>Passing Values Via Environment Variables .....</b>	<b>303</b>
<b>Copying the Script File to the Working Directory .....</b>	<b>303</b>
<b>Running Scripts in Existing Directory Structure .....</b>	<b>304</b>

## Passing Values Via Position-dependent Parameters

In this case, a single command is used in the Action script. This command refers to the pre-existing script and passes the appropriate arguments on the command line. If the pre-existing script (*myscript.csh*) looks something like the following:

```
#!/bin/csh -f
#
# Usage: myscript.csh <top-module> <seed> <ucdb-filename>
#
vsim -lib ./work -c $1 -sv_seed $2 -do "run -all; coverage save $3;
    quit -f"
```

then the corresponding *execScript* definition looks something like the following:

```
<execScript launch="exec">
<command>(%VRUNPATH%)/scripts/myscript.csh (%top%) (%seed%)
    (%ucdbname%)</command>
</execScript>
```

The single-line *execScript* command is executed from *vish* and it immediately transfers control to the pre-existing script, passing parameterized values as required. For Windows batch files, the *execScript* can be launched from *vsim* using the TCL *exec* command that should have the same effect:

```
<execScript launch="vsim">
    <command>exec "($VRUNPATH%)/scripts/myscript.bat (%top%) (%seed%)
    (%ucdbname%) "</command>
</execScript>
```

## Passing Values Via Environment Variables

In this case, a mini-wrapper script is defined as the Action script. This mini-wrapper sets a series of environment variables based on the relevant parameterized values and those variables are accessed from within the pre-existing script. If the pre-existing script (*myscript.csh*) looks something like the following:

```
#!/bin/csh -f
vsim -lib ..\work -c $TOP -sv_seed $SEED -do "run -all; coverage save
$UCDB; quit -f"
```

then the corresponding *execScript* definition looks something like the following:

```
<execScript launch="/bin/csh">
    <command>setenv TOP (%top%)</command>
    <command>setenv SEED (%seed%)</command>
    <command>setenv UCDB (%ucdbname%)</command>
    <command>($VRUNPATH%)/scripts/myscript.csh</command>
</execScript>
```

Obviously, the exact commands used in this mini-wrapper script will vary depending on the command shell selected.

## Copying the Script File to the Working Directory

The two examples above assume that the pre-existing script exists in a subdirectory of the directory from which *vrun* was launched. The pre-existing script is executed from that location by the Action script. However, one of the key reasons for copying the parameter-expanded Action script into the working directory is so the tests can be rerun manually under the same conditions as those under which they were run by VRM. If the pre-existing script is not present in the working directory, then some of that advantage is lost (or, at the very least, the user needs to make sure the pre-existing scripts remained accessible from their original paths).

One option the user has to circumvent this issue, and more clearly document exactly how an Action is executed, is to copy the pre-existing script into the working directory and run it from there. This is simply a matter of adding the relevant commands to the Action script definition. In

In the case of the position-dependent example above, the Action script could be rewritten as follows:

```
<execScript launch="exec">
  <command>if [ ! -e myscript.csh ]; then</command>
  <command> cp -p (%VRUNPATH%)/scripts/myscript.csh
            myscript.csh</command>
  <command>fi</command>
  <command>myscript.csh (%top%) (%dofilename%) (%ucdbname%)</command>
```

The conditional test is an optimization to make sure the copy is only done in the case of a **new** working directory (one that is lacking in the necessary script file).

Note that the user can also use a *localfile* element in the Runnable associated with the Action to copy the pre-existing script into the working directory. See “[Local File Generation](#)” on page 289 for details.

## Running Scripts in Existing Directory Structure

In some cases, there can already be an existing directory structure within which existing user scripts are executed, assuming they can find various HDL source and setup files at fixed locations within said directory structure. In order for these sort of scripts to run as is under VRM, executing from a specific directory within the test directory tree, a *cd* command (pointing to the directory in which the script is to be run) can be added to each Action script. This causes the script to run subsequent commands (that is, those following the *cd* command itself) in the directory in which the script is launched without VRM. In order to prevent file collisions between concurrent simulations, the sequential flag on any Group Runnables accessing the fixed directory structure should be set to “yes”, meaning VRM runs each Action script one-by-one rather than concurrently.

In order to enable concurrency, the following steps should be taken:

- If multiple simulations are to be run in the same directory at the same time, then the user script(s) may need to be modified to give explicit non-overlapping names to the various files generated by each simulation (such as WLF files, UCDB files, transcript files, or log files, and so on).
- If the simulations are to be run in separate working directories, then the *cd* command should be removed and the relative file paths in the Action script(s) should be changed to allow the tools invoked to locate the necessary input files.

For example, assume the path to the original (pre-VRM) working directory is saved in the RMDB database file as the parameter *runcwd* for convenience. All relative file names used in the *vsim* command or in any *do* script should be prefaced with a parameter reference to either *VRUNDIR* (directory from which *vrun* was launched) or *runcwd* (directory in which the test would have been launched prior to VRM). In addition, a *-lib* option should be added to the *vsim* command line to locate the work library.

For example, if the following *execScript* Action script is auto-generated:

```
<execScript>
  <command>cd (%runcwd%)</command>
  <command>vsim -do {set test test2; do runtest.do} -c -sv_seed 2 -vopt
    top</command>
</execScript>
```

The modified script might look something like the following:

```
<execScript>
  <command>vsim -lib (%runcwd%)/work -do {set test test2; do
    (%runcwd%)/runtest.do} -c -sv_seed 2 -vopt top</command>
</execScript>
```

This command points to the common work library in the original simulation directory and to the *do* script in that same directory. Because the simulation runs in its own working directory under the *VRMDATA* directory tree, the *cd* command is no longer needed. Any result checking or coverage merging scripts might need to be modified to look for the WLF/UCDB files generated by the simulation in their new location (under the *VRMDATA* directory tree). Finally, the value of the *sequential* attribute on the top-level Runnable may be changed to “no” to enable concurrent execution of all the tests.

If the *vsim* command calls out a *do \_dofile\_* command in its *do* command-line option, the contents of the *dofile* may also have to be modified to account for the difference in directories when the *dofile* itself is executed. It might be easier to copy the commands from the *dofile* directly into the *execScript* element where they can also make use of parameter references or to use a *localfile* element to generate the *dofile* in the working directory.

## Script Return Values

Each Action script has a return value that indicates, at a coarse level, whether the commands contained in the script encountered any errors. This return value is communicated to the launching *vrun* instance for use in pass/fail analysis. This function can require some cooperation from the (user's) Action script in order to correctly communicate the status of commands contained in the Action script.

In the case of a *vsim* mode Action script, the *onerror* command can be used to trap errors that occur while the script is running. The wrapper itself uses *onerror* to trap errors that can occur in the process of launching the Action script. For example,

```
onerror {runmgr::rm_status 1 {Script error}}
```

This *onerror* handler takes advantage of a procedure defined within the wrapper as follows:

```
proc rm_status {stat {mesg {}}} {
    variable rm_stat
    variable rm_mesg
    set rm_stat $stat
    set rm_mesg $mesg
    return
}
```

The *rm\_stat* variable defaults to zero (0) and the *rm\_mesg* variable defaults to an empty string. These two values are passed to the invoking VRM instance via the “done” message just before the wrapper script terminates. If the user script calls the *runmgr::rm\_status* function for any reason, it can supply its own status value and, if needed, an optional message that *vrun* adds to its own error message to log the failure. If this function is called multiple times from a single Action script, only the last set of values are passed, as there is only ever one “done” message sent from any one Action script wrapper to the invoking VRM.

In the case of an *exec* mode Action script, there is only a status (no optional message). If the Action script returns a non-zero value (that is, from an *exit <N>* command), that status is sent to the invoking *vrun* instance with the “done” message. If the script failed but, for some reason, the exit status is not available, then a status of one (1) is returned. If the Action script runs successfully (that is, the return status is zero and no script execution errors have occurred), then a status of zero (0) is returned.

When the “done” message is received by the invoking *vrun* instance, a zero status is interpreted as success. If a message is returned by the Action wrapper script along with a zero status, a warning message containing that message is emitted to the VRM log output. If the return status is non-zero, then the Action is considered to have failed. An error message is emitted to the VRM log file, including the optional message, if specified, and a “fail” status is passed to the pass/fail analysis stage.

Note that for Actions that produce a UCDB file (simulation runs), the pass/fail analysis is a bit more complex. The script return value, however, is still taken into consideration in determining

the success (or lack thereof) of the Action in question. See “[Post-execution Analysis](#)” on page 361 for details on how *vrun* uses the returned status value.

<b>Modifying vsim Mode User Scripts to Return Status.....</b>	307
<b>Modifying exec Mode User Scripts to Return Status .....</b>	308
<b>Execution Mode Simulations Hang .....</b>	308

## Modifying vsim Mode User Scripts to Return Status

A *vsim* mode script is executed via the *do* command in *vsim*, which does not return status. When an error occurs in a *vsim* TCL script, the interpreter executes the optional *onerror* handler. In order to trap errors and use them to modify the return status of the Action, the user must make use of the *onerror* handler. The wrapper script associated with the Action provides a procedure that can be called to set the return status and optional message string. The following example shows a *preScript* with the *onerror* handler set:

```
<execScript>
    <command>onerror {runmgr::rm_status 1 "Compile error"}</command>
    <command>vlib work</command>
    <command>vlog -sv (%RMDBDIR%)/src/top.sv</command>
</execScript>
```

In cases where the Action script needs to signal a failure when no TCL “error” has occurred (such as a non-matching comparison to expected results), the Action script can call the *runmgr::rm\_status* procedure manually. Since this procedure only records the status and does not force an exit, the Action script should invoke the *quit* command at the appropriate point in the processing as follows:

```
<execScript>
    <command>...some command...</command>
    <command>if {![string equal $results $golden]} {
        runmgr::rm_status 1 "Simulation results differ"; quit -f
    }</command>
</execScript>
```

The *runmgr* namespace is used to minimize the chances that the VRM wrapper implementation and the user script will experience name collisions. The *runmgr::rm\_status* procedure takes a mandatory integer status and an optional message string. These values are stored until the script returns. Only one return value and one message are returned to the invoking VRM per Action. It can also be possible to use the TCL *catch* command to provide a unique status value for each possible error in a script. In the event of multiple errors, the data provided in the last call to *runmgr::rm\_status* is the data sent to the invoking *vrun* process. In order to pass information on the first error instead of the last, the *onerror* handler should abort the Action script immediately after the error by calling *quit -f* as shown in the example above.

## Modifying exec Mode User Scripts to Return Status

An *exec* mode Action script is executed from the wrapper script for the associated Action via the *exec* command in *vish* (TCL). When a command in the Action script returns a non-zero status, or when the Action script itself exits with a non-zero status, that status is made available to the TCL *exec* command and is returned to the wrapper script and communicated to the invoking *vrun* process via the “done” event. Provided all the commands in the script return POSIX-like status values, there is no need to alter the Action script. However, if the user wants the script to exit at the first error, or on some error from a 3rd-party utility that does not return a non-zero status, the *exit* command can be used (assuming Borne-shell or C-shell; other scripting languages can have a different exit command). The following example exits with an error if the contents of a log file do not match a file containing the expected results for the simulation:

```
<execScript launch="exec">
  <command>#!/bin/csh -f</command>
  <command>...some simulation command(s)...</command>
  <command>diff results.log results.golden &gt; results.diff</command>
  <command>if (!(-z results.diff)) then</command>
    <command>  exit 93</command>
  <command>endif</command>
</execScript>
```

Note the *&gt;* used in the *diff* command instead of the redirection (*>*) character. This is because angle brackets (*<>*) and the ampersand (*&*) are special characters in the XML format and will be mistaken for XML markup if used in an Action script element.

Also note that most command shells provide only an integer status, with non-zero meaning simply “failure”. The TCL *exec* command does not provide easy access to the exact error message that caused a non-zero status return so *exec* mode Actions always return an empty string in the message string field of the “done” message. Messages produced by the commands in the Action script can be found in the log file for that Action (the path to which is included in the error message from *vrun*).

## Execution Mode Simulations Hang

One possible reason for VRM hanging is an Action script that falls into the *vsim* command-line prompt after completion. Action scripts launched in “exec-mode” (the *launch* attribute is set to something other than *vsim*) are especially susceptible because the *-do* option in *vsim* does not exit the application when all of the specified commands have been executed. Be sure to put a *quit -f* command at the end of the *-do* command list. For example,

```
vsim -c ... -do '...; quit -f'
```

This should prevent the shell-invoked *vsim* command from dropping into an interactive prompt. Also, do not forget to include the *-c* (non-GUI mode) option.

## Pipe-based Command Launching

By default, VRM launches scripts as background processes. Their only means of communicating errors back to the launching *vrun* process is the TCP/IP port on which the *vrun* process is listening. If the process simply fails to launch (for example, if the *vsim* or *vish* processes cannot be found), VRM will detect a launch error and treat the corresponding Action as it would any other failed job. If, however, the wrapper command launches as expected but fails to execute the wrapper script (for example, in the case of an error originating in a faulty method command), VRM could miss the error. In this case, the only recourse is a *timeoutQ* violation (that is, the timeout window between launching the Action and receiving the “start” message from the wrapper). By default, this delay is 31 seconds. It must be made longer in the case of a grid-based method command. After a timeout, the offending error condition will most likely be reported in the *(xxx)Script.stderr* file that is also echoed to the *vrun* log output.

There is an alternative launching algorithm available. If the *-pipelaunch* command-line option is specified or the *MTI\_VRUN\_PIPE\_LAUNCH* environment variable is set, then VRM will use pipes to launch the child processes rather than background mode. The child process runs in the background either way, but when pipes are used, the *vrun* process is able to verify the status of the launched job even before the wrapper takes control and sends the “start” message. This can be useful for debugging timeouts that occur due to various setup or system errors. Following are the two downsides to this algorithm:

- It requires one additional open file descriptor per running Action to monitor status.
- The handles for the *(xxx)Script.stdout* files for each running Action are owned by the *vrun* process instead of by the wrapper script process.

Since most operating systems have both fixed global and user-setable per-process limits on the number of available file descriptors, the pipe-based launch algorithm is expected to result in a lower capacity with respect to concurrent Actions launched by VRM. For this reason, the pipe-based algorithm is an option that must be enabled. When enabled, the *-j* option should also be used to ensure that the total number of file descriptors used does not exceed the limits set by the operating system.

Note that in the case of a *method* command that queues Actions to a compute grid, the grid-submission command itself is short-lived. It generally registers the job with the grid management software and then exits. In this case, the launch pipe is only connected to the grid-submission command. As the grid-submission commands exit, the file descriptors are released, even before the grid job has actually executed. While that means pipe-based launching can be used without a significant reduction in capacity, it also means that the grid system itself opens a window of time in which an Action can fail to run (that is, an error detected by the grid subsystem after the job has been submitted), but the error is not detected by the launching *vrun* process. In this case, a non-zero *timeoutQ* value ensures that the Action eventually times out and VRM does not appear to hang as a result.

## Sending User-defined Messages from an Action Script

On occasion a user's script needs to emit an error, warning, or informational message back to the executing *vrun* application for inclusion in the log output for that run. There are two ways to do this as follows:

- Include a message string with a zero-status return value.
- Send a user-defined message to the socket on which the executing *vrun* application is listening.

When the wrapper script returns the “done” message from a non-failing Action script, the status code is set to zero and the message string is generally empty. If the message string for a zero status “done” message is **not** empty, then the *vrun* application receiving said message will interpret the message string as a warning message. The message is emitted to the *vrun* log output as a warning.

For a *vsim* mode Action script, the user script should call the *runmgr::rm\_status* procedure passing a zero status and the warning string as follows:

```
runmgr::rm_status 0 "Compile warnings occurred -- check the logs"
```

Since a *vish* mode Action script (that is, a script containing commands to be interpreted by a command shell) returns only a numeric status, a warning message cannot be returned from these scripts. The alternative is to return an error or to send a user-defined message.

A user-defined message is sent by calling *vrun* in message sending mode with the following command-line arguments:

```
vrun -send user -dest <port@host> -message "This is my message..."  
<actionName>
```

In this case the *-code* value is not used and, therefore, does not have to be supplied. The *port@host* value is the TCP/IP port and hostname on which the launching *vrun* application has opened a port to listen for event messages from the running Action scripts. The message is reported as an informational message on the log output of the *vrun* process that receives the message. The *actionName* allows *vrun* to identify which Action has emitted the message.

For a *vsim* mode Action script, the user script should call the *runmgr::rm\_message* procedure, passing the message string and an optional status code as follows:

```
runmgr::rm_message "This is my message..."
```

Note that while a status code can be passed as the second argument to the *runmgr::rm\_message* procedure, the status code is not used.

For a *vish* mode Action script, an environment variable (*MTI\_VRUN\_SEND*) is defined with the pertinent options for the *vrun* command. All the user needs to do is “call” the *vrun* application via this environment variable, passing the *-message* command-line option with the message to be transmitted as follows:

```
$MTI_VRUN_SEND -message "This is my message..."
```

In both cases, the wrapper script for the Action takes care of setting the *<port@host>* value to point to the launching *vrun* application's event communication socket. The message is sent immediately and the listening *vrun* process emits the message to the log output asynchronously at the time it is received. Note that since *vrun* can launch any number of Action scripts concurrently, there is no guarantee as to the order in which messages emitted from multiple Action scripts will occur in the log output. All log output is emitted by *vrun* as received and the user-defined message can well be interspersed with other *vrun* log messages.

## Spoofing Default Behavior of Action Scripts

Use the role attribute of an Action script to mimic the role of another element type.

There are three Action behaviors that another Action can inherit, or spoof:

- Named queue selection — an execScript, whose role is set to mergeScript, will be placed in the “\_merge” named queue by default.
- Timeout computation — an Action, whose role is set to mergeScript or triageScript, will have an execution timeout value computed based on the number of input UCDBs in the operation. Refer to the section “[Timeouts](#)” for more information.

For an action, whose role is mergeScript, you should set the mergelist parameter to the path to the file that contains the list of input UCDBs.

For an action, whose role is triageScript, you should set the triagelist parameter to the path to the file that contains the list of input UCDBs.

- Local re-run behavior — an Action, whose role is set to mergeScript or triageScript, will be re-launched if they fail during execution for any reason (up to 10 attempts). You can disable this behavior with the *-nomergererun* option to *vrun*.

## Timeouts

When an Action is launched, VRM records the time by which the Action is expected to reach the next phase of execution. A background loop within *vrun* periodically checks for Actions that have not reached their next milestone in time. In this case, a “timeout” is declared and the Action is considered to have failed.

There are two defined timeout periods as follows:

- The first period starts when the Action (or its execution method script) is first launched and ends when a “start” message is received from the wrapper script associated with the Action.
- The second period starts when said “start” message is received and ends when the “done” message is received from the wrapper script.

These two timeouts are independent and can be given different values.

The first timeout, called  $tQ$  places a limit on the amount of time an Action can remain queued without actually starting. It is only relevant when the Action has been queued to a grid machine and it guards against possible loss-of-job issues in the compute grid. The *mintimeout* attribute on *method* elements controls  $tQ$ , since the *method* element also controls whether the Action is queued to a grid at all and, if so, which one and at what priority.

The second timeout, called  $tX$  places a limit on the amount of time an Action can take to execute. Since the “start” and “done” messages bracket the user’s script for that particular Action, it is essentially a timeout on the execution time of the user’s script as a whole. The *mintimeout* attribute on Action script elements (*preScript*, *execScript*, and *postScript*) controls  $tX$ , since the Action script element also controls the contents of the user script associated with the Action.

The attributes are called “minimum” timeout values because the value actually used is at least as long as either the global default value for the timeout in question or the per-element value provided by the element associated with the timeout in question (which means we must pick the **maximum** of the two values in order to ensure the timeout is **at minimum** at least as long as either supplied value).

The default global timeout value for  $tQ$  is 60 seconds and for  $tX$  it is 300 seconds. These defaults can be changed via an environment variable or a command-line option. The format is the same for both so it is covered in the paragraphs to follow. If the *MTI\_VRUN\_MIN\_TIMEOUT* environment variable is defined and meets the specified format, the value(s) specified by this variable will override one or both of the built-in global timeout values. If the *-mintimeout* command-line option is specified and the value passed to that option meets the specified format, the value(s) specified by the *-mintimeout* option will override one or both of the global timeout values, whether they came from the built-in defaults or the environment variable. All timeout values are expressed in seconds. Note that the *-mintimeout* option supports embedded parameter references.

The general format of either the environment variable or the command-line option is  $tQ:tX$  with either value being optional. If two numeric values are specified with the intervening colon, each is used to modify the corresponding global timeout value. If either value is non-numeric, that value is ignored irrespective of the other value. That is, the command-line option *=mintimeoutFRED:30* changes the global value for  $tX$  to 30 while  $tQ$  remains unchanged (since the  $tQ$  part of the option argument is not numeric and, thus, does not pass the test).

Note that no error message is emitted if a non-numeric timeout value is detected. However, if the *-debug* option is specified, a message verifying that the global default timeout has changed is emitted. The debug message refers to *QueTimeout* and *ExeTimeout*, the *vrun* internal names for the two values.

If either the environment variable or the command-line option contain a single numeric value and no colon, by default these values are used to modify *tX* (the execution timeout). If, however, there is also a colon specified, the timeout value affected depends on the placement of the colon, for example, *:tX* will modify *tX*; while *tQ:* will modify *tQ*. Using this syntax, it is entirely legal for the environment variable to modify only *tQ* and the command-line option to modify only *tX* (or vice-versa), since the two values are completely independent.

Both the environment variable and the command-line option can modify either of the global minimum timeout values either up (longer) or down (shorter). A timeout value of zero disables the timeout (as would be the case for the built-in default value of *tQ*). These global timeout values are global (as their name implies). They apply to every Action launched and cannot be changed once *vrun* begins launching Actions.

Per-Action minimum timeout values can be specified in the RMDB file. An Action script can specify an optional minimum value for *tX* via its *mintimeout* attribute. A *method* element can specify an optional minimum value for *tQ* via its *mintimeout* attribute. The default value for both *mintimeout* attributes is zero (which, because the values are computed as minimum timeout periods, would basically mean: 'use the global default value here'). For each Action launched, the execution timeout is determined by taking the maximum of the global *tX* value and the value of the *mintimeout* attribute on the Action script. If an execution method is found for the Action, the queuing timeout is determined by taking the maximum of the global *tQ* value and the value of the *mintimeout* attribute of the Action script. If no execution method is found for the Action, the global default value of *tQ* is used.

---

**Note**

 The queuing timeout for *mergeScript* or *triageScript* Actions launched via an execution method is computed by taking the number of UCDBs to be merged or triaged, multiplying by the value of *inctimeout*, and then adding the value of *mintimeout*.

---

If the resulting value for either minimum timeout is zero, *vrun* will not monitor the Action for timeout. Instead it will potentially wait forever for the Action to change states, or at least until someone sends a “stop” message to the launching *vrun instance*. In addition, if the *-notimeout* command-line option is given, then timeout behavior is disabled globally for the entire regression run. This can be useful if VRM is used to launch an interactive session that can be open for a very long time.

Timeouts are measured in wall-clock time elapsed since the start of the timeout period in question. When the specified time has elapsed, an error message is posted to the *vrun* log output and the Action is reported as “failed” in the event log.

## Auto-Timeout Calculation

For large testcases with long runtimes, anywhere from 10 minutes, 10 hours or longer, the auto timeout feature will analyze the elapsed time of passing actions to estimate the appropriate timeout value.

- Enable the auto-timeout calculation by adding the `-autotimeout` option to the `vrun` command line.
- Control the sensitivity of the calculation with the `timeoutmargin` attribute of any Action script, where the value is a percentage of the maximum execution time added for computation.
- Change the default margin for the calculation with the `-timeoutmargin` option to the `vrun` command.

## Execution Methods

By default, the wrapper associated with each Action launched by VRM is executed by either *vsim* or *vish* as a child process of the initiating VRM process. The exact command used to launch this process depends on the value of the *launch* attribute of the element associated with the Action. For Actions whose *launch* attribute is set to *vsim* (or which do not have a launch attribute, since *vsim* is the default launch type), the following command is used:

```
echo "do $script.do"
| $MODEL_TECH/vsim -c -l "" -onfinish stop
```

For Actions whose *launch* attribute is set to *exec* (or any non-empty value other than *vsim*), the following command is used:

```
$MODEL_TECH/vish $script.do
-c
```

Note that the *MODEL\_TECH* environment variable is maintained internally and must not be altered by the user.

If the *launch* attribute is set to anything other than *vsim* or *exec*, the value of the *launch* attribute is assumed to be the name of and/or path to an interpreter shell that can execute the commands defined in the Action script. In this case, the *exec* command is used and the appropriate interpreter is launched from within the wrapper script using the TCL *exec* command.

Any output from this process on *stdout* or *stderr* is redirected to a pair of files in the working directory associated with the Action (the user script output should be written to *\$script.log* due to the redirection specified in the wrapper file). By default, the wrapper script is launched in background mode to prevent VRM from hanging as the result of a hung user script. If the *MTI\_VRUN\_NOAUTOBACKGROUND* environment variable is set, however, the wrapper scripts is launched in foreground mode (in which case, only one Action can be running at any one time).

The *-onfinishstop* option to a *vsim* simulation prevents designs that call *\$finish* from exiting prior to sending the “done” message to the invoking VRM process. If the user script changes this setting to cause *vsim* to exit immediately, then the simulation Action exits without sending the “done” message and appears to timeout from the point of view of the VRM. Do not do this.

Note that the “pipe” construct in the *vish* launched case is necessary to prevent a nested *-do* option error while preserving the ability of the wrapper to set *onbreak* and *onerror* behavior. The user has no direct control over this behavior.

As an alternative, VRM supports user-defined execution methods. What an execution method does is allow the user to control, via the RMDB database, the actual execution command issued

for each Action. A user-defined execution method is defined by a *method* element in the RMDB database, which might look something like the following:

```
<method name="grid">
    <command>qsub -V -cwd -b yes (%WRAPPER%) </command>
</method>
```

When this method is selected (see [Selecting an Execution Method](#) below), the appropriate *vsim* or *vish* wrapper script command given above is written into a shell script file and that file name is substituted into the command as the value of the (%WRAPPER%) parameter. The shell script name is the name of the Action being launched but with a .bat extension (for example, *execScript.bat* for the *execScript* Action). In the VRM implementation, only the first *command* element listed in the *method* element is considered. The command string undergoes the same parameter expansion as any other command found in the RMDB database (see “[Parameters and Parameter Expansion](#)” on page 263 for details). The resulting command is then used in place of the wrapper launch command given above. VRM uses the TCL *exec* command to launch the method command so the command must be in a form that is considered executable on the platform on which VRM is executing.

<b>Selecting an Execution Method</b> .....	<b>316</b>
<b>Methods and Timeouts</b> .....	<b>317</b>
<b>Conditional Execution Methods</b> .....	<b>318</b>
<b>Methods and Base Inheritance</b> .....	<b>321</b>
<b>Execution Method Examples</b> .....	<b>327</b>

## Selecting an Execution Method

Execution method (*method*) elements can appear within any Runnable (that is, as a child of a given *Runnable* element). For each Action scheduled for execution, a search for an appropriate execution method is performed. This search follows the same rules as parameter expansion, following both the base and group inheritance chains starting from the Runnable on whose behalf the Action is being run (which, because of inheritance, may not be the same as the Runnable in which the Action script is defined). If no execution method is found for a given Action, then the Action is launched using the built-in default method described above.

If per-Action method selection is not required, “global” *method* elements can be added to the top-most Group in the regression suite. If multiple top-level Groups are defined in the RMDB database, a base Runnable can be used to contain the *method* elements and each top-level Group refers to this base Runnable. In this case, multiple conditional *method* elements can be used to select the appropriate execution method according to script type, run mode (via a mode parameter), or any other relevant criteria.

For example, in order to change the execution method of a subset of the regression suite to always queue its Actions on a grid system, the following *method* can be defined within the top-most Group Runnable for the affected subset of the regression suite:

```
<rmdb>
  <runnable name="gridtests" type="group">
    <method>
      <command>qsub -V -cwd -b yes (%WRAPPER%)</command>
    </method>
    ...
  </runnable>
  ...
</rmdb>
```

Note that since this execution method is “selected” by virtue of its position in the regression suite hierarchy, it does not need to have a name. The only reason a *method* element would have to be given a name is when it is named as a base element by some other *method* element (base elements are always referred to by name). Otherwise, it is perfectly legal for a *method* element to be anonymous.

## Methods and Timeouts

Execution methods can specify a per-method minimum timeout by adding a *mintimeout* attribute onto the *method* element. The *mintimeout* attribute is read from the *method* element that actually supplied the command, even if the command is inherited from a base *method* element. In other words, even though the *mintimeout* attribute is actually attached to the *method* element, it is easier to think of it as being attached to the *command* element itself. For example, consider the following configuration elements:

```
<rmdb>
  <runnable type="base" base="global">
    <method name="m1" mintimeout="30" base="m2"/>
    <method name="m2" mintimeout="60">
      <command>qsub -V -cwd -b yes (%WRAPPER%)</command>
    </method>
    ...
  </runnable>
  ...
</rmdb>
```

Since the *m1 method* is unconditional, any Actions for which a matching execution method is not otherwise found will use this top-level execution method. However, since the *m1 method* element does not define a *command* element, the execution method command is taken from the *m2* base *method* element. Because the *command* is actually taken from the *m2 method* element, the *mintimeout* value attached to the *m2 method* element is used as the per-method contribution to the timeout calculations. While this can seem like attribute inheritance (which is not allowed), it really is not. What is happening is that the *command* element is being inherited via the base chain and the *mintimeout* attribute attached to the *m2 method* element is actually considered to be attached to the *command* element.

Note that there are technical reasons for not attaching the *mintimeout* attribute to the command element itself. The main reason is that this same *command* element is used in the Action scripts, where there can be multiple *command* elements. If the *mintimeout* attribute is attached to a *command* element within an Action script, then each *command* element can provide a different value and then there would have to be complex rules for determining the timeout value for the Action itself. Eventually, the minimum timeout value can become an element in its own right, which would then allow timeouts to be properly inherited like any other bit of data.

The calculation of timeouts are discussed in detail in “[Timeouts](#)” on page 311.

## Conditional Execution Methods

By itself, the ability to add *method* elements to Runnables in order to affect subsets of the regression suite is useful but still somewhat inflexible. In many cases, there can be a legitimate reason for wanting to launch some Actions locally, some on specific machines (that is, large capacity machines), and some on server grids of various types. Compiles, for example, may need to be executed on the local machine (or on a specific remote machine) prior to launching any simulations. Simulations may need to run on a grid during overnight regressions but might also be run on the local machine for small interactive test runs during the day. Some simulations may need additional resources or may require a slightly different set of options when the job is queued. All of these things point to the need for a more flexible paradigm for specifying the command by which each Action script is launched.

The solution is to allow various “condition” attributes to be added to each *method* element and then to allow a set of these conditional *method* elements to be scanned for the first such element that matches the current conditions. As mentioned before, the search follows the base/group inheritance chains (like parameters). At each step along the search, any *method* elements defined at that Runnable are checked, in document order, for a match to the current Action. The first matching *method* element found defines the execution method to be used for that Action script and the command defined therein is used to launch the Action script just as if that were the only *method* element specified.

The basic test for a conditional match uses the same *if* and *unless* attributes as for conditional execution of Runnables. The value string defined in the *if* and *unless* attributes of the *method* element, like their equivalents in the *runnable* element, are parameter-expanded and passed to the TCL interpreter for evaluation. If either attribute is absent or empty (that is, has an empty string for a value), the condition associated with that attribute is considered to pass. Otherwise, if the [optional] *if* attribute evaluates to a “true” value **and** the [optional] *unless* attribute evaluates to a “false” value (true and false being defined as in the TCL language), the *method* element is considered to be eligible for selection. If there are multiple *method* elements from within a given set that are eligible for selection, priority is given to the one defined first in document order, similar to a chain of *if/elseif/else* blocks in most programming languages.

In addition, several other conditional attributes can also be added to the *method* element for convenience. These conditional attributes compare fixed strings and are neither parameter-expanded nor TCL evaluated. **Table 9-1** is the set of conditional attributes.

**Table 9-1. Conditional Attributes for the *method* Element**

Attribute	Required condition for the condition attribute to evaluate to “true”
ACTION	The Action script type (preScript, execScript, and postScript) matches the value of the action attribute.
RUNNABLE	The name of the Runnable on whose behalf the Action script is being executed matches the value of the runnable attribute. <sup>1</sup>
CONTEXT	The context chain under which the Action is being executed matches the value of the context attribute. <sup>2</sup>

1. The phrase “on behalf of” implies that the important data point is the Runnable currently under execution and not the Runnable in which the Action script (which, in the case of an execScript, can be inherited from another Runnable) is defined.
2. This matching relies on the new “partial context match” algorithm described later in this document.

These attributes are referred to as “conditional attributes” in this document because all of these string comparisons can also be done with a combination of parameter expansion and TCL evaluation from within an *if* attribute. For example, consider the following conditional *method* elements:

```
<method action="execScript">...</method>
<method runnable="task1">...</method>
```

Since both *SCRIPT* and *RUNNABLE* already exist as predefined parameters (see “[Predefined Parameter Usage](#)” on page 266), these conditional execution methods are essentially shortcuts for the following equivalent element definitions:

```
<method if="(%SCRIPT%) eq {execScript}">...</method>, and
<method if="(%RUNNABLE%) eq {task1}">...</method>
```

The only real difference is that the *action* and *runnable* attributes are direct string comparisons, saving both the parameter expansion and TCL evaluation overhead. And, the “conditional” syntax is a bit easier to read.

Any given *method* element can specify multiple conditions, as in the following example:

```
<method runnable="task1" action="execScript" if="(%mode%) eq
{batch}">...</method>
```

The execution method represented by this *method* element is eligible for selection only when an *execScript* action is being executed on behalf of an instance of the Runnable *task1* and the value of the *mode* parameter (from the point of view of the executing Action) is set to the string *batch*.

By creating a series of conditional methods, in a predetermined sequence, one can represent pretty much any level of complexity in the selection of execution methods for running Action scripts.

Note that while it is legal to specify multiple condition attributes on a given *method* element, the XML format prohibits adding multiple attributes of the **same name** to a single element. For example, it is not possible to have a single *method* element with two *if* conditions. If this is required, an *unless* attribute (inverted, of course) can be used along with the *if* attribute or the various conditional checks can (and probably should) be combined to form a single parameterized TCL expression. If said expression is too complex to be represented in a one-line TCL expression, it can also be encoded into a TCL procedure, loaded into VRM from a *userTcl* element, and called from within the *if* expression.

As a typical example, the following execution method queues all *execScript* Actions to run on a grid without affecting the execution of *preScript* or *postScript* Actions:

```
<method name="all-simulations" action="execScript">
    <command>qsub -V -cwd -b yes (%WRAPPER%)</command>
</method>
```

The following execution method launches all Action scripts for the Runnable named “*compile*” in the foreground on a specific machine:

```
<method name="compile-phase" runnable="compile">
    <command>rsh bigmachine (%WRAPPER%)</command>
</method>
```

Note that in this example, if the Runnable is a Group, the *preScript* and *postScript* Actions associated with that Group will both use this method (assuming this is the only method provided). If only the *preScript* is to be launched on the remote machine, then an additional *action* attribute can be added to the *method* element as follows:

```
<method name="compile-phase" runnable="compile" action="preScript">
    <command>rsh bigmachine (%WRAPPER%)</command>
</method>
```

Note that if the *compile* Runnable is a Group, it has no *execScript* Action, as Groups only execute a *preScript* and a *postScript*.

It can also occur that a shared RMDB database might be used in more than one *mode* under different circumstances. For example, an interactive user might use a given database to run a subset of tests on the user’s local machine prior to check-in while another user might use the same database to launch the entire suite onto a compute grid as a nightly regression run. In this case, the recommended method would be to define multiple *method* elements in a list, each with a condition attribute that tests a *mode* parameter. The *mode* parameter can be supplied from the *vrun* command line (that is, *-Gmode=grid*) or can be defined in a top-level Runnable unique to

that particular mode. The list of *method* elements to accomplish a global *mode* based execution method selection might look like the following:

```
<method name="grid-mode" if="{(%mode%)} eq {grid}">
  <command>qsub -V -cwd -b yes (%WRAPPER%)</command>
</method>
<method name="gui-mode" if="{(%mode%)} eq {gui}">
  <command>(%WRAPPER%)</command>
</method>
```

Since the *if* and *unless* attributes are also evaluated as TCL expressions, the whole range of information available from TCL can be used to help select the execution method. For example, if the *mode* from the example above is stored in an environment variable at the time VRM is invoked, the following *method* elements can be used to query that environment variable for the execution mode:

```
<method name="grid-mode" if="${::env(VRUN_MODE)} eq {grid}">
  <command>qsub -V -cwd -b yes (%WRAPPER%)</command>
</method>
<method name="gui-mode" if="${::env(VRUN_MODE)} eq {gui}">
  <command>(%WRAPPER%)</command>
</method>
```

In a more complex situation, the *mode* test can be added to any number of *method* elements in order to filter the list of eligible execution methods to match the circumstances under which the tests are being run. This is another area where VRM provides a high degree of flexibility which, for the majority of real world projects, can never be used to its full capacity.

Only one method is selected for each Action script, as the Action script is only launched once. VRM will not combine commands from multiple *method* elements into a single launch command, since that would be very difficult to get right under all circumstances. For example, if a given Action script is to be launched onto a grid but the grid submission command must be executed on a specific machine, the combined command must be hand crafted by the user and added to the RMDB database in a separate (possibly conditional) *method* element as follows:

```
<method name="grid-from-remote-machine" if="...">
  <command>rsh grumpy 'qsub -V -cwd -b yes (%WRAPPER%)'</command>
</method>
```

The commands used in the above examples are intentionally simplistic. In real life, the *rsh* and *qsub* commands would probably need additional command-line options in order to work correctly. The sequence and interaction of these options, as well as the need to quote embedded commands, are the primary reasons VRM does not attempt to automatically combine multiple execution method commands into a single launch command.

## Methods and Base Inheritance

One or more *method* elements can appear at the top-level of the RMDB file, as children of the document (*rmdb*) element and siblings of the *runnable* elements. These *method* elements cannot

be used to satisfy an execution method search. They exist only as base methods to be referred to by other *method* elements. Any *method* elements referred to via the base inheritance chain is searched for by name at the top-level of the database. Since attributes are never inherited, the only use of base execution methods is to provide a common execution method command for multiple *method* elements defined within the various runnable elements. For example, in order to assign the same execution method to two subsets of the regression suite (but not the entire suite), the following pattern can be used:

```
<rmdb>
  <method name="grid">
    <command>qsub -V -cwd -b yes (%WRAPPER%)</command>
  </method>
  ...
  <runnable name="long-tests" type="group">
    <method base="grid"/>
    ...
  </runnable>
  ...
  <runnable name="big-tests" type="group">
    <method base="grid"/>
    ...
  </runnable>
  ...
</rmdb>
```

A base *method* element referred to by another *method* element is never consulted during the search for a matching execution method. It is only consulted in order to locate the appropriate launch command and other component parts of a matching *method* element once the referring *method* element has been selected (assuming the selected *method* element did not define those component parts itself). Conditions on *method* elements found at the top-level of the RMDB file are ignored, as are anonymous *method* elements defined at the top-level of the RMDB file.

Note that the base Runnables of a given Runnable are searched for matching *method* elements. But, unless a given *method* element matches, any base *method* element to which it refers is not consulted. And, once the initially matching *method* element is selected, the condition attributes on any base *method* elements consulted in the search for a *command* element to use for the execution method are ignored.

Following is an example illustrating this point:

```

<rmdb>
    <method name="build">
        <command>rsh bigmachine (%WRAPPER%)</command>
    </method>
    <method name="grid">
        <command>qsub -V -cwd -b yes (%WRAPPER%)</command>
    </method>
    ...
    <runnable name="normal-group" type="base">
        <method name="comp" action="preScript" base="build"/>
        <method name="sim" action="execScript" base="grid"/>
    ...
    </runnable>
    <runnable name="mygroup" type="group" base="normal-group">
        <members>
            <member>mytask</member>
        </members>
        <execScript>
            <command>vsim -coverage top</command>
        </execScript>
    </runnable>
    <runnable name="mytask" type="task">
        ...
    </runnable>
    ...
</rmdb>
```

When the *execScript* Action for the Task *mytask* is executed, a search for a matching execution method is initiated at the *runnable* element *mytask* (even though the *execScript* element itself was actually inherited from another Runnable). Since the *mytask* Runnable defines no execution methods itself, the *mygroup* Runnable is consulted as the immediate group ancestor (that is, parent) of the *mytask* Runnable. The *mygroup* Runnable, via its base Runnable *normal-group*, defines two *method* elements, one conditional on the Action type being a *preScript* and one conditional on it being an *execScript*. Because the Action in question is an *execScript*, the *sim* *method* element to launch the Action is selected. However, the *sim* *method* element defines no command. But because it does define a base method (*grid*), VRM looks for a top-level method element by that name. Within that *method* element we find the *qsub* command for launching the Action onto a server grid. This becomes the execution method of choice and the *execScript* script is posted to the grid using the *qsub* command provided.

This example illustrates several practices recommended in the paragraphs above. First, both the *execScript* itself and the method used to launch it are “owned” by the Group Runnable of which the *mytask* Task is a member. Presumably, this Group includes other members that will also use the same *execScript* and execution method, possibly differing only in the values of some parameters unique to each Task (that is, to each simulation). The second thing to note is that the list of conditional methods is actually defined in a base Runnable that can be reused elsewhere in the regression suite (presumably for another Group that shares the same execution method configuration). The last point is that the conditional execution methods in the *normal-group* base Runnable do not themselves define commands but simply “point” to base *method* elements

defined at the top-level of the database. So there could be an *abnormal-group* base Runnable somewhere in the regression suite that defines a slightly different set of conditional execution methods but uses the same commands (that is, points to the same base *method* elements) for some or all of the conditional execution methods.

It is important to remember that the test for a matching conditional execution method and the search for a command defined by that method are two separate operations. In the following example, the *build* and *grid* methods have both been modified to add additional conditions:

```
<rmdb>
    <method name="build" action="execScript"><!-- condition is ignored...
        see below -->
        <command>rsh bigmachine (%WRAPPER%)</command>
    </method>
    <method name="grid" runnable="anothergroup"><!-- condition is
        ignored... see below -->
        <command>qsub -V -cwd -b yes (%WRAPPER%)</command>
    </method>
    ...
    <runnable name="normal-group" type="base">
        <method name="comp" action="preScript" base="build"/>
        <method name="sim" action="execScript" base="grid"/>
        ...
    </runnable>
    <runnable name="mygroup" type="group" base="normal-group">
        <members>
            <member>mytask</member>
        </members>
        <execScript>
            <command>vsim -coverage top</command>
        </execScript>
    </runnable>
    <runnable name="mytask" type="task">
        ...
    </runnable>
    ...
</rmdb>
```

Assume you are about to launch the *execScript* Action for the *mytask* Runnable. Note carefully what does **not** happen. Since the *comp* method element comes before the *sim* method element in the *normal-group* base Runnable, the conditions on that method are tested first. Given that the *comp* method element is conditional on the Action type being *preScript*, it does not match the Action being executed (which, in this example, is an *execScript*). The fact that its base method *build* **does** match the conditions of the Action in question is moot since the base method element is never consulted (since the *comp* method element did not match in the first place).

Second, the condition on the *grid* method element, if it were considered, would fail in this case since the Action script is not being run from under a Runnable called *anothergroup*. But since the *sim* method in the *normal-group* base Runnable has already matched and has, thus, been selected as the execution method to be used, the *grid*

A *method* element is only queried for its ability to provide an actual execution method command to be used. Per the policy on base inheritance, the contents of the *grid* method element are being treated as if they were defined directly in the *sim* method element whose conditional attributes **do** match the conditions for the Action in question. The condition attributes of the *grid* method element, however, are not inherited by the *sim* method element because attributes always apply only to the element on which they are specified. Therefore, the launch command defined in the *grid* method element is used despite its apparent non-match.

Base method elements can be chained. That is, a base method may refer to another method via its own *base* attribute. The search along the base chain continues until a method containing a *command* element is found, at which point the *method* element containing the *command* element becomes the new selected execution method. If no *method* element containing a *command* element is found in the search along the base chain, then the original *method* element (the matching one inside a Runnable) remains the selected execution method.

In the event that an execution method is selected but no execution method command can be found, the selected execution method is used for its attribute values (possibly changing things like the minimum timeout and the queue selection (see “[Named Execution Queues](#)” on page 333 for details) but the Action itself is launched in the same way it would have been launched had there been no matching execution method.

With the exception of the conditional attributes (which, as noted above, only apply to the *method* elements occurring inside *runnable* elements), all other attribute values (like *mintimeout*) are read from the selected execution method (that is, from the *method* element that supplies the execution method command or, lacking an execution method command, the original matching *method* element from inside the Runnable). For example, consider the following RMDB fragment:

```
<rmdb>
    <method name="b1" mintimeout="20">
        <command>...command two...</command>
    <method>
    <method name="b2" mintimeout="35"/>
    <runnable name="mygroup" type="group">
        <method name="m1" runnable="test1" base="b1" mintimeout="10">
            <command>...command one...</command>
        </method>
        <method name="m2" runnable="test2" base="b1" mintimeout="25"/>
        <method name="m3" runnable="test3" base="b2" mintimeout="30"/>
        <members>
            <member>test1</member>
            <member>test2</member>
            <member>test3</member>
        </members>
    </runnable>
    <runnable name="test1" type="task">...</runnable>
    <runnable name="test2" type="task">...</runnable>
    <runnable name="test3" type="task">...</runnable>
</rmdb>
```

When the *test1* Runnable executes and a search is made for a matching execution method, *method* element *m1* is selected. Since *method* element *m1* already defined an execution method command, no further search is mounted. The *base* attribute in *method* element *m1* is moot. The minimum queue timeout used for the *execScript* Action defined in Runnable *test1* is 10 seconds, since that attribute is taken from the selected execution method, represented by *m1*.

When the *test2* Runnable executes, the *method* element *m2* is found to match. Since this *method* element contains no *command* element, the base methods is searched, starting with *method* element *b1*. The *method* element *b1* contains a *command* element so the search terminates there and *b1* becomes the selected method element. The minimum queue timeout used for the *execScript* Action defined in Runnable *test2* is therefore 20 seconds.

When the *test3* Runnable executed, *method* element *m3* matches and becomes the selected execution method. However, since *method* element *m3* contains no *command* element, the search continues along the base chain, starting with *method* element *b2*. In this case, *method* element *b2* contains neither a *command* element or a *base* attribute. Therefore, the commandless *method* element *m3*, being the matching execution method element from inside a Runnable, becomes the selected execution method. As a result, the minimum queue timeout used for the *execScript* Action defined in Runnable *test3* is 30 seconds.

## Execution Method Examples

---

The following are VRM execution method examples.

<b>Recommended Execution Method Configurations.....</b>	<b>327</b>
<b>Timing the Execution of Each Action .....</b>	<b>330</b>
<b>Queuing Actions to Run on a Server Grid .....</b>	<b>330</b>

## Recommended Execution Method Configurations

Note that there are actually only two recommended execution method configurations. If the user wishes to maintain a single list of *method* elements that apply to the entire database, relying entirely on the conditional attributes to select the proper method for each Action, the *method* elements should all be defined in the top-most Group of the regression suite, or in a base Runnable referred to by each top-level Group Runnable. If, however, there is a need to define a different set of execution methods for different parts of the regression suite topology, then the top-most Runnables for each of the major sections of the suite should carry a complete set of *method* elements pertaining to the section of the regression suite defined by that Runnable.

In the first recommended use model, the user defines a global set of methods that apply to the entire regression suite. In this case, the *nightly* Runnable is at the top-level of the regression suite hierarchy and a perfect place to define our global methods. This particular configuration results in the *preScript* Action for the *top* Runnable being launched on a dedicated build machine, all the *execScript* Actions being queued to a grid, and all other Actions being run on the local machine (by default, since no other matching *method* elements exist). Following is a common use model for simple suites:

```
<rmdb>
  <runnable name="nightly" type="group">
    <method runnable="top" action="preScript">
      <command>rsh buildmachine (%WRAPPER%)</command>
    </method>
    <method action="execScript">
      <command>qsub -V -cwd -b yes (%WRAPPER%)</command>
    </method>
    ...
  </runnable>
  ...
</rmdb>
```

Combined with a global parameter passed in from the command line (that is, *-Gmethod=grid*), this scheme can be used to implement a global method as follows:

```
<rmdb>
  <runnable name="top" type="group">
    <method if="%method%" eq {remote}>
      <command>rsh buildmachine (%WRAPPER%)</command>
    </method>
    <method if="%method%" eq {grid}">
      <command>bsub (%WRAPPER%)</command>
    </method>
    ...
  </runnable>
  ...
</rmdb>
```

In the second recommended use model, the user defines a list of one or more *method* elements within the top-most Group Runnable for a subset of the suite that requires special treatment. In this example, the suite contains a set of tests that are large enough that they must be launched on a specific high capacity machine. The *method* to launch the Actions on the more powerful machine is placed within the Group Runnable under which all these large tests are defined. This execution method only applies to the *execScript* Actions launched by Runnables under this particular Group (because of the *action* conditional attribute and the position of the *method* element itself) and all other Actions defined outside this Group are run under the default execution method (assuming no other *method* elements are defined elsewhere). For example,

```
<rmdb>
  <runnable name="nightly" type="group">
    <members>
      <member>bigtests</member>
      ...
    </members>
    ...
  </runnable>
  <runnable name="bigtests" type="group">
    <method action="execScript">
      <command>rsh bigmachine (%WRAPPER%)</command>
    </method>
    ...
  </runnable>
  ...
</rmdb>
```

If the same execution method command is used for multiple subsets of the regression suite, the command itself can be moved to a top-level *method* element and referred to from the various Runnables in which it is required as follows:

```

<rmdb>
  <method name="bigmachine">
    <command>rsh bigmachine (%WRAPPER%)</command>
  </method>
  ...
  <runnable name="bigdirectedtests" type="group">
    <method action="execScript" base="bigmachine"/>
  ...
  </runnable>
  <runnable name="bigrandomtests" type="group">
    <method action="execScript" base="bigmachine"/>
  ...
  </runnable>
  ...
</rmdb>
```

Since a typical regression suite might define, on average, maybe a half-dozen different execution methods (various grid configurations, remote shell commands, and so on), defining them all at the top-level of the database and then assigning them to various parts of the regression suite via anonymous (potentially conditional) *method* elements at the Runnable level, using base inheritance to find the execution method command, is better than typing the same execution method command over and over every time it is used.

If the *method* element corresponding to the selected execution method does not define a command, either itself or via a base *method* element, then the default execution method (launching on the local machine as a background task) is used. This is true even if there are other *method* elements that would have matched had the command-challenged method element not matched. This is true because there can be certain conditions under which the user wishes to define an execution method that intentionally refers to the default VRM execution method. In the following example, an unconditional empty method element is used to terminate the search for an execution method and force VRM to use the default method:

```

<runnable name="mygroup" type="group">
  <method action="execScript">
    <command>qsub -V -cwd -b yes (%WRAPPER%)</command>
  </method>
  <method/>
</runnable>
```

For the *mygroup* Group Runnable and all its descendants (barring more locally defined *method* elements), all *execScript* Actions are posted to a server grid via the *qsub* command and all other Actions (both *preScript* and *postScript*) are launched using the default execution method. The reason for this is that the second (empty) *method* element, being unconditional, always matches. This terminates the execution method search without ever considering any *method* elements defined later in the same Runnable or defined in ancestors of this Group. However, since the unconditional method defines no command (and has no base method), the predefined default

execution method is used. In the case where a limited subset of the regression suite is to be run under its own set of execution methods, this technique can be used to effectively isolate that subset of the suite from the list of execution methods defined at higher levels.

Adding an unconditional *method* element containing an execution method command to the top-most Runnable of the regression suite is a reasonable way to globally replace the built-in default execution method for Actions executed as part of that suite. In fact, multiple top-level Runnables can be defined that differ only in the execution method defined in each. For example, in the following case, the same suite that is executed as *nightly-local* on the local machine can also be executed on the grid by selecting *nightly-grid* instead. The bulk of the contents of the top-most Runnable come from the *top-level* base Runnable. The only difference is the execution method that is used to launch the resulting Actions. For example,

```
<rmdb>
  <runnable type="base" name="top-level">
    ...member list, parameters, and so on...
  </runnable>
  <runnable type="group" base="top-level" name="nightly-local"/>
  <runnable type="group" base="top-level" name="nightly-grid">
    <method>
      <command>qsub -V -cwd -b yes (%WRAPPER%)</command>
    </method>
  </runnable>
  ...
</rmdb>
```

Note that, unlike parameters, *method* elements defined within a *runnable* element are never (in fact, cannot be) referred to by name. Therefore, the *name* attribute is optional on those elements. Likewise, since a *method* element defined at the top-level of the RMDB file can only be referred to as a base, the *name* attribute is required.

## Timing the Execution of Each Action

The following method definition executes each Action under the *time* command (to measure the overall CPU time consumed by the Action script). The time stats are emitted to *stderr*, which must be redirected in order to capture the data. Note the use of the (%TASKDIR%) parameter reference (containing the path to the working directory for the Action) and the (%SCRIPT%) parameter reference (containing the name of the Action itself, used here to build a filename for the output). For examples,

```
<method name="testing">
  <command>time (%WRAPPER%) 2>(%TASKDIR%)/(%SCRIPT%).time</command>
</method>
```

## Queuing Actions to Run on a Server Grid

The following method definition causes each Action to be queued for execution on a Univa UGE server grid. The basic queue submission command is *qsub*. The other options cause the

queued job to be launched from the same directory as that in which VRM is invoked, to duplicate the existing environment on the grid machine, and to interpret the command as a binary executable.

```
<method name="uge-grid">
    <command>qsub -V -cwd -b yes (%WRAPPER%) </command>
</method>
```

In theory, the following method definition should queue Actions to an LSF server grid:

```
<method name="lsf-grid">
    <command>bsub (%WRAPPER%) </command>
</method>
```

Both methods can be defined in the same RMDB database. Actions can select one of the other in various ways as follows:

- If the *execScript* of a single Task (or all descendant Actions of a given Group) are to be queued to a specific grid, the corresponding method could be added directly to the Task or Group.
- If all *execScript* Actions in the entire regression suite are to be queued to the same grid, the corresponding method can be added to the document element with its *action* attribute set to *execScript*.
- If the intention is to make grid submission optional (to be enabled by the user at runtime), the corresponding method can be added to either the document element or a given Runnable and an **if** conditional attribute added which tests for a specific value in a parameter, such as *mode*. Then, the user can override the given parameter with a string that will match the setting in the *if* attribute in the *method* element (see example below).

In order to emulate the “global execution method” command-line option supported by VRM, one or more method *elements* can be added to the top-most Runnable element as follows:

```
<rmdb>
    <runnable type="group" name="nightly">
        <method if="{{(%mode%)} eq {uge}}">
            <command>qsub -V -cwd -b yes (%WRAPPER%) </command>
        </method>
        <method if="{{(%mode%)} eq {lsf}}">
            <command>bsub (%WRAPPER%) </command>
        </method>
        <method if="{{(%mode%)} eq {rsh}}">
            <command>rsh (%host%) (%WRAPPER%) </command>
        </method>
    </runnable>
    ...
</rmdb>
```

Then, in order to select one of these methods, use the *-G* command-line option to override the value of the *mode* parameter. For example,

**vrun -Gmode=uge**

...

**vrun -Gmode=rsh  
-Ghost=bigserver ...**

One of the defined methods can also be selected by default for all or a given subset of the regression suite by defining a value for the *mode* parameter as follows:

```
<rmdb>
  ...
  <Runnable name="top">
    <parameters>
      <parameter name="mode">uge</parameter>
    </parameters>
    ...
  </Runnable>
  ...
  <Runnable name="largetests">
    <parameters>
      <parameter name="mode">rsh</parameter>
      <parameter name="host">bigmachine</parameter>
    </parameters>
    ...
  </Runnable>
</rmdb>
```

See “[Execution Methods](#)” on page 315 for more details about how execution methods can be defined and selected.

# Execution Queues and Job Aggregation

---

Tests and other user scripts are specified in the RMDB in terms of Actions. The content of the *preScript*, *execScript*, and *postScript* elements are examples of Actions specified in the *RMDB*.

The *vrun* process can also generate Actions internally (such as *mergeScript* or *triageScript*) in order to launch concurrent scripts to implement features like auto-merge or auto-triage. By default, these Actions are launched individually, as background jobs on the local machine, in the order in which they are encountered. The “[Execution Methods](#)” on page 315 explains how the RMDB can be configured to launch certain Actions in some other manner, such as submitting said Actions to another server or to a server grid. In this section, it shows how the RMDB can be configured to divide the Actions into multiple queues and to combine multiple Actions from a single queue into a single multi-Action job.

<b>Named Execution Queues .....</b>	<b>333</b>
<b>Aggregating Multiple Actions in a Single Job.....</b>	<b>336</b>
<b>Limiting the Execution Time of a Given Named Queue .....</b>	<b>344</b>

## Named Execution Queues

As Actions become eligible to run, they are added to a single internal queue for processing. A background thread then pulls each Action, in turn, from this internal queue and generates the necessary script and wrapper files.

It is at this time that the execution method is determined, as discussed in “[Execution Methods](#)” on page 315.

Each Action executed by VRM in the course of a regression run is reduced to a single command that the *vrun* utility launches via the TCL *exec* command. By default, this single command consists of an invocation of *vish* or *vsim* pointing to a wrapper script (generated by *vrun*) that performs the bookkeeping necessary to execute and monitor the script supplied by the user for that Action. If an execution method is specified for a given Action, then the execution method command itself becomes the single command for *vrun* to launch directly. The execution method command itself must point to a wrapper-launch script that contains the original launch command, which causes either *vish* or *vsim* to be launched, either locally or on another host, again pointing to the wrapper script for the Action.

Certain peripheral capabilities accrue via the use of execution “method” commands. For example, a “method” can define its own timeouts (launch-to-start and start-to-completion) that are used in place of the default timeouts. These timeouts can be specified on a per-method basis so each execution method can carry timeout settings that are pertinent to that particular execution method (for example, the launch-to-start time is likely to be much larger when the resulting job is queued to a server grid than if it is executed locally). The reason this is possible is because, internally to *vrun*, timeouts are associated with the Action itself. The per-method timeout settings are really controlling per-Action settings that *vrun* maintains internally.

Other functional settings, such as “maxrunning” (a limit on the number of Action scripts that can be running simultaneously) cannot be defined on a per-Action basis since they are not associated with one particular Action but with the execution queue itself. By default, there is a single execution queue and the “maxrunning” setting for that execution queue is controlled on a global basis by the `-j` command-line option. In order to support a per-method “maxrunning” setting, each method must be assigned a separate execution queue.

Assigning an execution method to a particular named queue is as simple as adding a queue attribute to the method element and setting it to a unique name as follows:

```
<method queue="myqueue" . . .>
  ...
</method>
```

There is only one predefined named queue and that is the named queue whose name is a blank string (sometimes known as the “unnamed named queue”). Other than that, named queues are created as needed. There is no technical limit to the number of named queues that can be created save that each execution method defined in the RMDB can be assigned to only one named queue.

For example, assume the environment dictates that system integration tests must be done on a specific remote server but unit tests should be queued to a server grid. Assume each user is also required to limit the number of concurrent system integration tests to no more than two. You cannot simply specify the `-j 2` command-line option because that would also affect the unit tests which, supposedly, do not have a maximum concurrent execution limit. The way to setup the RMDB to support this environment might be as follows:

```
<rmdb>
  <runnable name="regression" type="group">
    <members>
      <member>integration</member>
      <member>unitests</member>
    </members>
    <method context="integration" queue="remote" maxrunning="2">
      <command>rsh bigmachine (%WRAPPER%)</command>
    </method>
    <method context="unitests" queue="grid">
      <command>gridsub (%WRAPPER%)</command>
    </method>
  </runnable>
  ...
</rmdb>
```

Because the value of *maxrunning* is associated with a given execution queue rather than with the Actions themselves, these two execution methods need to be assigned to different queues. The first method is conditioned on the Action falling within the *integration* context, in which the system integration tests are presumably defined. The value assigned to the *maxrunning* attribute prevents *vrun* from launching more than two Actions from this queue at any one time. The second execution method is conditioned on the Action falling within the *unitests* context and has no specified limit as to how many Actions can be launched concurrently. Any Actions

which fall outside of the two specified contexts (such as */regression/postScript* at the end) will be launched on the local machine, since neither of the specified execution methods match.

The name of the queue via which an Action is ultimately launched may be viewed during the run by specifying the *-verbose* option on the *vrun* command line.

## Aggregating Multiple Actions in a Single Job

By default, VRM launches each Action as a separate job, even when launching jobs on a grid. However, because the grid queuing process involves some overhead, it is sometimes more efficient to combine two or more short-running Actions into a single grid job submission. A multi-Action grid job is created when several eligible Actions are clubbed together into a single script that is launched as a single job on a compute grid. Generation of multi-Action jobs is controlled on a per-queue basis. Each named queue is configured via attributes attached to the *method* element that creates the named queue and that queue's settings apply to any Action which is launched via that named queue.

Prior to converting eligible Actions into running jobs, the various Action-specific scripts (including the *.wrap* wrapper script) for each Action are generated as usual and *vrun* searches for an execution method matching the relevant conditions. The Action in question is then added to the appropriate named queue, as specified in the execution method element (or to the default “unnamed” named queue if there is no matching execution method or the matching execution method does not specify a named queue). Once all eligible Actions have been processed and queued, *vrun* then scans each named queue to determine if a multi-Action job is called for or if the Actions contained in the named queue are to be executed individually.

There are currently two different kinds of multi-Action jobs as follows:

- Fixed-list clubbed jobs
- Grid array jobs

The rules governing each are very similar, but the VRM behavior for each is slightly different. The sections below outline the similarities and differences.

<b>Multi-Action Clubbing Fixed-list Mode .....</b>	<b>336</b>
<b>Creating Grid Array Jobs.....</b>	<b>337</b>
<b>Common Rules for Clubbed Jobs and Array Jobs.....</b>	<b>338</b>
<b>Controlling the Grid Submission Command.....</b>	<b>339</b>
<b>Generating and Launching Multi-Action Script Files .....</b>	<b>341</b>
<b>Multi-Action Grid Jobs and Concurrency .....</b>	<b>343</b>
<b>Multi-Action Grid Jobs and Local Rerun .....</b>	<b>343</b>
<b>Multi-Action Grid Jobs and maxrunning Execution Limits .....</b>	<b>343</b>
<b>Multi-Action Grid Jobs and Job Control .....</b>	<b>343</b>

## Multi-Action Clubbing Fixed-list Mode

In fixed-list mode, *vrun* assembles a list of one or more eligible Actions into a multi-Action wrapper script and launches a single execution method command (that is, a single grid job), which is responsible for running the Actions in the list. Each Action in the list is executed in

sequence, one Action at a time, as though the Action scripts had been concatenated into a single script. The list of Actions and the sequence in which they are executed is fixed and cannot be altered once the multi-Action command has been launched (hence the name “fixed-list mode”). Also, from that point forward, the individual Actions can no longer be controlled independently, although each individual Action’s wrapper script is still responsible for sending the requisite “start” and “done” messages to the launching *vrun* process.

As with most other queue-based controls, fixed-list mode is enabled by adding an attribute to the execution method element. If the execution method element is annotated with the *maxclubbing* attribute and the value of that attribute is an integer number greater than one, then fixed-length clubbing is enabled. The value of the *maxclubbing* attribute controls the maximum number of Actions that may be clubbed into a single job and is applied to the named queue associated with the execution method on which the attribute was found.

For example, the following execution method element associates itself with the named queue known as “grid” and enables fixed-length clubbing with a maximum of five Actions per job:

```
<method queue="grid" maxclubbing="5">
  ...
</method>
```

If you set the *maxclubbing* attribute to 1, the method element will treat the attribute as if had not been specified.

## Creating Grid Array Jobs

Some Grid Management system support the concept of “array jobs.” These are multi-Action jobs that are submitted to the grid as a single job but which the Grid Management system divides into single jobs as grid servers become available to run the jobs. Platform Computing Corporation Load Sharing Facility (LSF), Univa Grid Engine (UGE), Oracle Grid Engine, previously known as Sun Grid Engine (SGE), and Realtime Design Automation’s NetworkComputer (RTDA) support array jobs and the knowledge of how to construct grid jobs in both environments is built into VRM. The advantage of an array job over a fixed-list clubbed job is that the Actions combined into an array job can still run concurrently if the grid has the available capacity (as mentioned above, Actions clubbed in fixed-list mode are always executed sequentially).

To enable grid array jobs for a given named queue, the *maxarray* attribute is added to the execution method associated with the named queue. The *maxarray* attribute works in a similar manner to the *maxclubbing* attribute and actually supersedes the *maxclubbing* setting. That is, if both *maxarray* and *maxclubbing* attributes are present, then the *maxclubbing* attribute is ignored. If the value of the *maxarray* attribute is greater than one, then array jobs are enabled for that named queue. The value of the *maxarray* attribute controls the maximum number of Actions that may be aggregated into a single array job and is applied to the named queue associated with the execution method on which the attribute was found.

For example, the following execution method associates itself with the named queue known as *grid* and enables grid array jobs with a maximum of five Actions per job:

```
<method queue="grid" maxarray="5">
  ...
</method>
```

If you set the *maxarray* attribute to 1, the method element will treat the attribute as if had not been specified, but any *maxclubbing* attribute will be considered.

## Common Rules for Clubbed Jobs and Array Jobs

Since fixed-mode clubbing and grid array jobs are processed by the same algorithm, many of the rules that govern said processing are common for both modes. This section outlines those rules.

The value of the *maxarray* and/or *maxclubbing* attributes represents the maximum number of Actions that are aggregated into a single job. In the examples above, this maximum is set to five Actions. In this case, if nine Actions were to become eligible for execution at the same time, those nine Actions would be combined into, and launched as, two separate jobs (one job is responsible for the first five Actions and the other job responsible for the remaining four Actions). Note that if the number of eligible Actions waiting in the named queue is not evenly divisible by the maximum value at the time the jobs are launched, a short job is launched containing the remaining Actions in order to completely clear the queue. The algorithm will not wait for further Actions to become eligible, as this could cause deadlocks under certain circumstances.

If the value of the *maxarray* (or *maxclubbing*) attribute is zero, there is no limit on the number of Actions that can be combined into a single job. If the value of *maxarray* (or *maxclubbing*) is one, then a multi-Action job script is still created and launched, but only one Action is included in the job.

In the event that multiple execution method elements with differing *maxarray* (or *maxclubbing*) values refer to the same named queue, the queue uses the minimum of all those values attached to the *method* elements matching Actions that have been added to that queue up to the point that the list of Actions is de-queued and launched. In the following example, two execution methods point to the same named queue but each has a different *maxclubbing* value:

```
<method queue="grid" action="execScript" maxclubbing="10">...</method>
<method queue="grid" maxclubbing="5">...</method>
```

In this case, *execScript* Actions are allowed to be clubbed and launched ten at a time (that is, ten Actions in one grid job). Other Actions (like *preScript* and *postScript*) are only executed in jobs of five Actions each. However, since both execution methods point to the same named queue (in this case *grid*), any *preScript* or *postScript* queued to this named queue causes the *maxclubbing* value associated with the queue itself to be capped at five for the duration of the regression run. In other words, the *maxclubbing* value is cumulative and equal to the minimum *maxclubbing*

value from the execution method of all Actions that are queued to that particular named queue up to that point. However, if no Actions other than *execScript* actions are queued (because the *preScript* and *postScript* for all groups are either empty or undefined), the second execution method will never match and the *maxclubbing* setting for the named queue *grid* remains at 10 for the duration of the regression run.

Likewise, if multiple execution methods specify different aggregation modes, one mode will override the other just as if both attributes had been present on both execution methods, as shown in the following example:

```
<method queue="grid" action="execScript" maxclubbing="10">...</method>
<method queue="grid" maxarray="15">...</method>
```

In this case, the first Action to be queued which is not an *execScript* Action causes the *maxarray* value of the *grid* named queue to be set to 15 which, despite the fact that the previous value of 10 is actually less than the current value of 15, still overrides the previous value since *maxarray* always overrides *maxclubbing*. For clarity, it is strongly recommended to use a different named queue for execution methods whose limit attributes (like *maxarray* and/or *maxclubbing*) are not identical.

Note that nothing in the multi-Action aggregation algorithm is tied to the use of a specific Grid Management system, nor is it even necessary to launch multi-Action jobs on a grid. Whether or not a given job is submitted to a Grid Management system depends on the execution method command. Even if jobs are launched by the default method (that is, in the background on the local machine), Actions can still be aggregated into multi-Action jobs.

## Controlling the Grid Submission Command

Different Grid Management systems require different command-line formats to submit array jobs. In keeping with the design goal of allowing environment-specific behavior to be overridden to allow for local customization, an execution method may call upon a user-definable procedure to construct part of the execution method command. Built-in procedures to cover LSF, UGE, SGE, and RTDA grids are built into the *vrun* executable. Other Grid Management systems that support array jobs require an additional procedure to be defined.

Like the other grid-based user-definable procedures, the grid type must be specified in order to know which Grid Management system is being invoked. The grid type is specified in the *gridtype* attribute attached to the execution method element, as shown below:

```
<method queue="grid" gridtype="lsf">...</method>
```

The grid type designation for LSF grids is “lsf”, UGE grids is “uge”, SGE grids is “sge”, and RTDA grids is “rtda”. Other types can be defined by supplying the necessary user-definable procedures as discussed elsewhere. The grid type is a property of the named queue and is updated every time an Action is queued to a particular named queue. It is a bad idea to have

multiple execution methods point to the same named queue if the execution methods do not have identical *gridtype* attributes.

The name of the user-definable procedure used to help develop the execution method command is formed by appending the title-case form of the grid type to the string “SubmitOptions”. For example, the procedure called for a grid type of “lsf” is “LsfSubmitOptions”. As the name implies, this procedure returns pre-tokenized options (or a string if your RMDB version is 1.0, the chapter “[RMDB Versions](#)” for information on how this procedure differs between versions) containing command-line options to be used in the grid submit command. These options include: job name, any options necessary to designate a grid array job (including the size of the array), and the options for supplying file names and/or paths for the *stdout/stderr* log output of the job itself.

Any time an execution method is used to generate an execution method command for launching the job, the *gridtype* value for the named queue is consulted. If the value is non-blank, then it is used to construct the name of a user-defined procedure as described above (that is, `<gridtype>SubmitOptions`). If a procedure by that name exists, it is called with the usual data array containing certain particulars of the proxy Action associated with the job. Included in this data is a count of the number of Actions that have been aggregated to form the job in question. The `<gridtype>SubmitOptions` procedure returns a string with the appropriate command-line options for that grid type and for that number of Actions. This option string is placed in the predefined *GRIDOPTS* parameter. In addition, the grid type is placed in the predefined *GRIDTYPE* parameter. These parameters may be referenced from the execution method command.

In addition, if the Grid Management system in question supports grid array jobs, the user-defined `<gridtype>SubmitOptions` procedure must set an environment variable by the name of `MTI_VRUN_JOBIDX_ENV`. This environment variable should contain the name of another environment variable used by the Grid Management system to indicate which of the components of the array job is being executed by each invocation. For LSF grids, this variable name is `LSB_JOBINDEX`, for UGE grids it is `UGE_TASK_ID`, for SGE grids it is `SGE_TASK_ID`, and for RTDA grids it is `RTDA_TASK_ID`. Grid Management engines that do not pass the array index to the job script via an environment variable are not supported. In order to take advantage of this user-defined procedure, the execution method command must include a reference to the *GRIDOPTS* predefined parameter. This parameter is defined only when the *gridtype* attribute is specified and the appropriate `<gridtype>SubmitOptions` procedure is found (this is so that an error occurs if the execution method command is likely to be

incomplete as a result). An example of a properly constructed pair of execution methods for LSF and SGE grids, each configured to support array jobs, would look like the following:

```
<method if="[string equal {(%GRID:%) } {UGE}]" gridtype="uge" maxarray="0"
mintimeout="300">
    <command>qsub (%GRIDOPTS%) (%WRAPPER%)</command>
</method>
<method if="[string equal {(%GRID:%) } {LSF}]" gridtype="lsf" maxarray="0"
mintimeout="300">
    <command>bsub (%GRIDOPTS%) (%WRAPPER%)</command>
</method>
...

```

Each of these methods is conditional based on a *GRID* parameter that can be specified on the *vrun* command line. The actual grid submission command is not included in the *GRIDOPTS* parameter since the path would necessarily depend on the local environment (not all environments would necessarily have LSF, UGE, SGE, and RTDA grid submission commands on their default search path). The *WRAPPER* parameter is referenced, as usual, to point to the multi-Action wrapper script. Each execution method is set to accept arbitrarily large array jobs with an additional queuing timeout of 5 minutes.

## Generating and Launching Multi-Action Script Files

If an integer *maxarray* or *maxclubbing* value is associated with a named queue when it comes time to launch one or more Actions, a multi-Action job control script is created and launched via the execution method command. This script contains a list of per-Action launch commands, each of which matches the command normally stored in the execution method *.bat* file for that Action (that is, the command that was launched under single-action mode). The script executes the commands in this list sequentially and each command generates the *start* and *done* messages used by the controlling *vrun* instance to monitor and track the status of each Action. No messages are sent from the multi-Action job control script to the controlling *vrun* instance.

Following is an example of a multi-Action job control script:

```
set commands [list \
    {echo {do /path/to/VRMDATA/nightly/test~1/execScript.wrap} | \
     /path/to/modeltech/linux/vsim -c -l {} -onfinish stop} \
    {echo {do /path/to/VRMDATA/nightly/test~2/execScript.wrap} | \
     /path/to/modeltech/linux/vsim -c -l {} -onfinish stop} \
    {echo {do /path/to/VRMDATA/nightly/test~3/execScript.wrap} | \
     /path/to/modeltech/linux/vsim -c -l {} -onfinish stop} \
    {echo {do /path/to/VRMDATA/nightly/test~4/execScript.wrap} | \
     /path/to/modeltech/linux/vsim -c -l {} -onfinish stop} \
    {echo {do /path/to/VRMDATA/nightly/test~5/execScript.wrap} | \
     /path/to/modeltech/linux/vsim -c -l {} -onfinish stop} \
    ]
}

if {[info exists ::env(MTI_VRUN_JOBIDX_ENV)]} {
    set varname $::env(MTI_VRUN_JOBIDX_ENV)
} else {
    set varname {}
}

if {![string equal $varname {}] && [info exists ::env($varname)]} {
    uplevel #0 exec [lindex $commands [expr {$::env($varname) - 1}]]
} else {
    foreach command $commands {
        uplevel #0 exec $command
    }
}
```

As you can see, the multi-Action wrapper first looks for the *MTI\_VRUN\_JOBIDX\_ENV* environment variable. If this variable is set, it indicates that *vrun* believes the Grid Management system supports array jobs and that such an array job has, in fact, been submitted. It is assumed (by design, in fact) that the environment variable named in the *MTI\_VRUN\_JOBIDX\_ENV* environment variable contains the specific index of the Action to be executed with each invocation. In the supported Grid Management systems, this index is one-based (that is, starts at one and increments to the upper limit of the array). Therefore, it subtracts one when choosing the Action to execute. The commands in the list are the same commands that would have been placed into the execution method wrapper (*.bat*) if the Action were executed separately.

If the *MTI\_VRUN\_JOBIDX\_ENV* environment variable is not found, then the multi-Action wrapper assumes that the Actions in the list should be executed in sequence, as would happen in a fixed-list clubbed job. In fact, the same multi-Action wrapper script is used for grid array jobs and fixed-list clubbed jobs.

One of the Actions in the clubbed list is selected as the “proxy” Action. The multi-Action job control script and that script’s *stdout* and *stderr* output logs are written to the task directory associated with the proxy Action. The job ID returned by the grid submission command is associated with the proxy Action. All other Actions that were clubbed together with the proxy Action are annotated with the proxy Actions context string to make job control feasible.

The multi-Action job control script is named by combining the base name of the proxy Action (that is, *preScript*, *execScript*, *postScript*, and so on) with an extension of *.multi*. The remaining files, such as the wrapper (*.wrap*) and script (*.do*) files, are generated for each Action as usual.

Refer to “[Controlling Grid Jobs](#)” on page 477 for additional information.

## Multi-Action Grid Jobs and Concurrency

Actions clubbed into a single grid job are executed sequentially, without regard for the sequential mode of the group(s) from which they originated. Since Actions that originate in a sequential group can only become eligible to run one-at-a-time, those Actions will never be eligible to be clubbed together in a single grid job. In order for the members of a given group runnable to take advantage of multi-Action clubbing, they must be members of a non-sequential group; however, said group members, once clubbed, will actually run sequentially.

## Multi-Action Grid Jobs and Local Rerun

Actions that fail or timeout are still subject to local rerun. However, because the Action is simply submitted to the named queue a second time, it will almost certainly be clubbed with a different set of Actions and/or run in its own grid job (depending on whether any other Actions are eligible at the time). This could cause the multi-Action job control script and *stdout/stderr* logs for a given proxy Action to be overwritten.

## Multi-Action Grid Jobs and maxrunning Execution Limits

You can specify both *maxclubbing* and *maxrunning*.

In this case, the value of *maxrunning* is measured in grid jobs. So if the value of *maxclubbing* is set to five and *maxrunning* is set to 2 then the named queue can be a maximum of 5 jobs, each containing 2 actions.

## Multi-Action Grid Jobs and Job Control

Clubbed Actions running under a single multi-Action job control script receive only one job ID. For that reason, individual Actions within the clubbed job cannot be individually controlled. If the proxy Action for the clubbed job is passed to the *suspend*, *resume*, or *kill* job control options (or the equivalent commands via the control port), the entire multi-Action job is suspended, resumed, or killed, depending on the command or option specified. If the context string of a clubbed Action other than the proxy Action for the given job is passed to one of these commands/options, then nothing happens.

## Limits the Execution Time of a Given Named Queue

A related but independent function is the ability to limit the “execution window” of a given named queue. If the *window* attribute is specified for a given execution method, that attribute is used to establish a window of time during which the jobs launched from that queue are allowed to run. The value of the *window* attribute is parsed into one or more semicolon-separated expressions. Each expression consists of a limit type identifier and a limit value string separated by a colon. The only limit type identifier defined at this time is “wall” that means wall-clock time measured from the time the first job is launched. The limit value string follows the rules outlined in the man page for the TCL *clockscan* command.

As an example, below is an execution method that is associated with the named queue “grid”. This execution method clubs together all eligible Actions in a single grid job (since the value of *maxclubbing* is zero). It also specifies an execution window of 30 minutes from the time the first job is launched.

```
<method name="lsfgrid" gridtype="lsf" queue="grid" maxclubbing="0"
window="wall:30 mins">...</method>
```

What this implies is that any job launched via this execution method must finish within 30 minutes of the time the first job was launched from this queue or it will be killed. Moreover, any jobs launched or any Actions queued during this 30 minute window will also be killed or dropped when the execution window expires. Actions whose status is “executing” (that is, a *start* message is received) is reported as “killed” while Actions that have not yet started (because they were never launched or the multi-Action script had not reached that point in the list) are reported as “skipped.”

The sequence is as follows:

1. One or more Actions become eligible to run and are added to the named queue.
2. The first Action (or clubbed group of Actions) to be launched from this named queue “opens” the execution window.
3. Subsequent Actions may be added to and/or launched from the named queue without affecting the execution window.
4. When the execution window expires:
  - o Any pending jobs launched from this named queue are killed, and
  - o Any queued Actions still in the queue are dropped (and reported as skipped).
5. Once the execution window has expired, Actions subsequently added to the same named queue start the process anew.

Note that the named queue itself is not disabled. It may be used to launch a subsequent set of Actions via the same execution method. However, since consistent behavior depends on the

timing of the eligibility of the subsequent Actions, one should only consider this for cases where a clear dependency between several distinct sets of Actions is defined.

Note also that the execution window limit is independent of the multi-Action clubbing. One can take advantage of either without the other or of both together.

## Failure Rerun Functionality

One requirement of a good regression management system is the ability to rerun failed tests, possibly with additional debug output enabled. There are a number of ways in which VRM allows the user to rerun Actions from a previous run according to their status. The most obvious use for this function is to rerun tests that failed, either as is to filter out “noisy” failures or with different command-line options to enable additional debugging output.

The various methods available include the following:

- A local rerun method, enabled by default, whereby failing Actions can be re-executed.
- A global (command-line based) method, enabled by a command-line option, where *vrun* compiles a list of tests that failed across the entire regression run.
- A global (parameter-based) method, enabled by a predefined parameter, where *vrun* compiles one or more lists of failed tests according to parameters embedded in the RMDB file.
- A semi-automatic mode where *vrun* searches the event log for the immediately previous run looking for failed or not run tests.
- An automatic mode where *vrun* collects makes a second pass through a new set of Runnables, possibly with different command-line options.

Which method the user chooses depends on the complexity of the rerun methodology under consideration.

Note that the examples in this document assume a RMDB file named *default.rmdb* is located in the current directory, and that VRM defaults for any options not listed are adequate to run the regression suite defined therein.

<b>Local (Immediate) Rerun .....</b>	<b>347</b>
<b>Global (Command-line Based) Method .....</b>	<b>351</b>
<b>Global (Parameter-based) Method .....</b>	<b>353</b>
<b>Semi-automatic (Two-command) Mode .....</b>	<b>354</b>
<b>Fully Automatic (One-command) Mode .....</b>	<b>357</b>

## Local (Immediate) Rerun

VRM supports various levels of rerun functionality whereby failing Actions (that is, tests) can be re-executed, either as part of the same regression run or as part of a subsequent regression run. This section describes a level of failure rerun support called “local rerun” or “immediate rerun” (this document uses the local term).

Using the “global” rerun features (see “[Global \(Command-line Based\) Method](#)” on page 351 and “[Global \(Parameter-based\) Method](#)” on page 353), failing Actions may be automatically re-executed but only collectively at the end of the initial run and with only a single set of debug options to be used for the second pass. Local rerun schedules failing Actions for rerun as the failures occur without waiting for the end of the run. In addition, local rerun provides more granular control over which Actions should be rerun, how many times each should be rerun (depending on the Action, the nature of the failure, or any other visible attribute of the Action) and which parameters should be overridden for subsequent runs. A default behavior covers the common case and a user-definable TCL procedure allows nearly unlimited customizing.

The local rerun functionality is enabled by default. To disable it, specify the *-nolocalrerun* option on the *vrun* command line. With local rerun disabled, *vrun* reverts to its prior behavior of running each Action once and reporting the pass/fail status. Enabled or disabled, local rerun is completely independent of the existing global rerun functionality (activated by the *-rerun* command-line option) and neither affects the other (in fact, both can be used together in the same regression run).

Action Flow .....	347
Default Behavior .....	348
OkToRerun Procedure .....	349
Possible Use Models .....	350

## Action Flow

When local rerun is enabled, VRM calls the *OkToRerun* procedure immediately after *AnalyzePassFail* (in the case of a failure) and if this procedure returns a true value, then VRM re-queues the Action.

The status of the Action is analyzed by the *AnalyzePassFail* TCL procedure, after which the status is reported to the user log output and the Status Event Log. After that, the completion is propagated to any downstream Actions that may be waiting on the completion of the Action in question and the Action is deleted from the queues.

The execution graph node associated with this Action is marked with a list of reasons for each failure (a given Action can be rerun several times, depending on the situation) and that list is passed to the *OkToRerun* user-definable procedure to aid in deciding whether or not to attempt the Action yet one more time.

## Default Behavior

A default behavior is built into VRM so users can take advantage of local rerun without having to write an override implementation for the *OkToRerun* user-definable procedure. Under the default behavior, queue timeouts (status: timeout/queued) are treated separately from other kinds of failures. If an Action fails with a queue timeout (that is, the Action was launched but never reported having started), the Action is rerun unconditionally, regardless of its past history of failure. A queue timeout is most likely caused by a failure in the underlying grid management software or by a machine accepting a job but being unable to actually run it. In this case, re-launching the Action could fix the problem. So queue timeout failures always trigger a local rerun.

For non-timeout errors, only *execScript* Actions are eligible for local rerun. Actions that report queue timeouts are always rerun. Auto-rerun for Actions that fail to start (that is, status timeout/queued) are rerun regardless of the Action type or parameters (up to the auto-rerun limit).

For all other errors that involve an *execScript*, the history of reasons for failure (see above) is consulted and the Action is rerun when there is only one failure for the most recent reason. For example, if a given Action fails with an execution timeout (status: timeout/executing), then the history is consulted to see if this is the second time the Action had experienced an execution timeout and, if so, the Action is not queued for rerun. The theory is that a hard error which shows up twice in the same run is likely to continue to show up no matter how many times the Action is rerun. And while the whole point of rerunning an Action in the case of a queue timeout is because the Action is probably working fine but the execution infrastructure encountered a temporary problem, in the case of other types of failures, the point is to execute the Action a second time with debug instrumentation enabled. Therefore, rerunning an Action with a hard failure once the debug data has been collected is generally a waste of time.

For errors other than the queue timeout, a built-in parameter called *DEBUGMODE* is checked. The behavior is as follows:

- If the *DEBUGMODE* parameter is not defined, no rerun takes place.
- If the *DEBUGMODE* parameter is already set to true, yes, or 1, then the Action is not rerun.
- If the *DEBUGMODE* parameter is set to false, no, or 0, then the value of the parameter is set to the true, yes, or 1 (respectively) just before the second and subsequent runs.

This *DEBUGMODE* parameter can be used to control the command-line options used in the script such that subsequent runs will collect the debug information necessary to analyze the cause of the failure.

Note that in order to collect debug information on the rerun pass using local rerun, a debug version of the library and/or design must already be available. There is no way to rerun a compile step based on the failure of a simulation and, even if there were, you would likely not want to do that since it could affect subsequent un-run Actions that have not failed (yet). If debug instrumentation requires re-execution of earlier compile steps, then the existing global

rerun should be used (global rerun automatically picks up the *preScript/postScript* steps associated with the selected failed Actions).

The *DEBUGMODE* parameter is not defined for the initial run. In order to test their values without incurring an error, a local default should be used in the parameter reference as follows:

```
<command>echo "The current DEBUGMODE value is: (%DEBUGMODE:0%) "</command>
```

This is done so that when the local rerun functionality is disabled via the *-nolocalrerun* command-line option, the built-in parameters associated with local rerun are not defined (and so cannot collide with user parameters of the same name).

## OkToRerun Procedure

The *OkToRerun* procedure, like all other user-definable procedures, is passed a reference to a TCL array containing various context-specific data items. To override this procedure, the user writes a TCL procedure, generally following the example template shown below, and include it either in an auto-loaded *usercl* element or in an external file loaded with the *-loadcl* command-line option.

```
proc OkToRerun {userdata} {
    upvar $userdata data

    ... (some computation resulting in a true/false value) ...

    return $okToRerun
}
```

The *OkToRerun* procedure is called whenever an Action reports a failure or timeout status. The procedure inspects the current failure, in light of any previous history of failures, and determines whether or not another execution attempt for this Action is appropriate. If a true value is returned, then the Action is re-queued for execution. If a false value is returned, the Action is reported as having failed and is not re-executed in the current regression run (unless global rerun is activated (see [page 351](#)), which is not related to the local rerun functionality).

The *OkToRerun* procedure can also call the *OverrideRmdbParameter* procedure to set a local override value for any parameter visible to the Action in question (or to define a parameter that was not previously visible to the Action in question). The override value applies only to that Action and is in effect from the time the override is installed until that value is subsequently overridden by some other value.

A global hard limit is imposed by the *vrun* core that cannot be overridden via the *OkToRerun* user-definable procedure. The default global limit is 10 reruns per Action. This may be changed for a given regression run by using the *-maxrerun* command-line option. This limit is designed to prevent a regression run from looping forever on a failure condition that the user's *OkToRerun* procedure failed to detect. The default behavior depends on this to limit the number of queue timeout reruns that are executed for any given Action (since queue timeout errors are always rerun).

Refer to “[OkToRerun](#)” on page 512 for additional information.

## Possible Use Models

The use models suggested above are the two obvious common cases. In some regression environments, the simulation server grid (or the software that manages the grid) could drop one or more queued jobs at random. When this happens, the grid job will fail to run. As a result, the Action wrapper script will fail to report the *start* event, which will cause *vrun* to report a queue timeout. In order to combat this particular irregularity, the default local rerun functionality will always rerun an Action that fails due to a queue timeout. The parameters used to rerun the Action will remain unchanged from the previous run.

In the case of a failure induced by a fault in the design itself, subsequent executions are not likely to resolve the failure. Instead, a single rerun with debug options enabled is sometimes useful in order to gather waveform traces and other data that the designer may find helpful for determining the cause of the failure. By using local rerun to enable a debug rerun, the data can be gathered during an overnight regression run and be ready to use the following morning, as opposed to wasting the developer’s time rerunning the test in debug mode manually.

The default local rerun behavior is optimized to cover the most number of common situations with the most obvious possible evasive action. Consider, for example, the following scenario:

1. On the initial execution of a given Action (which represents one test), the grid system drops the job after it is submitted.
2. A queue timeout is detected and the Action is re-queued for execution with the same parameters.
3. This time the simulation runs but fails (because of a design error).
4. The local rerun process defined the *DEBUGMODE* parameter, sets it to 1, and queues the Action for rerun.
5. When the debug rerun pass is launched, the grid system again drops the job after it is submitted.
6. A queue timeout is detected and the Action is re-queued with the same parameters (including the *DEBUGMODE* parameter override).
7. The test executes in debug mode, leaving behind the necessary trace information, but still fails due to the design error.
8. Because the Action has now failed twice due to a hard design error (that is, not a queue timeout), the local rerun gives up and reports the failure.

In this complex sequence of events, the simple rules followed by the default local rerun implementation combine to effectively evade simultaneous grid problems and design errors, leaving the designer with a usable debug run to analyze once the regression run is done.

## Global (Command-line Based) Method

The *vrun* process can be instructed to dump out a log containing the full context path of the Runnable responsible for any Action that fails. Actions that encountered a timeout after launch are counted as failures, and Actions that did not run due to a failed dependency and/or a manual interrupt are also counted as failures.

Logging Failed Actions .....	351
Replaying Failed Actions.....	351

### Logging Failed Actions

The logging of failed Actions is enabled with the *-faillog* option whose option value is the path to a file in which the list of failed Runnables are recorded. If the path is not absolute, it is assumed to be relative to the directory from which *vrun* is launched. Any existing content in the file at the time *vrun* starts up is deleted.

The command line is as follows:

```
vrun <options> -faillog  
<path-to-faillog> <runnable(s)>
```

as in the example:

```
vrun -Gmode=fast -faillog ./failed-tests  
nightly
```

Given the above command, if you have a Task whose full context path is as follows:

```
nightly/directedtests/test123
```

and the *execScript* for that Runnable happened to fail, that full context path of the Runnable (minus the Action script component) is written to the *failed-tests* file under the directory where *vrun* is launched.

Note that the path given in the *-faillog* option must fall under the *vrun \$cwd* or under the *SCRATH* directory tree or *vrun* emits an error message and terminates the run (see “[File Safety](#)” on page 295 for additional information).

### Replaying Failed Actions

Failed Actions logged in the *-faillog* file from a previous run can be rerun by inserting the full context chains listed therein on a subsequent *vrun* command line as follows:

```
vrun <options>  
'cat <path-to-faillog>'
```

or the path to the failure log file can be passed as the value of the *-tasklist* command-line option, which reads the context chains from the file (one-per-line) and treats them as if they had been passed as bare arguments on the command line as follows:

```
vrun <options> -tasklist  
<path-to-faillog>
```

Note that the *-tasklist* option expects to be passed a file that contains only Runnable context chains, one-per-line. It is not a general purpose option file.

Once the initial *vrun* regression run is complete, the failure log file should contain a list of the Runnables that must be re-executed to cover all the failing Actions. When this list is expanded to form the execution graph, the graph also includes all *preScript* and *postScript* Actions defined in the intermediate Groups. In the example above, even if the *execScript* for *nightly/directedtests/test123* is the only failing Action, the following Actions, if defined, is re-executed in the sequence given as follows:

```
nightly/preScript  
nightly/directedtests/preScript  
nightly/directedtests/test123/execScript  
nightly/directedtests/postScript  
nightly/postScript
```

By placing critical compile commands in the *preScript* and post-processing commands in the *postScript*, rerunning a failed test also reruns the compile and post-processing steps in the proper sequence. If, however, the *directedtests* Group is a sequential Group with other members occurring prior to *test123* in the sequence, then the other Group members are not run automatically.

Note that this behavior is not unique to failure rerun. Anytime a context chain with multiple components is passed to *vrun* for execution, the *preScript* and *postScript* Actions for the intermediate Groups are included in the execution graph.

If the value of the *-vrmdata* option points to the same VRMDATA area as was used in the previous run, the re-execution will take place in the same working directories as were used for the original run, and any UCDB files from previously passed tests can also participate in the generation of new merged coverage results. Since the *preScript* and *postScript* are also executed for any Group in which a failing test was found, performing merge operations in the *postScript* should cause the right thing to happen without any user intervention.

Since the faillog file is read by the *-tasklist* command before the actual execution begins, it is safe to use the *-faillog* option on the same command line as a *-tasklist* option pointing to the same file. By doing so, the user can create a script that tries rerunning the failing tests under different scenarios until they all pass.

Refer to “[Automating Failure Rerun With a Macro Runnable](#)” on page 355 for information on how these manual commands can be packaged as a macro Runnable in the RMDB file.

## Global (Parameter-based) Method

The Global method of specifying a failed test log file involves adding a special predefined parameter to each Group for which a failed test log file is to be maintained. This method only works with *execScript* Actions. If *vrun* is able to resolve the parameter reference (*%faillog:%*) to a non-empty string for any given *execScript* Action, it assumes the parameter value represents the path to a file in which a list of failing Tasks, one-per-line, is to be stored. If the *execScript* reports a failure, it adds the context path of the Task (without the trailing *execScript* component) to this file. There can be more than one such file. You might, for example, have two or three top-level Groups, each of which maintains its own failure log. In that case, you would define a unique *faillog* parameter in each Group and the file path specified in each of those parameters would apply to all the Tasks under that particular Group (at any depth, since inheritance knows no limits).

Note that the reason this applies only to *execScript* Actions is because the general model is to reserve *preScript* for set-up or compile operations and *postScript* for cleanup, analysis, and reporting operations. If you use the recommended model, then *faillog* really represents failed simulations.

If the *faillogparameter* contains a relative path, it is assumed to be relative to the top of the VRMDATA directory tree. The user is responsible for clearing the *faillog* file(s) at the start of each *vrun* regression run. But since the parameter is also visible from the *preScript* of the Group in which they are defined, that should not be a problem.

A few notes on the sequence. When an Action completes, the user-definable *AnalyzePassFail* procedure is called. Among other things, the path to the UCDB (if defined) is passed to this function and its job is to return either pass or fail to its caller. All other behavior predicated on pass/fail is based on this return value. The *faillog* value is determined just before this procedure is called and that value is passed to the *AnalyzePassFail* procedure via the *\$data(faillog)* argument. In this way, a user-defined version of *AnalyzePassFail* could, in theory, write some text to the failed test log itself rather than rely on VRM to append the Task context to the file. In that case, however, the override procedure should also clear the *\$data(faillog)* hash entry so the default VRM algorithm does not also write to the file.

A failed test log generated with this method would be replayed using the same procedure documented in “[Replaying Failed Actions](#)” on page 351.

## Semi-automatic (Two-command) Mode

This section describes a way to identify failing Actions from a previous run. The *-select* command-line option takes one or more status keywords and searches the event log from the immediately previous run for Actions that match the specified keywords. It then converts the Action context path into a Runnable context path and then selects that context path for execution as if it had been included directly on the *vrun* command line.

Each *vrun* run emits an event log containing a record of every Action launched, when it started, and when (if ever) it completed. It also records events for Actions that were not executed, either due to a failing dependency or due to a manual interrupt. The status of the test is encoded into a “done” event in this log file. [Table 9-2](#) contains the status types.

**Table 9-2. Status Test Keyword Types**

Status	Description
passed	Action passed.
failed	Action failed.
timeout	Action exceeded timeout.
skipped	Action was not run due to earlier errors.
killed	Action was killed by user intervention.
dropped	Action was not run for unknown reason.

The keywords that the *-select* command recognizes include any of the above status keywords or any of the following composite keywords (or any unique abbreviation of either). [Table 9-3](#) contains the composite keywords.

**Table 9-3. Composite Keywords Recognized**

Keyword	Description
run	All Actions that executed (passed + failed).
notrun	All Actions that neither passed nor failed (timeout + skipped + killed + dropped).
all	All Actions previously selected (run + notrun).

For example, if there are three failing Actions and two more that did not execute because of one of the failures, the following command selects all five tests for re-execution:

```
vrun -select failed,notrun
```

The value passed to the *-select* option can be a comma or space separated list of keywords from the table above. If spaces are used, then the option value may have to be quoted, according to the rules of the command shell.

As in the manual mode, the *preScript* and *postScript* Actions of the intermediate Groups associated with the failing Actions are also selected for execution.

Refer to “[Reuse of Random Seeds in Automated Rerun](#)” on page 358 for information on the effect of failure rerun on random seeds.

**Automating Failure Rerun With a Macro Runnable..... 355**

## Automating Failure Rerun With a Macro Runnable

Two or more *vrun* commands are required to implement a complete failure rerun scheme in semi-automatic mode. In order to encapsulate the required command-line options into the RMDB file, a macro Runnable can be defined that recursively calls *vrun* as many times as needed to complete the regression run. For example, suppose your regression environment requires that failing and/or timed out tests be rerun once with a slightly longer timeout (to weed out noisy failures) and those that still fail to be run with a debug-mode parameter set so the tests are ready to debug when the designer arrives the next morning. The set of three *vrun* commands might look like the following:

```
vrun nightly -Gmode=fast
vrun -select failed,notrun
-Gmode=fast -mintimeout=300
vrun -select failed,notrun
-Gmode=debug -mintimeout=300
```

These commands can be placed in the *execScript* of a standalone Task to automate the chain as follows:

```
<rmdb>
  <runnable name="runall" type="task">
    <parameters>
      <parameter name="SAME">-rmdb (%RMDBFILE%) -vrmdatas
        (%DATADIR%)</parameter>
    </parameters>
    <execScript>
      <command>vrun (%SAME%) nightly -Gmode=fast</command>
      <command>vrun (%SAME%) -select failed,notrun -Gmode=fast
        -mintimeout=300</command>
      <command>vrun (%SAME%) -select failed,notrun -Gmode=debug
        -mintimeout=300</command>
    </execScript>
  </runnable>
  ...
</rmdb>
```

Then, the following command:

**vrun runall**

runs the regression suite rooted at *nightly* three times. Appropriate TCL code can also be inserted to detect failed Actions (possibly using the *-faillog* option) and bypass the subsequent runs altogether if the number of failures is below a fixed threshold (if there were **no** failures, the latter two *vrun* invocations will exit right away as there are no Runnables selected for execution).

Note that the options listed in the *SAME* parameter in the example above are required to pass the RMDB file name to the nested *vrun* command, and to prevent an additional *VRMDATA* directory (under *runall*) from being created.

## Fully Automatic (One-command) Mode

VRM supports a single-command automated way to run two passes over a given regression suite, picking up additional command-line options on the second pass. The *-rerun* option takes a string that contains the additional command-line options for the second pass (which must be an empty argument should you decide to run the regression suite twice with the same arguments both times). The command is as follows:

```
vrun  
<first-pass-options> -rerun <second-pass-options> <runnable(s)>
```

For example:

```
vrun nightly -Gmode=normal  
-rerun "-select failed,notrun -Gmode=debug"
```

In the example above, the regression suite rooted in the *nightly* Runnable executes with the *mode* parameter set to *normal*. Once that run is complete, the failed and/or un-executed Actions from the previous run (as well as the *preScript* and *postScript* Actions from the intermediate Groups) are re-executed with the *mode* parameter set to *debug*. Note that the string passed to the *-rerun* option is quoted to prevent the command shell from mistaking it for multiple command-line options.

Prior to executing the second pass, *vrun* clears the list of selected Runnables so that only those Runnables explicitly listed in the *-rerun* option string are considered for the second pass. In addition, *vrun* disables any *-runlist* option that may have been passed on the original command line. If no Runnables are selected by any of the “second pass” options specified in the *-rerun* option string, *vrun* will exit its second pass with the usual “Nothing to run” error.

In addition, certain options are blocked when the *-rerun* option embedded “second pass” options are being processed. Options such as *-rmdb*, *-vrmda*, and *-loadtcl* affect global state and therefore cannot be changed between the original run and the rerun. Also, the *-rerun* option itself is blocked from the “second pass” options.

The options specified in the value string passed to the *-rerun* option are parsed in a shell-line way (honoring embedded single and/or double quotes) and are processed in the same way as options on the *vrun* command line itself. With the exceptions noted above, any options specified on the original *vrun* command line remain specified for the second pass unless overridden by an option in the *-rerun* option string. Option errors detected at this point will terminate the rerun but will not invalidate the results of the original run.

The original run and the second-pass run share a common event log. That means that any GUI or 3rd party tools that may be monitoring the progress of the regression run will see both the original pass/fail status of each Action and then, later, the updated status of any Actions that were re-executed. Also, the *RegressionStarting* and *RegressionCompleted* user-definable procedures are called only once each for the combined run. The failure log (enabled by *-faillog*) may not be useful, as it will contain the union of all the context paths associated with every test that failed in either run. A separate status summary is emitted to the *vrun* log output for each run

and a second “begin” event is logged to the event log to demarcate the start of the second pass. Validation and semantic checking of the RMDB file is performed only once at the start of the *vrun* invocation.

Refer to [Reuse of Random Seeds in Automated Rerun](#) below for information on the effect of failure rerun on random seeds.

**Reuse of Random Seeds in Automated Rerun .....** 358

## Reuse of Random Seeds in Automated Rerun

When running a random simulation, a random seed can be supplied to *vsim* via a command-line option. The simulator guarantees that the same design run twice with the same seed will produce the same results. This is known as “random stability” and is essential to allow bugs exhibited by purely random simulations to be reproduced and debugged. When a series of random simulations are executed for the purpose of locating seeds with good coverage (or which turn up corner-case bugs), it is common to pass the word “random” to the simulation as a seed, as opposed to a fixed numeric seed value. This tells the simulator to select a seed at random at the start of the simulation. Of course, if a test is run twice with the word “random” as the seed, the results of the second run will almost certainly not match the results of the original run.

This becomes a problem for the failure rerun algorithm. The purpose of the rerun is usually to execute the simulation a second time, possibly with additional visibility or trace/dump options enabled, in order to give the user a head start at debugging any failures. This purpose is thwarted if the second run differs significantly from the initial failing run. To solve this problem, *vrun* has a mechanism whereby it can capture the seed from the initial (random seed) run and use that to override the random seed value for the second run, thus re-establishing random stability across the two runs.

In order to accomplish this, the “done” record in the event log records the seed used for each simulation. It reads this seed from the test data record of the UCDB file for the test in question (if no UCDB file is defined for a given Action, no seed is recorded). When the *-select* option is used to select Runnables from a previous event log file for execution, the seeds are also read from the event log. This happens regardless of the status of tests being selected (that is, if the option is *-select all*, then all tests from the previous run are re-executed, each one with the same random seed used in the initial run of that test). By default, any time the *-select* command-line option is used to select tests to rerun, the random seeds for each test are reused for those tests so selected.

The default behavior is that the random seed is always reused when a test is re-executed in the same VRM regression run regardless of whether the local rerun or global rerun mechanism triggered the rerun.

If this default behavior is not desirable for some reason, the *-noreuseseeds* option disables the propagation of seed values from the previous run to the current run. The option to not reuse the

random seeds is global to that run (that is, if `-noreuseseeds` is specified, no seeds from the previous run is propagated forward to the current run). Note that the behavior relies on the UCDB test data record so a test that does not generate a UCDB file, or whose UCDB file has been altered or deleted before `vrun` has a chance to retrieve the seed, will not reuse earlier seeds on subsequent runs. In other words, if there is no UCDB, then there is no seed reuse.



# Chapter 10

## Post-execution Analysis

---

After generating the necessary files, *vrun* launches the Action and waits for the “done” message to arrive from the wrapper script for that Action, indicating that the user’s script has exited and returned control to the wrapper. From this point until *vrun* continues with the next Action is called “post-execution analysis.” The behavior of VRM as it analyzes and reacts to the results of each completed Action is highly customized. Therefore, most of the steps involved in post-execution analysis are user-definable, with reasonable default behavior for the common case.

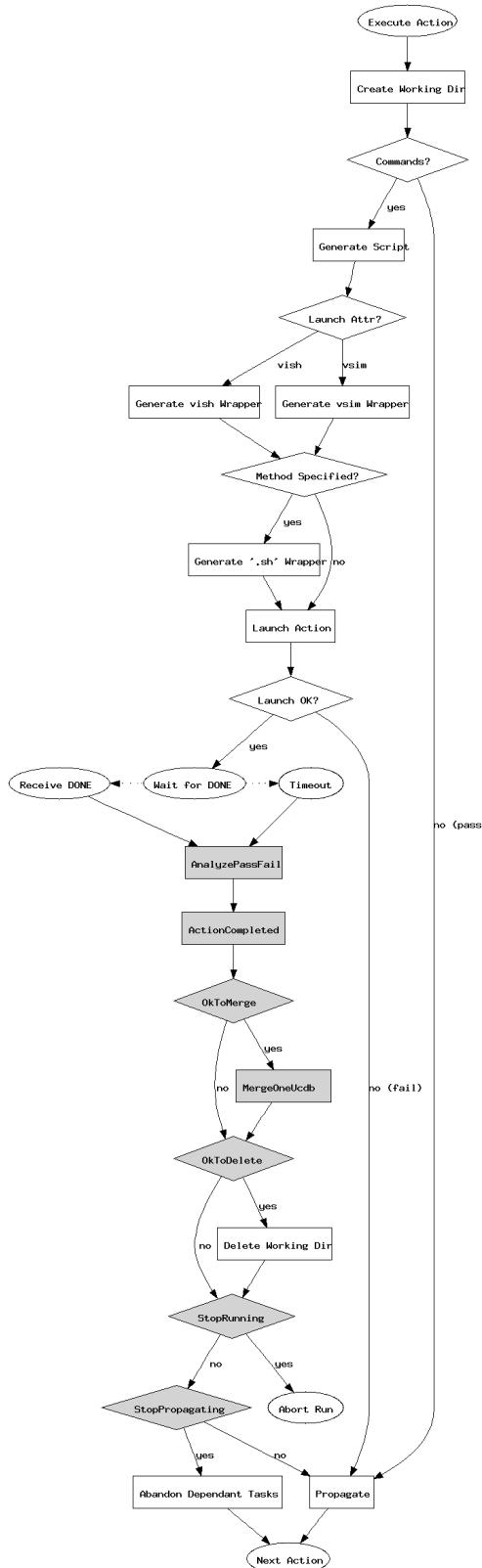
The major steps each Action encounters in its progression from “eligible to run” to “done” is illustrated in [Figure 10-1](#) on page 362, where the blocks in gray represent the behavioral units that the user can override. These overrides are in the form of TCL procedures. The *vrun* process calls each of these user-definable procedures at the proper time to determine how to handle the recently completed Action. The user-definable procedures correspond to the names in the gray blocks as follows:

- “[Pass/Fail Analysis](#)” on page 364
- “[ActionCompleted Concept](#)” on page 369
- “[OkToMerge Concept](#)” on page 369
- “[MergeOneUcdb Concept](#)” on page 370
- “[OkToDelete](#)” on page 517
- “[StopRunning Concept](#)” on page 370
- “[StopPropagating Concept](#)” on page 371

These sections describe how *vrun* post-processes a completed Action, taking each behavioral unit in turn.

For details on the user-definable procedures, see “[User-definable Procedures and Loading of Arbitrary TCL Code](#)” on page 412.

For a complete list of user-definable procedures, see “[Catalog of User-Definable and Utility Procedures](#)” on page 499.

**Figure 10-1. Post-execution Analysis Flow**

<b>Pass/Fail Analysis</b>	<b>364</b>
<b>Auto-Delete Functionality</b>	<b>372</b>
<b>Coverage Merge Options</b>	<b>376</b>
<b>Coverage Merging Use Models</b>	<b>379</b>
<b>Multi-level Auto-merge Example</b>	<b>403</b>
<b>User-definable Procedures and Loading of Arbitrary TCL Code</b>	<b>412</b>
<b>Collate Pass/Fail UCDB Files for Merge and/or Triage</b>	<b>422</b>

## Pass/Fail Analysis

Before *vrun* can decide how to react to a completed Action, it must determine if the Action in question passed or failed. For purposes of pass/fail analysis, each *execScript* Action is assumed to run, at most, one simulation. The *preScript* and *postScript* Actions are assumed not to run simulations (the *preScript* and *postScript* Actions are primarily used for compile, setup, and/or post-processing commands). These assumptions can be overridden in the RMDB database as described below.

In order to determine the status of a simulation, it is necessary to locate the UCDB file associated with that simulation in order to read its test data record. For an *execScript* Action, the easiest way to specify the name and location of the UCDB file is to specify a value for the *ucdbfile* parameter either in the Task itself, in a base runnable associated with the Task, or in a parent Group (as described in “[Inheritance, Overrides, and Search Order](#)” on page 271). This parameter can, itself, contain nested parameter references (as can all parameters) and its value can also be used as the argument for the *coverage save* command in the *execScript*. For example:

```
<runnable type="task" name="test1">
  <parameters>
    <parameter name="top">top</parameter>
    <parameter name="seed">123</parameter>
    <parameter name="ucdbfile">../test(%seed%).ucdb</parameter>
  </parameters>
  ...
  <execScript>
    <command>vsim -lib ../work -sv_seed (%seed%) -c (%top%)</command>
    <command>coverage attribute -name TESTNAME -value
      test(%seed%)</command>
    <command>coverage save -onexit (%ucdbfile%)</command>
    <command>run -all</command>
  </execScript>
  ...
</runnable>
```

In this case, the *ucdbfile* parameter depends on the value of the *seed* parameter. The simulation Tasks in this particular group are also designed to save their UCDB files in the working directory of their parent Group. Therefore, the UCDB filename comes out, in the case of “test1” above, to *../test123.ucdb*. This string is substituted in place of the (%*ucdbfile*%) parameter reference in the script (to save the UCDB file), and is also used by VRM to determine the status of the simulation. If the *ucdbfile* is a relative path, it is assumed to be relative to the working directory from which the *execScript* Action was executed.

In the implementation, if a UCDB file is not specified for a given *execScript* Action, then that Action is not included in the pass/fail analysis. If a UCDB file is specified but cannot be found, or if the test data record cannot be read, then the Action is counted as having failed with the status “UCDB error.”

The default pass/fail analysis process assumes that the Action in question represents a single simulation. Therefore, it also assumes that there is only one test data record in the UCDB file. If multiple test data records are found in the UCDB file, only the first test data record is read and the rest are silently ignored. If UCDBs with multiple test data records must be examined as part of the pass/fail analysis, then a user-defined procedure should be used to override the *AnalyzePassFail* behavior.

The numeric status code returned from the Action script is also passed to the *AnalyzePassFail* procedure. By default, if the status code returned from the Action script is non-zero, the Action is automatically considered to have failed and the UCDB is not consulted.

<b>Failure Propagation</b> .....	<b>365</b>
<b>Determining Pass/Fail for a Given Action</b> .....	<b>367</b>
<b>ActionCompleted Concept</b> .....	<b>369</b>
<b>OkToMerge Concept</b> .....	<b>369</b>
<b>MergeOneUcdb Concept</b> .....	<b>370</b>
<b>StopRunning Concept</b> .....	<b>370</b>
<b>StopPropagating Concept</b> .....	<b>371</b>

## Failure Propagation

A failed Action (one which either fails to launch, returns a non-zero status, or is marked as failed in the UCDB) can also have an effect on subsequent Actions that depend on (that is, follow) the failed Action. This mechanism is called “failure propagation” and is intended to help prevent cases where some critical Action within a regression suite (such as design compilation) fails but the remainder of the regression continues on as if nothing happened.

Failure propagation follows the connectivity of the VRM execution model, as described in “[Execution Graph Expansion](#)” on page 256 and summarized below.

In a sequential Group, the first member of the Group depends on the completion of the *preScript*, each member of the Group depends on the completion of the previous member in the Group, and the *postScript* depends on completion of the last member in the Group. In a non-sequential Group, each member of the Group depends on the completion of the *preScript* and the *postScript* depends on the collective completion of all the members of the Group. These dependencies are nested in the case of Groups which have, as members, other Groups.

The default rules governing failure propagation are as follows:

- A failed *preScript* Action causes any actions dependent upon that *preScript* (such as the *execScript* Actions of member Tasks or the *postScript* Action of that Group) to be abandoned.

- A failed member of a sequential Group causes both subsequent members of the Group and the *postScript* for that Group to be abandoned.
- Failed members of a non-sequential Group have no effect on the *postScript* Action of that Group.
- An abandoned Action appears as a failure to downstream dependent Actions so the whole process repeats for subsequent Actions.
- Failures propagated from the *postScript* of a member Group are treated as failures of the member (in other words, nested Groups can also propagate failure to their parent Groups).

The reasoning behind these rules is as follows:

- The *preScript* is most often associated with design compilation or other preparatory steps, the failure of which will most likely render further processing within that Group to fail and, therefore, to waste CPU resources.
- A failing *execScript* member of a non-sequential Group most likely results from a failed simulation. Simulation failure should be reflected in the merged UCDB test data records (and that merged UCDB file only exists if the *postScript* is allowed to execute). Post-simulation analysis (that is, *postScript*) Actions are less likely to fail themselves in the face of one or more simulation failures.
- The members of a sequential Group are most likely defined to execute sequentially because each step depends on the results and/or collateral files from a previous step (that is, in some RMDB files, compilation, simulation, and post-regression analysis are configured as members of a sequential Group).

There are two ways to override this default behavior. The easiest (and least flexible) is to add an *onerror* attribute to the *runnable* element corresponding to the Group whose behavior you wish to modify. For example, by default, failure of one member of a non-sequential Group is not propagated to the *postScript* of that Group or to other downstream Actions. In order to cause member failures to be propagated from inside a non-sequential Group, the *runnable* should be configured as follows:

```
<runnable name="myGroup" type="group" onerror="abandon">
  ...
</runnable>
```

An *onerror* value of *abandon* means that member failures should always cause the *postScript* of that Group to be abandoned (default behavior for sequential Groups). Likewise, an *onerror* value of *continue* means that member failures should not be propagated to subsequent members or to the *postScript* of the Group (default behavior for non-sequential Groups). If the *onerror* attribute of a given *runnable* element contains any other value, then the Group follows its default behavior as outlined above.

A more flexible means to override the default VRM failure propagation behavior is to override the *StopPropagating* user-definable procedure (as covered in the section “[StopPropagating Concept](#)” on page 371). This procedure is called after the completion of each Action. If the procedure returns a true value, downstream Actions are abandoned. If the procedure returns a false value, any failure of the Action in question is ignored (regardless of whether or not the Action really failed).

Whenever an Action in the execution graph is abandoned due to earlier errors, a warning to that effect is emitted to the VRM log output. In *-verbose* mode, this warning also includes a list of those Actions whose failure caused the Action in question to be abandoned.

Note that this behavior is in addition to any user-specified premature termination heuristic. Premature termination (see “[StopRunning Concept](#)” on page 370) causes all unexecuted Actions to be abandoned and VRM to stop launching new Actions. The behavior described here affects only those Actions that depend on the failed Action, either directly or indirectly.

There are two status count messages emitted at the end of each VRM run as follows:

- The launch status indicates how many *preScript*, *execScript*, or *postScript* Actions were launched and, of these, how many successfully ran and how many failed (success, in this case, is the script launch itself not failing and the “done” message returning a zero (*nonerror*) status).
- The status message is a count of the number of actual simulations (UCDB files) that returned each kind of status message. The counts include the five valid UCDB status codes plus “UCDB error” (if the UCDB cannot be found or cannot be read) or “Unknown” (if the status value found in the test data record is not valid).

## Determining Pass/Fail for a Given Action

The *AnalyzePassFail* procedure uses the information at its disposal to determine whether the completed Action passed or failed. It returns either pass or fail (as a case-sensitive string).

If UCDB files are used in the regression suite, then a utility procedure is provided for reading the simulation status from the test data record. The *GetUcdbStatus* procedure takes, as its sole argument, the path to a UCDB file. It returns one of the status strings in [Table 10-1](#) on page 368, depending on the run status stored in the UCDB test data record. In the event there are multiple test data records in the UCDB file passed to *GetUcdbStatus*, the procedure retrieves the status value from the first test data record (if the suite is structured properly, however, this should never happen).

[Table 10-1](#) is the possible UCDB return values.

**Table 10-1. Possible UCDB Return Values**

UCDB Status	String Returned	Interpretation
UCDB_TESTSTATUS_OK	Passed	Simulation passed.
UCDB_TESTSTATUS_WARNING	Warning	Simulation warnings are issued.
UCDB_TESTSTATUS_ERROR	Error	Simulation errors occurred.
UCDB_TESTSTATUS_MISSING	Missing	Fatal errors occurred.
UCDB_TESTSTATUS_MERGE_ERROR	Merge error	Incomplete coverage merge occurred.
(n/a)	Unknown error	Bad status in test data record.
(n/a)	UCDB error	Cannot find/open UCDB file or cannot find test data record. Actions that define ucdbfile in the RMDB but fail to produce a UCDB file

The *AnalyzePassFail* procedure is also responsible for notifying VRM of the status discovered in the UCDB file. For each Action that represents a real simulation, the *ConvertUcdbStatus* procedure should be called with one of the above strings as the only argument. The number of times the *ConvertUcdbStatus* procedure is called with each status string is counted and displayed the end-of-run summary report. In addition, the *ConvertUcdbStatus* returns the “pass” or “fail” string to be used as the return value from the *AnalyzePassFail* procedure.

Along with these two utility procedures, the user should be able to support just about any pass/fail analysis needed for their regression suite. For example, if pass/fail is determined mostly from a comparison with some golden output but there could also be assertion failures recorded in the UCDB, the following algorithm can be adopted:

1. Read the UCDB with *GetUcdbStatus* and pass the return value to *ConvertUcdbStatus* to be counted.
2. Launch a *diff* process to compare the simulation output with the golden reference.
3. If the *diff* fails or the UCDB status returns “fail,” return the string “fail.”
4. Otherwise, return the string “pass.”

If the regression suite does not use UCDBs to collect coverage or determine pass/fail status, there is no need to call *ConvertUcdbStatus*. If the counts associated with all the possible UCDB status values are zero at the end of the regression run, then the end-of-run summary report is not emitted.

## ActionCompleted Concept

The *ActionCompleted* procedure is called after every Action and provides the user with a means to notify other processes of the completion of the Action, save off collateral files, or perform other housekeeping duties (this notification/cleanup procedure should not be used to run commands that should be, more properly, a part of the test suite itself; the purpose of this procedure is to provide a way for VRM to integrate into pre-existing flows).

The procedure is passed the pass/fail result of the *AnalyzePassFail* procedure, the path to the UCDB file (if any), and various other bits of information. It is also passed a list of files **not** to delete and can add additional files to the list to protect them from any delete-on-pass algorithm implemented in the *OkToDelete* procedure.

The *ActionCompleted* procedure returns nothing. It can copy collateral files from the working directory to long-term storage. It can also rename files in the working directory if necessary. If the UCDB file name is changed or the file is moved, then the *ucdbfile* array member should be updated. Also, if there are any files left in the working directory that should not be deleted, their names should be added to the *nodelete* array member.

Note that the notification from the *vrun* process to the Questa GUI, when used, is handled via a different mechanism. The user does not need to override the *ActionCompleted* user-definable procedure unless there are non-Questa applications that need such notification.

## OkToMerge Concept

It is assumed that in the majority of cases, VRM is used to launch simulations whose coverage data is accumulated and recorded in UCDB files (if this is not the case, this section may be ignored).

As discussed in “[Pass/Fail Analysis](#)” on page 364, the expectation of a UCDB file is communicated to VRM by the presence of a *ucdbfile* parameter in the Task definition. If such a parameter is defined, VRM uses the value of that parameter as a path (relative to the working directory of the Task, if necessary) to locate the UCDB and read the pass/fail status from the test data record.

The *OkToMerge* procedure returns a true value (1) if the coverage data in the UCDB file should be merged. The decision can be based on any of the data supplied in the *userdata* array member, the contents of any file, or any other information available from within VRM. If the UCDB file is moved or renamed in the process, the *ucdbfile* array member should be updated. If the *OkToMerge* procedure returns a true value, the UCDB file named in the *ucdbfile* array member (upon return) is automatically added to the *nodelete* list and the *MergeOneUcdb* procedure is called. Note that this procedure is only called if the *ucdbfile* is defined and non-empty so it is perfectly valid to return true in all cases (the default, unless the user overrides this procedure, is to return true in all cases).

## MergeOneUcdb Concept

The *MergeOneUcdb* procedure is passed the path to a single UCDB file and the path to a merge file. The procedure then executes the steps necessary to merge the single UCDB into the merged UCDB file. The procedure returns nothing. The default behavior of this procedure (if not overridden by the user) is to merge the UCDB defined by the value of the *ucdbfile* array member into the UCDB file defined by the value of the *mergefile* array member. The *vcover merge* executable is used without any additional options. In addition, the log output of the *vcover* executable defaults to *stdout*, and is echoed to any explicitly supplied log file. This flow could be called an *incremental merge* flow because each UCDB is merged into the common file as it is encountered.

Note that in the *incremental merge* flow, the merge is actually carried out by VRM. So if the Tasks in a particular regression run are being queued to a server grid, the *vcover merge* operations still occur in the foreground as the “done” notification for each Task is processed. This ensures sequentially of the merges at the expense of potential parallelism. If the goal is to do the merges on the server grid, the user should either place the *vcover merge* command into the *execScript* for each Task or in the *postScript* for the Group.

If both the *mergefile* and the *mergelist* parameters are defined and neither value is defined to be an empty string, the *mergefile* parameter takes precedence and the *incrementalmerge* flow is used.

Note that there is nothing precluding the user from implementing some other coverage merge scheme within the *preScript*, *execScript*, and *postScript* structure of the regression suite. These built-in merge procedures are provided to make the common case more simple.

## StopRunning Concept

In some cases, a fundamental design or testbench failure can cause a large number (if not all) of the Tasks in the regression suite to fail. In this case, early termination of the regression run can prevent processing resources from being wasted on simulations that cannot provide any additional information. By default, VRM always attempts to run every Action defined in the database. The user can override this behavior and provide a heuristic function that will terminate a regression run prematurely in the face of such catastrophic failure.

Because the definition of “catastrophic failure” depends on the design and on project policy, the heuristic for determining whether or not to continue is provided as a user-definable TCL procedure. This procedure is passed a data structure containing the number of simulations run, the number of simulations that passed, and the number of simulations that failed (Actions that failed to launch are included in the latter count). The heuristic procedure would use these numbers to compute a true/false Boolean value that is returned to VRM. If the return value is true, then further regressions are terminated. The procedure is called as each Action completes so the calculation should not be made too “heavy” and should never block execution.

Note that “stopping” the regression simply means not running or scheduling any Actions that are not already in progress. Jobs that have already been queued to a compute grid are not affected. When Actions are run in line (that is, not in the background and not on a compute grid), Actions that have not yet been launched will not run.

## StopPropagating Concept

A typical regression suite consists of a large number of inter-related Actions. Many of those Actions depend on the successful completion of previous Actions in the suite. For example, one of the uses of the *preScript* Action is to compile the Design Under Test (DUT) in preparation for running the actual simulations. If the compile step fails for some reason, the simulations will also be unlikely to succeed and, thus, executing them would be a waste of disk space and compute capacity. VRM is capable of pruning the execution graph in response to failed Actions. This is not unlike the behavior of the *make* utility when faced with a failing build step. Unless otherwise directed, the default behavior of *make* is to stop the build on the first error encountered.

VRM default behavior depends on the type of Action that failed. A *preScript* Action is generally used for design compilation or other preparatory steps. A failure here is more likely to render the entire run unsuccessful. Therefore, when the *preScript* Action of a given Group fails, VRM (by default) abandons processing of the other Actions in that Group. However, when the *execScript* Action of a given Task fails, that failure can require further analysis or cleanup that can be included in subsequent *postScript* Actions. Likewise, a failing *postScript* associated with a member Group can also require the same analysis or cleanup. Therefore, when an *execScript* or *postScript* Action fails, VRM (by default) reports the failure and continues to process the Actions in the regression suite.

This behavior can be altered via the *StopPropagating* user-defined procedure. The *StopPropagating* procedure returns a true (1) value if subsequent Actions should be abandoned and a false (0) value otherwise. The default behavior of VRM (if this procedure is not overridden) is to return a true (1) value only when a *preScript* Action fails or when the *execScript/postScript* Action of a member of a sequential Group fails. Any other status results in a false (0) value, which signals the VRM to continue processing Actions that were dependent on this Action.

## Auto-Delete Functionality

The auto-delete mechanism in VRM allows the user to configure VRM to selectively delete files from the *VRM Data* directory as a regression run progresses. This is primarily useful for those situations in which a limited amount of disk space is available and some of the collateral files generated by a given test (or by VRM to support the test execution) are no longer needed once that test finishes. For example, the user can configure a regression such that all files associated with a passing test are deleted once the test is complete and the coverage is merged into the merged UCDB file. Likewise, the user can configure a regression to preserve all UCDB and/or log files from passing tests and to preserve all files from failing tests (to facilitate reproduction of the failure) but to delete everything else.

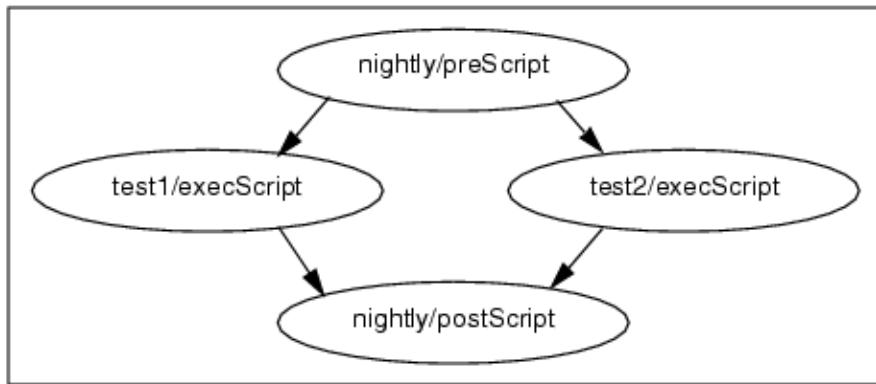
Auto-delete is enabled by overriding the *OkToDelete* user-definable procedure. By default, VRM does not delete any files or directories.

**Execution Graph** ..... [372](#)

## Execution Graph

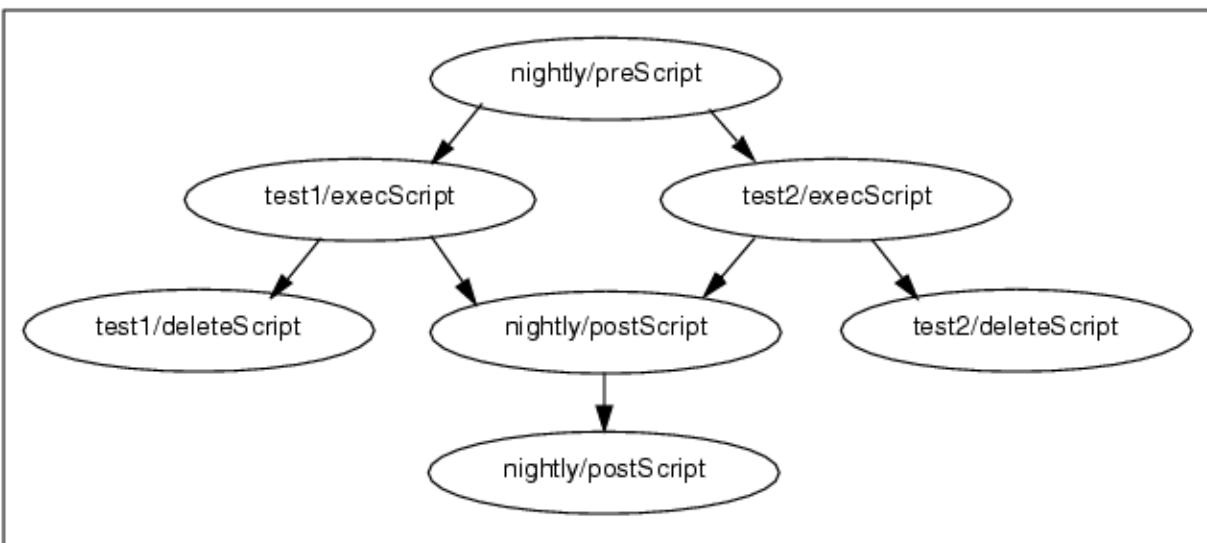
The basic execution graph for a single Group Runnable with two Task Runnable members is as shown.

**Figure 10-2. Execution Graph**



As described elsewhere, the *preScript* Action of the Group Runnable executes first. Once the *preScript* completes, the *execScript* Actions of each of the two member Task Runnables are launched (concurrently or sequentially, depending on the configuration of the RMDB), then once these two Actions complete, the *postScript* Action of the Group Runnable is launched. This is the basic VRM execution model that may be nested as necessary.

In addition to the *preScript*, *postScript*, and *execScript* Actions, VRM also generates *deleteScript* Actions. These do not represent real Actions that can appear in the RMDB. The *deleteScript* Actions are internally defined pseudo-Actions which tell the execution algorithm that there are no more real Actions pending for a particular Runnable. [Figure 10-3](#) shows the execution model including the *deleteScript* pseudo-Actions.

**Figure 10-3. Execution Graph Including deleteScript**

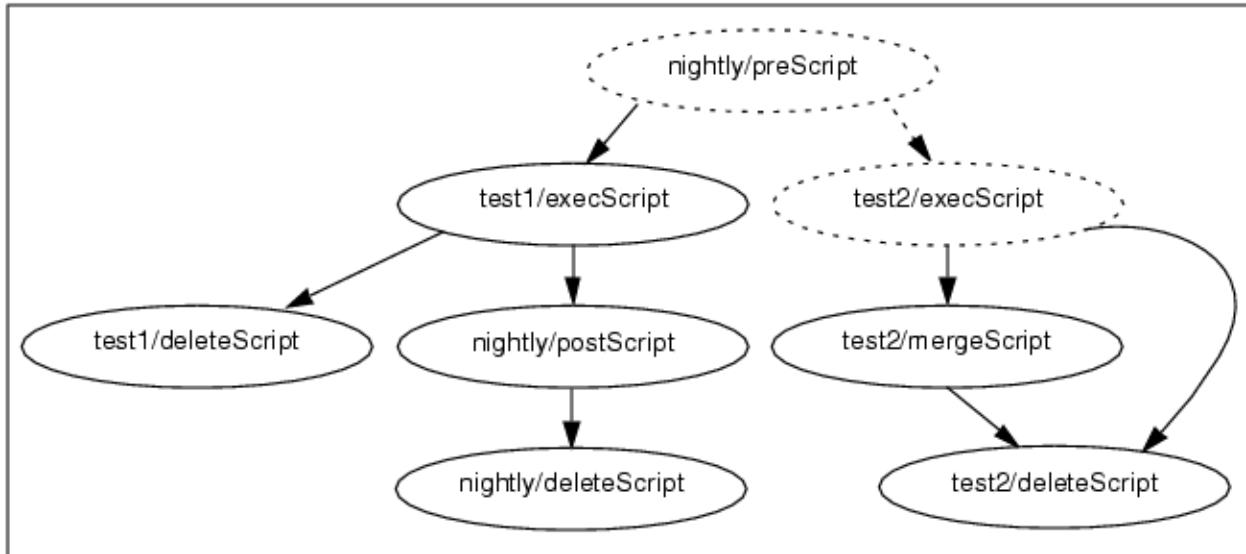
Note that the *deleteScript* pseudo-Actions for each Runnable depend on the *execScript* (or *postScript*) Action for that same Runnable (see note below) but no Action depends on any of the *deleteScript* pseudo-Actions. This implies that the *deleteScript* pseudo-Actions do not participate in the order-of-execution algorithm. The sequence in which a given set of Actions is executed depends entirely on their Group/Task relationship. The *deleteScript* pseudo-Actions become eligible for execution when the *execScript* Action (in the case of a Task) or the *postScript* Action (in the case of a Group) associated with that particular Runnable completes. When they become eligible to run, they are tracked in the same queue as all the other Actions. However, once launched, they are tracked in their own queue, in order to ensure that all the *deleteScript* pseudo-Actions complete before the regression process exits. A *deleteScript* does not have a *PostExecutionAnalysis* phase, nor does it propagate as the real Actions do. The *vrun* process managing the regression run performs the actual deletion and manages the life-cycle of these pseudo-Actions internally (they are also not reported to the Status Event Log).

Note that for a Group Runnable, the *postScript* Action and the *deleteScript* pseudo-Action also depend on the *preScript* Action associated with the same Group Runnable. These dependencies are not shown in the diagram because they are largely just for assurance. The diagram, as shown in [Figure 10-3](#), guarantees the order-of-execution described above, even without these extra dependencies.

If auto-merge or auto-triage are enabled, two additional types of Action may be generated as the regression progresses. Auto-merge can create *mergeScript* Actions and auto-triage can create *triageScript* Actions. These Actions, when needed, are connected into the execution graph as direct dependents of the *execScript* Action on whose behalf they were created. These newly added Actions also inherit the dependencies of the *execScript* Action on which they themselves depend.

Figure 10-4 illustrates the execution graph after a *mergeScript* has been added to merge the results from the test at *nightly/test2*.

**Figure 10-4. Execution Graph After *mergeScript* is Added**



Note that since *nightly/test2/mergeScript* is added to the graph by the *MergeOneUcdb* user-definable procedure as part of the *PostExecutionAnalysis* phase for *nightly/test2/execScript*, by the time the *mergeScript* Action has been added to the execution graph, the Actions *nightly/preScript* and *nightly/test2/execScript* will have already completed. Thus, they are shown in the Figure 10-4 as dotted-line nodes simply to allow visual comparison with Figure 10-3.

Note that only *mergeScript* and *triageScript* Actions can be created and each may only be created from an *execScript* Action. The *deleteScript* Action associated with any given Runnable will always become eligible to run only after all other Actions associated with that Runnable have completed.

The *nightly/preScript* and *nightly/postScript* Actions both belong to the “nightly” Runnable and, hence, are executed in the directory (%DATADIR%)/nightly. Each of the *execScript* Actions belong to one of the member Task Runnables and execute in their own directories ((%DATADIR%)/nightly/test1 and (%DATADIR%)/nightly/test2). Each of the *deleteScript* pseudo-Actions is associated with the same working directory as the *execScript* or *postScript* Action on which it depends. That makes it easy for the *vrun* process to track which directories are no longer needed, as all the pending Actions to be executed in each directory have completed by the time its associated *deleteScript* pseudo-Action becomes eligible to execute.

The net effect is that the *deleteScript* is guaranteed to be the last execution graph node associated with a particular Runnable to become eligible for execution. When a *deleteScript* pseudo-Action appears at the top of the internal execution queue, the *vrun* process knows that it is safe to schedule the deletion of any unprotected files in the directory associated with that

Runnable. The decision on whether to delete files and/or directories, and exactly how much to delete are described in [Enabling Auto-Delete](#).

Note that unlike other Actions, the *deleteScript* pseudo-Action does not run in a separate process. All *deleteScript* Actions are executed by the *vrun* process managing the regression run. This is because it is impossible to delete a directory and its contents while running a process inside of that directory. Therefore, *deleteScript* pseudo-Actions have no wrapper and no *.do* file; they generate no logs, and they are not displayed on the GUI or reported to the Status Event Log. The only way to see what auto-delete is deleting is to include the *-debug* option on the *vrun* command line.

## Coverage Merge Options

VRM supports several levels of automation of UCDB-based coverage merging. This functionality is a Questa specific value-added feature that must be enabled through the use of predefined parameters in the database. Coverage merging can also be performed manually via Action scripts defined by the user. The various coverage options available for merging coverage information follows.

<b>Assumptions</b> .....	<b>376</b>
<b>Automated Merging Example</b> .....	<b>376</b>
<b>Post-execution Processing</b> .....	<b>378</b>

## Assumptions

The following assumptions are made about the environment and the flow when merging coverage data under VRM:

- Each regression test (that is, individual simulation run) generates a single UCDB file containing pass/fail and coverage information.
- Task Runnables are associated one-to-one with regression tests. That is,
  - Each Task Runnable for which the *ucdbfile* parameter expands to a non-empty string executes exactly one test that results in exactly one UCDB file.
  - Only Task Runnables (*execScript* Actions) run simulations and/or generate test-level UCDB files (Groups, however, can generate merged UCDB files from the UCDB files generated by member Tasks and/or Groups).

This does not mean that VRM cannot be used in other configurations or even in non-Questa flows. But if the above assumptions are not met, automated merging (and possibly UCDB-based automated pass/fail checking) as described herein is not directly supported.

Note that so long as the UCDB files generated contain the requisite information and can be read via the UCDB API, no assumption is made about the source of the files. It is possible for a 3rd-party simulation application to generate a UCDB for the sole purpose of communicating detailed pass/fail information to VRM.

## Automated Merging Example

The following general example is used to illustrate the various levels of automated merging. The following RMDB database does not include any provision for coverage merging, as that is added (in various ways) throughout the remainder of this chapter.

### Figure 10-5. Automated Merging Example

```

<rmdb>
    <runnable name="nightly" type="group">
        <parameters>
            <parameter name="testname">test (%seed%)</parameter>
            <parameter name="ucdbfile">(%testname%).ucdb</parameter>
        </parameters>
        <members>
            <member>test1</member>
            <member>test2</member>
            <member>test3</member>
        </members>
        <preScript launch="vsim">
            <command>vlib work</command>
            <command>vlog -sv -cover bces (%RMDBDIR%)/top.sv</command>
        </preScript>
        <execScript launch="vsim">
            <command>vsim -c top -lib (%DATADIR%)/nightly/work -sv_seed
                (%seed%)</command>
            <command>run -all</command>
            <command>coverage attribute -name TESTNAME -value
                (%testname%)</command>
            <command>coverage save (%ucdbfile%)</command>
        </execScript>
    </runnable>
    <runnable name="test1" type="task">
        <parameters>
            <parameter name="seed">123</parameter>
        </parameters>
    </runnable>
    <runnable name="test2" type="task">
        <parameters>
            <parameter name="seed">456</parameter>
        </parameters>
    </runnable>
    <runnable name="test3" type="task">
        <parameters>
            <parameter name="seed">789</parameter>
        </parameters>
    </runnable>
</rmdb>

```

This regression suite consists of three tests. A single source file, located in the same directory as the RMDB database, is compiled into a work library. The top-level module (top) is executed with one of three fixed seeds. This module presumably executes a constrained random test against the design. It then generates a UCDB file in the working directory of the test whose name is based on the value of the random seed.

The goal of the selected merge option is to generate a single UCDB file that contains the combined coverage results of the three tests.

## Post-execution Processing

As each Action script reports completion (or times out), VRM does some internal post-processing on the results. With a few minor exceptions, the post-processing steps can all be controlled via user-supplied TCL code. This is because the post-processing is actually performed by calling a series of user-definable TCL procedures, each of which has a predefined default behavior. In theory, there are an infinite number of possible behaviors for this post-processing (which necessarily includes the coverage merging step). This chapter covers the default behavior provided within VRM itself (without any TCL-based user overrides).

The following is an overview of the post-processing steps performed by VRM for each Action script as it reports completion:

1. The *AnalyzePassFail* (see “[Pass/Fail Analysis](#)” on page 364) procedure is called to determine whether or not the regression test passed.
2. The *ActionCompleted* (see “[ActionCompleted Concept](#)” on page 369) procedure is called to notify 3rd party applications of the completion.
3. The *OkToMerge* (see “[OkToMerge Concept](#)” on page 369) procedure is called to determine if the coverage results from this Action should be merged into the target merge file.
4. If the *OkToMerge* procedure returns true, then the *MergeOneUcdb* (see “[MergeOneUcdb Concept](#)” on page 370) procedure is called to do the actual merging.
5. The *OkToDelete* (see [OkToDelete](#)) procedure is called to determine if the VRMDATA directory for the Action should be deleted.
6. If the *OkToDelete* procedure returns true, then the working directory for the Action is deleted.
7. The *StopRunning* (see “[StopRunning Concept](#)” on page 370) procedure is called to determine if the entire regression run should be abandoned due to errors.
8. The *StopPropagating* (see “[StopPropagating Concept](#)” on page 371) procedure is called to determine if downstream Actions should be abandoned due to errors.

These steps are applied to each Action script although many of them do nothing unless the Action is an *execScript* (therefore, represents a regression test). The reason for treating all Actions the same in this regard is to allow the user more flexibility in controlling VRM behavior via the user-definable procedures. The above list is simply for quick reference (the user-definable procedures and the method for overriding them are covered “[User-definable Procedures and Loading of Arbitrary TCL Code](#)” on page 412).

# Coverage Merging Use Models

By default, VRM supports three levels of automation for coverage merging, as described in the following sections.

<b>Do-It-Yourself (DIY) Merging .....</b>	<b>379</b>
<b>List-based Merging .....</b>	<b>384</b>
<b>Incremental Merging.....</b>	<b>386</b>
<b>Queued Merging .....</b>	<b>388</b>
<b>Queued Merging Example .....</b>	<b>390</b>
<b>Auto-Merge Tips and Traps .....</b>	<b>392</b>
<b>Automatic Deletion of Merge/Triage Files .....</b>	<b>394</b>
<b>Auto-merge and Auto-triage Use of -inputs Argument .....</b>	<b>395</b>
<b>Testplan Auto-import .....</b>	<b>395</b>
<b>Enabling and/or Disabling Coverage Merge Flows .....</b>	<b>396</b>
<b>Automated Results Analysis .....</b>	<b>398</b>
<b>UCDB File “clubbing” in Automated Merge and Automated Triage .....</b>	<b>400</b>
<b>Automated Trend Analysis .....</b>	<b>402</b>

## Do-It-Yourself (DIY) Merging

Under the DIY merging flow, the user is responsible for calling *vcover merge* with the proper arguments at the proper time. Those *execScript* Actions that represent regression tests (that is, those which run simulations) should always generate a UCDB file at the location designated by the *ucdbfile* parameter in order to support automated pass/fail checking. However, VRM itself will not perform any automated merging. Any coverage merging is performed by the user’s code in one of two ways.

- One or more of the user-supplied Action scripts (either the *execScript* on behalf of an individual Task or the *postScript* on behalf of a Group) could launch the *vcover merge* utility in order to merge coverage results.
- The users can override the *ActionCompleted*, *MergeOneUcdb*, and/or *RegressionCompleted* user-definable procedures to implement the merging scheme of their choice.

Beyond the pass/fail checking and calling the user-definable procedures in the prescribed sequence, VRM will not participate in the merging process.

## Assumptions and Detailed Behavior

- Each Task Runnable identifies a *ucdbfile* parameter (which is usually inherited from a lineal parent).

- The value of this parameter is a path to the UCDB file generated by the regression test represented by the Task.
- If a relative path is given, it is assumed to be relative to the working directory for the Task.
- If the Task intentionally does not generate a UCDB file, then the *ucdbfile* parameter must be undefined or empty or the pass/fail checking reports the test as failed.
- Merge algorithm
  - Each *execScript* dumps a UCDB file to the location specified by the *ucdbfile* parameter, and (optionally) calls *vcover merge* to merge the *ucdbfile* to a user-defined merge file (the location of which may or may not be parameterized). The individual UCDB files generated in a given Group of Tasks could also be merged in a single operation as part of the *postScript* for the Group.
  - A Group *postScript* Action can optionally merge intermediate merge files into higher-level merge files, as dictated by the hierarchy of the regression suite (or the user can elect to merge all the test-level UCDB files into a single merge file as part of the *postScript* Action associated with the top-most Group in the regression suite).

All the user-definable override procedures (*OkToMerge*, *MergeOneUcdb*, and so on) are called in sequence. However, in DIY merging mode, these procedures do nothing by default. Instead of calling *vcover merge* from the Action scripts, it is possible to override these user-definable procedures such that they invoke the merge operation from within VRM. In fact, the user can re-implement incremental merging mode via user-definable procedures if desired.

The user Action script (or override procedure) is responsible for determining whether or not the coverage from a given simulation is eligible to be merged into the merge file (for example, if failed tests are to be omitted from the merge file).

The default VRM behavior, unless otherwise specified or enabled, assumes the user will handle **all** post-simulation processing, including coverage merging, in the Action scripts or by way of user-defined TCL override procedures. By default, the contents of the *ucdbfile* parameter is only used to extract the pass/fail status for the associated simulation for status reporting purposes.

Note that the locking mechanism in *vcover merge* protects the user from collisions due to multiple concurrent simulations completing at once. However, this is not as efficient as having the *postScript* for the uppermost Group in the regression suite perform the merge in one step. In order to accomplish a one step merge, the *OkToMerge* user-definable procedure can be modified to write the absolute path of the UCDB file to a common location where it can be picked up for the *vcover merge* command.

### DIY Merge Example

There are two ways in which DIY merging can be implemented in a database. The first is via the Action scripts, calling *vcover merge* either after a simulation or at the end of the regression run.

For example, the user can modify the *execScript* used by the Task Runnables to invoke *vcover merge* after each simulation as shown in [Example 10-6](#).

**Figure 10-6. DIY Merge Invoking vcover merge After Each Simulation**

```

<rmdb>
  <runnable name="nightly" type="group">
    <parameters>
      ...
      <parameter name="mfile">(%DATADIR%)/nightly/merge.ucdb</parameter>
    </parameters>
    ...
    <preScript>
      ...
      <command>file delete -force (%mfile%)</command>
    </preScript>
    <execScript launch="vsim">
      <command>vsim -c top -lib (%DATADIR%)/nightly/work -sv_seed
        (%seed%)</command>
      <command>run -all</command>
      <command>coverage attribute -name TESTNAME -value
        (%testname%)</command>
      <command>coverage save (%ucdbfile%)</command>
      <command>if {[file exists (%mfile%)]} {</command>
        <command> vcover merge (%mfile%) (%mfile%
          (%ucdbfile%))</command>
        <command>} else {</command>
          <command> file copy (%ucdbfile%) (%mfile%)</command>
        <command>}</command>
      </execScript>
    </runnable>
    ...
  </rmdb>

```

Note that the location of the merge file is independent of the working directory of any specific test. That means the Tasks that represent simulations can be located at any hierarchical level in the regression topology and the *vcover merge* command will still find the right merge file. Also note that if the merge file does not already exist, the simulation in question must be the first one to have completed and no merge is actually necessary (this also assumes the merge file is deleted at the start of the regression run, as shown in [Example 10-6](#)).

Also note that if the location of the merge file is parameterized, as in [Example 10-6](#), care must be taken not to use the keyword *mergefile*, as the presence of that keyword in the RMDB database trigger the VRM automated merge process.

Another approach might be to add a *postScript* to the top-level Group Runnable (nightly) that merges all the test-level UCDB files in one pass as shown in [Example 10-7](#).

### Figure 10-7. DIY Merge Adding postScript to Top-level Group Runnable

```
<rmdb>
  <runnable name="nightly" type="group">
    <parameters>
      ...
      <parameter name="mfile">(%DATADIR%)/nightly/merge.ucdb</parameter>
    </parameters>
    ...
    <execScript launch="vsim">
      <command>vsim -c top -lib (%DATADIR%)/nightly/work -sv_seed
        (%seed%)</command>
      <command>run -all</command>
      <command>coverage attribute -name TESTNAME -value
        (%testname%)</command>
      <command>coverage save (%ucdbfile%)</command>
    </execScript>
    <postScript launch="vsim">
      <command>vcover merge (%mfile%) test*/*.ucdb</command>
    </execScript>
  </runnable>
  ...
</rmdb>
```

Note that this approach requires there are no UCDB files under the `(%DATADIR%)/nightly` tree other than the test-level UCDB files. It also assumes all the UCDB files should be merged regardless of the pass/fail status of the corresponding test. Yet another approach might be to create a directory under `(%DATADIR%)/nightly` to hold the UCDB files from regression tests that passed and to have the *execScript* copy the UCDB into this directory. The *postScript* can then merge all the files in this directory after the simulations have been completely executed. This approach also avoids the inefficiency inherent in the *vcover merge* locking mechanism.

The other way to accomplish DIY merging is through the use of user-defined TCL override procedures. There are probably an infinite number of ways in which this can be done but one of the simplest is to check the pass/fail status in the *OkToMerge* procedure, add the passing test-level UCDB files to a list as part of the *MergeOneUcdb* procedure, and then invoke the *vcover merge* process in the *RegressionCompleted* procedure. [Example 10-8](#) shows one way of doing this.

**Figure 10-8. DIY Merge User-defined TCL Override Procedure**

```

<rmdb loadtcl="diy-merge">
  <usertcl name="diy-merge">
    proc OkToMerge {userdata} {
      upvar $userdata data

      if {[string equal $data(ACTION) "execScript"]} {
        return 0 ;# the Action is not a regression test
      }

      if {[string equal $data(ucdbfile) ""]} {
        return 0 ;# the UCDB filename is not specified
      }

      if {[string equal $data(passfail) "pass"]} {
        return 0 ;# the test did not pass, do not merge
      }

      return 1 ;# otherwise, approve the UCDB file for merging
    }

    proc MergeOneUcdb {userdata} {
      upvar $userdata data

      global MyUcdbFiles ;# user-maintained list of files to merge

      lappend MyUcdbFiles $data(ucdbfile) ;# store list of passing UCDB
          files
    }

    proc RegressionCompleted {userdata} {
      upvar $userdata data

      global MyUcdbFiles ;# user-maintained list of files to merge

      if {[llength $MyUcdbFiles] > 1} {
        set mergefile "merge.ucdb"

        set cmd [list [file join $::env(MODEL_TECH) vcover] merge -out
            $mergefile]

        # If merge file exists, do an "appending" merge

        if {[file exists $mergefile] && [file readable
            $mergefile]} {
          lappend cmd $mergefile
        }

        # Issue the "vcover merge" command, listing all passed UCDB files

        if {[catch {eval [concat exec $cmd $MyUcdbFiles]} error]} {
          echo "Error: Cannot merge to '$mergefile': '$error'"
        }
      }
    }
  </usertcl>

```

```
</rmdb>
```

The user-defined override procedures are placed in a *usertcl* element that is marked for auto-loading. So the overridden behavior becomes part of the default VRM behavior whenever this RMDB database is used. The code shown here is overly simplistic in order to illustrate the point. The behavior can be further customized to meet the user's needs and/or the regression environment in which VRM is being used.

Note that [Example 10-8](#) (overriding the user-definable procedure) implements a flow very similar to [List-based Merging](#).

## List-based Merging

The list-based merging flow is similar to the DIY flow in that each simulation produces a UCDB file and the *postScript* Action of a higher-level Group must manually invoke the *vcover merge* command. The difference is that the *postScript* Action does not have to search for the UCDB files with a wildcard, as *vrun* maintains a list of UCDBs from tests that are eligible for merging.

### Assumptions and Detailed Behavior

The assumptions are the same as for DIY merging. In addition, the user designates a file (usually within the *VRMDATA* directory tree) and passes the path to that file to *vrun* via the *mergelist* parameter. The assumptions and behavior are summarized by the following:

- Each Task Runnable identifies a *ucdbfile* parameter (which is usually inherited from a lineal parent).
  - The value of this parameter is a path to the UCDB file generated by the regression test represented by the Task.
  - If a relative path is given, it is assumed to be relative to the working directory for the Task.
  - If the Task intentionally does not generate a UCDB file, the *ucdbfile* parameter must be undefined or empty or the pass/fail checking reports the test as failed.
- Each Task Runnable (or an ancestor Group) defines a *mergelist* parameter that points to a plain test file.
  - This file is cleared upon the start of each regression run.
- Merge algorithm
  - Each *execScript* dumps a UCDB file to the location specified by the *ucdbfile* parameter, and (optionally) calls *vcover merge* to merge the *ucdbfile* to a user-defined merge file (the location of which may or may not be parameterized). The

individual UCDB files generated in a given Group of Tasks can also be merged in a single operation as part of the *postScript* for the Group.

- If the *OkToMerge* user-definable procedure returns a true value, the *MergeOneUcdb* procedure is called which saves the full path of the UCDB file in question to the text file specified by the *mergelist* parameter.
- A Group *postScript* Action can optionally merge intermediate merge files into higher-level merge files, as dictated by the hierarchy of the regression suite. The contents of the text file pointed to by the *mergelist* parameter can be used as a list of input files to the *vcover merge* command.

Note that if the *mergefile* parameter is defined for any given *execScript* Action (thereby triggering the auto merge algorithm), the *mergelist* parameter is ignored.

### List-based Merge Example

List-based manual merging is most commonly used when the coverage merge is handled by a single *postScript* Action at the end of the simulation run as shown.

**Figure 10-9. List-based Merge postScript Action at end of Simulation**

```

<rmdb>
  <runnable name="nightly" type="group">
    <parameters>
      ...
      <parameter name="mergelist">(%DATADIR%)/nightly/mergelist
      </parameter>
    </parameters>
    ...
    <execScript launch="vsim">
      <command>vsim -c top -lib (%DATADIR%)/nightly/work -sv_seed
      (%seed%)</command>
      <command>run -all</command>
      <command>coverage attribute -name TESTNAME -value
      (%testname%)</command>
      <command>coverage save (%ucdbfile%)</command>
    </execScript>
    <postScript launch="vsim">
      <command>vcover merge -out (%DATADIR%)/nightly/merge.ucdb
      -inputs (%mergelist%)</command>
    </execScript>
  </runnable>
  ...
</rmdb>

```

Since *vrun* automatically clears the text file pointed to by the *mergelist* parameter, there is no reason for the *preScript* Action to delete the file. Any tests that pass the *OkToMerge* test (by default, any test with a valid UCDB containing a test data record with a passing status) will cause the path to the UCDB file to be written to the “mergelist” file under the working directory for the “nightly” Runnable. At the end of the regression run, the *postScript* of the “nightly” Runnable uses this file as the input list to a *vcover merge* command, thus merging all passing tests into a single file.

## Incremental Merging

The incremental merging flow automates one of the merging schemes mentioned in the previous section. This semi-automated use model serves mostly as a user convenience. Under this flow, parameters are used to point to the UCDB file for each individual regression test and to a target merge file. As before, the *execScript* Actions that represent regression tests (those which run simulations) each generate a UCDB file at the location designated by the *ucdbfile* parameter in order to support automated pass/fail checking. VRM then calls *vcover merge* as each *execScript* completes in order to merge the coverage of the individual tests into the target merge file.

### Assumptions and Detailed Behavior

- Each Task Runnable identifies a *ucdbfile* parameter and a *mergefile* parameter (both parameters are usually inherited from a lineal parent).
  - The *ucdbfile* parameter points to the UCDB file generated by the regression test represented by the Task.
    - If a relative path is specified, it is assumed to be relative to the working directory for the Task.
    - If the Task intentionally does not generate a UCDB file, then the *ucdbfile* parameter must be undefined or empty.
  - The *mergefile* parameter points to the target of the merge operation.
    - The path specified can be relative to the working directory for the Task or, more likely, it is an absolute path, possibly based on one of the VRM built-in path parameters (such as *VRMDATA*).
- Merge algorithm (*execScript*)
  - Each *execScript* dumps a UCDB file to the location specified by the *ucdbfile* parameter (expanded from the point of view of the Task).
  - The pass/fail status is then checked by the *AnalyzePassFail* user-definable procedure and sent to the *OkToMerge* user-definable procedure.
  - If the *OkToMerge* procedure returns true, then the *MergeOneUcdb* user-definable procedure is called to merge the test-level UCDB into the target merge file.
  - The *MergeOneUcdb* procedure checks for a non-empty *mergefile* parameter. If found, the *MergeOneUcdb* procedure does the following:
    - Constructs and executes a *vcover merge* command to merge the *ucdbfile* into the *mergefile* (also expanded from the point of view of the Task).
- Merge algorithm (*postScript*)

- When the *OkToMerge* procedure is called on a *postScript* Action (in incremental merging mode), it always returns false; therefore, preventing any automated merging at the Group level.
- The user's *postScript* optionally merges the merge file of the current Group to the merge file of a parent Group, if any.

There are other behavioral points common to the incremental and queued forms of auto-merging. Refer to “[Auto-Merge Tips and Traps](#)” on page 392 for more details.

### Incremental Merge Example

In order to use incremental merging mode, all that is necessary is to supply a *mergefile* name. Note that the use of a *mergefile* parameter is in addition to a *ucdbfile* parameter, as described earlier in this section.

**Figure 10-10. Incremental Merge with *mergefile* Parameter**

```
<rmdb>
  <runnable name="nightly" type="group">
    <parameters>
      ...
      <parameter name="mergefile">(%DATADIR%)/nightly/merge.ucdb
    </parameter>
  </parameters>
  ...
</runnable>
...
</rmdb>
```

Note in this case that the target merge file is located at an absolute path, within the working directory of the top-most Group Runnable. Therefore, no intermediate merge commands are needed. As each of the three regression tests complete, a *vcover merge* command is issued to merge the test-level UCDB file into the target merge file. Assuming the tests complete in order and there is no merge file to start with, the commands issued would look something like the following:

```
vcover merge -out /path/to/nightly/merge.ucdb
/path/to/nightly/test1/test123.ucdb

vcover merge -out /path/to/nightly/merge.ucdb
/path/to/nightly/merge.ucdb/path/to/nightly/test1/test456.ucdb

vcover merge -out /path/to/nightly/merge.ucdb
/path/to/nightly/merge.ucdb/path/to/nightly/test1/test789.ucdb
```

Note also that the first command does not include the merge file as an input since the assumption is that the merge file did not yet exist. The end result of these commands meets the goal of having a single file, (%DATADIR%)/nightly/merge.ucdb, containing the coverage results of the three regression tests.

## Queued Merging

The queued merging flow is the default if both the *ucdbfile* and the *mergefile* parameters are defined for a given *execScript* Action.

Queued merging is a variation on incremental merging, and the default auto-merge algorithm. With incremental merging, a separate *vcover merge* command is launched for each regression test and the individual *vcover* processes compete for access to the target merge file. With queued merging, VRM accumulates a list of UCDB files to be merged to a particular target merge file and then manages the launching of one or more *vcover merge* commands as follows:

- The UCDB files from multiple regression tests that finish within a small time window are merged as part of a single *vcover merge* operation.
- The overhead of file locking on the target merge file is avoided.

This merge methodology can improve the overall merge performance by taking advantage of spare CPU cycles to get a head start on the merge process while some of the simulations are still running.

### Assumptions and Detailed Behavior

The initial steps of this behavior are identical to those of the incremental merging flow. Only the merging process is different.

- Each Task Runnable identifies both a *ucdbfile* parameter and a *mergefile* parameter (both parameters are usually inherited from a lineal parent).
  - The *ucdbfile* parameter points to the UCDB file generated by the regression test represented by the Task. If the Task intentionally does not generate a UCDB file, the *ucdbfile* parameter must be undefined or empty.
  - The *mergefile* parameter points to the target of the merge operation. The value of this parameter can be a path relative to the working directory for the Task or it can be an absolute path, possibly based on one of the VRM built-in path parameters (such as *VRMDATA*).
- Merge algorithm (*execScript*)
  - Each *execScript* dumps a UCDB file to the location specified by the *ucdbfile* parameter (expanded from the point of view of the Task).
  - The pass/fail status is then checked by the *AnalyzePassFail* user-definable procedure and sent to the *OkToMerge* user-definable procedure.
  - If the *OkToMerge* procedure returns true, then the *MergeOneUcdb* user-definable procedure is called to merge the test-level UCDB into the target merge file.

- The *MergeOneUcdb* procedure (when called in queued merge mode) determines whether a merge to the specified target merge file (as defined by the *mergefile* parameter) is currently ongoing.
  - If no such merge is ongoing and the target merge file does not yet exist, then the UCDB file to be merged is simply copied to the *mergefile* location (this prevents a file locking race condition).
  - If no such merge is ongoing and the target merge file exists, then a *mergeScript* Action is scheduled to perform the merge. This Action is treated like any other Action in the RMDB database, except that the downstream dependencies of the *execScript* are copied to the *mergeScript* in order to prevent the next Action from starting before the merge associated with the prior Action is completed.
  - If a merge to the target file is currently ongoing, the UCDB file associated with the just-completed *execScript* is queued for execution once the ongoing *mergeScript* has completed.
- Merge algorithm (*postScript*)
  - When the *OkToMerge* procedure is called on a *postScript* Action (in incremental merging mode), it always returns false; therefore, preventing any automated merging at the Group level.
  - The user's *postScript* optionally merges the merge file of the current Group to the merge file of a parent Group, if any.

Each unique *mergefile* parameter value identifies a unique queue. Therefore, while the *vrun* algorithm prevents multiple merges from running against the target merge file *A.ucdb*, another merge process can be running against a different UCDB file (possibly *B.ucdb*) at the same time. The basic plan is to have only one *vcover merge* process writing to any given target merge file at any one time. While this merge process is running, any test-level UCDB files that become available (because of regression test completions) are queued until the currently running merge process has completed. Doing this incurs the following two performance benefits:

- By merging multiple test-level UCDB files at one time, the overhead of reading and rewriting the target merge file is reduced.
- By launching no more than one *vcover merge* process against any single target merge file at any one time, the overhead associated with waiting for an existing file lock to be released is avoided.

One bit of bookkeeping is necessary. The execution model of VRM guarantees that when the *postScript* Action of any given Group is executed, the Action scripts associated with the members of that Group (*execScript* Actions in the case of Tasks or *postScript* Actions in the case of sub-Groups) have completely executed (if the final product of the Group member is a UCDB file, then the *postScript* of the Group might want to use that UCDB file as the input to another higher-level merge). Therefore, the queued merging algorithm must modify the execution graph on-the-fly. When the test-level UCDB files from one or more regression tests

are compiled into a *vcover merge* command, a node representing the merge process itself is inserted into the graph and the vertices from the associated Task Runnables to the downstream *postScript* node are copied to the node associated with the *vcover merge* process. When that process completes, the vertices are removed; only after all the test-level UCDB files generated by Tasks that are members of the given Group are merged into the target merge file will the *postScript* for the Group become eligible to run. Note that this is true in general, even if the *postScript* for the Group does not make use of the target merge file at all and even if multiple Groups point to the same target merge file.

For example, the user can have a top-level Group (*nightly*) made up of three sub-Groups (*group1*, *group2*, and *group3*), each of which has three Task members. The number of regression tests in this case is nine (assuming a one-to-one correspondence between Task Runnables and simulations). The user can decide to merge all nine regression tests to a single merge file at the top-level of the scratch directory tree. The *mergefile* parameter can be defined in the top-level Group Runnable, so that all nine Task Runnables can share the same parameter value. In this case, the *postScript* Action for *group2* would initially be dependent on the three *execScript* Actions that are listed as members of the *group2* Runnable. However, if the UCDB files generated by these three *execScript* Actions were sent to a single *vcover merge* process, the *postScript* Action for the *group2* Runnable also becomes dependent on the completion of that *vcover merge* process. Once the three UCDB files generated by the *execScript* Actions under the *group2* Group are merged successfully into the target merge file, the *postScript* for *group2* becomes eligible to run. This is true even if the merging of some of the UCDB files resulting from the execution of *execScript* Actions under the two other sub-Groups are still pending (or even if the *execScript* Actions themselves are still pending).

There are other behavioral points common to both the incremental and queued forms of auto-merging. Refer to “[Auto-Merge Tips and Traps](#)” on page 392 for more details.

## Queued Merging Example

Since the queued merging model differs from the incremental merging model only in the internal implementation details, the examples given for incremental merging are equally useful here.

### Multi-level Merging

Sometimes it is convenient to merge tests from multiple subtrees of the regression suite topology to individual merge files, and then to merge those merge files together in a second-level merge process. Sometimes doing so is necessary in order to translate a Group of unit tests into a system-test hierarchy or vice versa.

The incremental merge and queued merge auto-merge flows support multi-level merging. An example is available in “[Multi-level Auto-merge Example](#)” on page 403.

### Pass/Fail Analysis and Auto-Merging

Fully-automated coverage merging, especially in the multi-level case, can support merges of UCDB files that themselves result from merges of multiple simulation-level UCDB files. Only UCDB files with a single test data record are eligible for pass/fail checking. Any UCDB file that has more than one test data record is assumed to be a merge file and is considered to have “passed” for purposes of pass/fail analysis. This prevents the second-level and subsequent merges from appearing to have failed if the first test data record in the lower-level merge file reveals a failing status.

## Auto-Merge Tips and Traps

The following two subsections discuss tips and traps of auto-merging.

Auto-Merging and Delete-on-Pass .....	392
Sorting Coverage Information Based on Test Status .....	392

### Auto-Merging and Delete-on-Pass

VRM implements a pair of user-definable functions that can be used to implement a “delete-on-pass” algorithm (where intermediate files generated by passing tests can be deleted on-the-fly to save disk space). When auto-merging (incremental or queued) is enabled, the *MergeOneUcdb* post-execution step only queues the *mergeScript* Action for later execution. If the *OkToDelete* user-defined procedure returns a true value, the script files necessary to perform the merge can be deleted before the merge itself has been completed. The *OkToDelete* user-defined procedure should not be overridden if auto-merging is enabled.

### Sorting Coverage Information Based on Test Status

The *OkToMerge* user-defined procedure can be overridden to control whether or not a given UCDB is to be merged into the *mergefile* target (for example, based on whether the test passed or failed). But since *MergeOneUcdb* shares the same data array parameter as *OkToMerge*, the *OkToMerge* procedure can actually be overridden to do the following:

- Provide a yes/no return value indicating whether the UCDB in question should be merged at all.
- Modify the *mergefile* element in the data array in order to control the actual path of the target merge file into which the UCDB in question should be merged.

Therefore, by overriding the *OkToMerge* user-definable parameter, it should be possible to create multiple merge files, one for passing tests and one for failing tests.

Since the auto-merge *mergeScript* Actions must all complete before the Group *postScript* will run, the *postScript* can be used to create a single merge file from the two auto-merge targets that

can be fed into the failure triage tool. Consider, for example, the following RMDB database fragments:

```

<rmdb version="1.1" loadtcl="split-merge">
    <usertcl name="split-merge">
        proc OkToMerge {userdata} {
            upvar $userdata data

            if {[! [string equal $data(ucdbfile) {}]} {
                if {[string equal $data(passfail) pass]} {
                    set data(mergefile) [file join [file dirname $data(mergefile)] 
                        passed.ucdb]
                } else {
                    set data(mergefile) [file join [file dirname $data(mergefile)] 
                        failed.ucdb]
                }
                return 1 ;# merge UCDB if it exists
            }
            return 0;
        }
    </usertcl>
    ...
    <runnable name="nightly" type="group">
        ...
        <parameters>
            ...
            <parameter name="mergefile">(%DATADIR%)/merge.ucdb</parameter>
        </parameters>
        ...
        <postScript launch="vsim" mintimeout="60">
            <command>vcover merge (%mergefile%) (%DATADIR%)/passed.ucdb
                (%DATADIR%)/failed.ucdb</command>
            <command>triage dbfile (%DATADIR%)/failed.ucdb</command>
            <command>triage report</command>
            <command>coverage open (%mergefile%)</command>
            <command>coverage attribute -test * -name TESTNAME -name
                TESTSTATUS</command>
        </postScript>
    </runnable>
</rmdb>
```

In this example, the “nightly” Group (which you can assume is the top-level Group of the regression suite) defines a single target merge file at `/path/to/VRMDATA/merge.uccb` via the `mergefile` parameter. A `ucdbfile` parameter (not shown here) defines the name of each leaf-level UCDB. As each *execScript* finishes, the *vrun* post-execution analysis algorithm calls the *OkToMerge* user-definable procedure that is overridden in the *usertcl* element above. The modified *OkToMerge* procedure checks the pass/fail status of the *execScript* UCDB file (assuming one is defined) and modifies the `mergefile` element of the data array to point to one of two separate merge files. Since the value of the original `mergefile` parameter (from the RMDB file) is available in the data array passed to *OkToMerge*, you can use that path to locate the target merge files (`passed.ucdb` and `failed.ucdb`) in that same directory.

When the default *MergeOneUcdb* procedure is called, the UCDB files from passed tests are queued to be merged into the `passed.ucdb` target merge file while, at the same time, any UCDB

files from failed tests are queued to be merged into the *failed.ucdb* target merge file. Because separate queues are maintained for each target merge file, a merge to *failed.ucdb* can be running concurrently with a merge to *passed.ucdb* without interference.

By the time the *postScript* from the “nightly” Group is executed, there can be either a *passed.ucdb*, a *failed.ucdb*, or both in the top of the VRMDATA directory tree. The *postScript* commands shown above make one final merge to join all the tests together for browsing in the GUI. But only the UCDB with the failed tests is passed to the triage application. By expanding on this basic theme, the user can create any combination of UCDB files for whatever purpose that may be required.

Note that the *vcover merge* command in the *postScript* does not take into account the case where all the tests pass or all the tests fail. This can be handled with TCL *if* statements or by generating an empty UCDB for both target files in the *preScript* of the “nightly” Group, whichever is easier.

## Automatic Deletion of Merge/Triage Files

The auto-merge and auto-triage algorithms can detect the first merge/triage operation directed at a specific *mergefile* or *triagefile* and to detect whether a file already exists at the target location. If so, then the existing file is assumed to be from a previous regression run and is deleted/cleared prior to the first merge/triage operation.

In the case of auto-merge, the merge file is simply deleted. In the case of auto-triage, the following command is used to clear the database:

```
triage dbfile -name (%triagefile%)
-clear
```

If the merge and/or triage results of the current regression run are to be intentionally combined with the merge and/or triage results from the previous run, the deletion/clearing can be disabled via options on the *vrun* command line. The *-noautomergedelete* option causes auto-merge to not delete the existing *mergefile*. In addition, the verification plan is not automatically imported (see [Testplan Auto-import](#) below). Likewise, the *-noautotriageclear* option causes auto-triage to not delete the existing messages from the target Triage DataBase (TDB) file.

Note that when the auto-merge algorithm is enabled, it assumes that all coverage merging is done automatically. In this case, the user should refrain from using any commands that will alter the merge file, such as manually merging some UCDB files and relying on the auto-merge algorithm to merge others. If there is a need to bypass auto-merge for some part of the regression suite, auto-merge should be disabled completely and all merging should be done via Action script commands. The same recommendation applies to auto-triage as well, although auto-triage and auto-merge are independent and are not required to both be enabled in any given run.

The *mergeScript* and *triageScript* Actions go to separate named queues (called *merge* and *triage*, respectively) in order to make sure auto-merge and auto-triage are not blocked by other pending Actions.

The automatic deletion of the *mergefile*, and/or the automatic clearing of the TDB file, only happens once per regression run. If tests are rerun as a result of automatic failure rerun, the results from those tests are merged into the *mergefile* or inserted into the *triagefile* as though they were a part of the same regression run (automatic failure rerun is simply a second pass within the same regression run). Since coverage from failing tests is not usually merged into the *mergefile*, this presents no additional anomalies. However, it may be prudent to add the *-noautotriage* command to the *-rerun* option argument in order to prevent message from the failing test from being loaded a second time should the test fail on rerun.

## Auto-merge and Auto-triage Use of -inputs Argument

When queued auto-merge and/or queued auto-triage are enabled in a regression run, UCDBs representing Actions that completed during an active merge or triage operation are queued and clubbed together in a subsequent merge or triage operation.

By default, the list of UCDB files is written to a text file, which is passed as the value of the *-inputs* argument to the merge or triage command.

The list of UCDB files is created in the same directory as the corresponding script file, such as *mergeScript.do* or *triageScript.do*. Given the names of these .do files, the default name for list file is either *mergeScript.files* or *triageScript.files*. The path to this list file is available as the parameter *mergelist* (for auto-merge) or *triagelist* (for auto-triage). In addition, if either parameter is defined in your RMDB file, it will be used instead.

## Testplan Auto-import

Testplan auto-import is an optional adjunct to the auto-merge functionality in VRM. The testplan is usually represented as a XML file exported from a documentation application, such as Microsoft Excel or Microsoft Word. A Questa utility converts this XML file into UCDB format and the resulting UCDB is merged with the UCDB files generated from the individual tests. While there are any number of ways by which to accomplish the same task via commands in the VRM *preScript* and *postScript* Actions, the common case can be automated via parameters in the RMDB file.

Automation of the testplan import step is enabled by the presence of a non-blank *tplanfile* parameter. It is recommended that the parameter is visible from the Runnable that initiates the first merge operation to any given *mergefile*. This is to ensure timely coverage reporting for that *mergefile*.

When auto-merge is triggered for any mergefile, the tplanfile parameter is checked. If it is the first occurrence of the tplanfile for that target mergefile, a tplanScript is scheduled.

The value of the *tplanfile* parameter should include the absolute path to the XML source file and the command-line options to be passed to the XML Import Utility. The default tplanScript command used to import the XML testplan file is the following:

```
eval [list xml2ucdb -ucdbfilename  
(%tplanucdb%) (%tplanoptions:) (%tplanfile%)]
```

For example, if the testplan is extracted from a Microsoft Excel spreadsheet, the *tplanfile* parameter definition might look something like the following:

```
<parameter name="tplanfile">/testplan.xml</parameter>  
<parameter name="tplanoptions">-format Excel</parameter>
```

Note that the path to the XML file is absolute. The tplanucdb parameter is an internally defined parameter that specifies the target of the testplan import operation.

There is no command line option to disable testplan auto-import

Testplan auto-import is only enabled when auto-merge is enabled and when the *tplanfile* parameter is defined and non-blank.

## Enabling and/or Disabling Coverage Merge Flows

This section summarizes how the various merge flows are triggered and who is responsible for the *vcover merge* command.

**Table 10-2. Coverage Merge Flow Summary**

Has ucdbfile?	Has mergelist?	Has mergefile?	Has-noqueuemerge?	Type of merge	Merge command
No	--	--	--	Unspecified	No UCDB file specified
Yes	No	No	--	Post-process (DIY) merge	Part of user postScript
Yes	Yes	No	--	List-based manual merge	Part of user postScript
Yes	--	Yes	Yes	Incremental auto-merge	Generated by vrun
Yes	--	Yes	No	Queued auto-merge	Generated by vrun

The *ucdbfile* parameter must be specified for any merging to take place within VRM (unless a user *postScript* scans for potential UCDB files). In addition, either the *mergefile* or the *mergelist* parameter is required for *vrun* to make any attempt to assist in the merging process. If *ucdbfile* and *mergefile* are both specified for a given Task Runnable, the default is queued merging, unless the *-noqueuemerge* option is specified; in which case, incremental merging (queued merging without the queue) is performed. If the *mergefile* parameter is not specified but the *mergelist* parameter is specified, then the path to the UCDB file is added to the *mergelist* file for later manual merging.

Even if auto-merge is enabled due to the presence of the *mergefile* attribute, the auto-merge algorithm can be disabled manually with the *-noautomerge* command-line option. This prevents the creation of *mergeScript* Actions to perform the merge. It does not, however, disable the generation of a merge list via the *mergelist* parameter.

The *OkToMerge* user-definable procedure can also set the *mergefile* or *mergelist* elements of the passed data array (that is, *data(mergefile)* or *data(mergelist)*), in which case the values provided by the *OkToMerge* procedure will override any values present in the RMDB database file.

The default commands used in the auto-merge *mergeScript* Action are as follows:

```
set mergefile {(%mergefile%)}
set ucdbfiles {(%ucdbfiles%)}
set cmd [list vcover merge (%mergeoptions%) -out $mergefile]
if {[file readable $mergefile]} {lappend cmd $mergefile}
eval exec $cmd $ucdbfiles
```

This built-in script can be overridden simply by adding a *mergeScript* element to the Runnable associated with the UCDB file or to some Runnable visible from that Runnable by inheritance. If a *mergeScript* element is found in the RMDB file, then that script is used to perform the merge. If no *mergeScript* element is found, the above default script is used instead. The merge options, if any, should be placed in a parameter called *mergeoptions*, which must also be visible from the Runnable associated with the UCDB file to be merged.

In queued mode, the *vrun* process accumulates a list of UCDB files to be merged and ensures that only one merge is ever executing against a single target merge file at any one time. The files are clubbed according to the contents of the generated *mergeScript* (including the *mergeoptions* value), thus drawing a balance between minimizing the total number of merge operations and reducing the latency of the merge results. See “[UCDB File “clubbing” in Automated Merge and Automated Triage](#)” on page 400 for further details. The list of UCDB files to be merged is supplied by *vrun* in the *ucdbfiles* parameter.

This list is constructed from the contents of the *ucdbfile* parameters specified by eligible *execScript* Actions that completed since the last merge against this particular target merge file (in incremental mode, there is always exactly one file listed in the *ucdbfiles* parameter whereas, in queued mode, there can be one or more UCDB files in the *ucdbfiles* list).

If the file specified in the *mergefile* parameter already exists, then the *mergeScript* merges the specified UCDB files into the existing target merge file using the appropriate *mergeScript* commands as described above. If the target merge file does not exist, then the first UCDB file targeted for merging is instead copied directly to the target merge file (the copy operation is performed in-line in order to avoid a known race condition).

Note that in the case of queued auto-merge, several UCDB files can be queued for merging in a single command. In this case, the Runnable from which the *mergeScript* will be drawn is the Runnable associated with the last *execScript* to have contributed to the merge. Since test completion times can vary randomly, it is highly recommended that all *execScript* Actions which merge into a single target merge file override the *mergeScript* commands in exactly the same way. The user should not make any assumptions about the order in which the component UCDBs will become available.

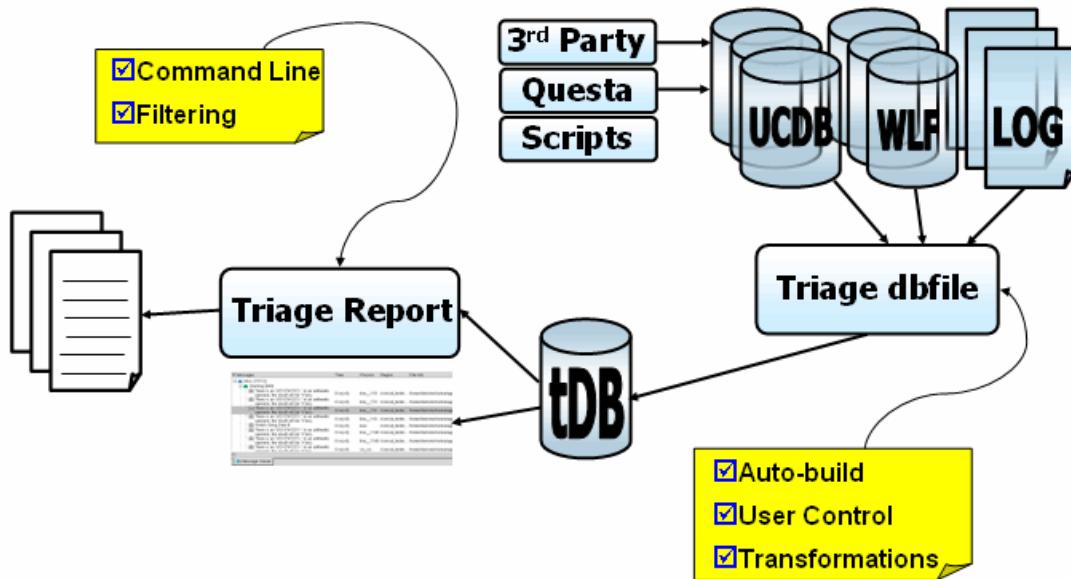
## Automated Results Analysis

An automated flow is available for results analysis (triage). Triage is a means of collating messages from multiple verification tasks that make the regression into a single view. Through grouping and filtering, the user can easily determine common failures and patterns.

Results analysis allows the errors across multiple verification runs to be stored and analyzed to allow triage of the tests across a complete regression run. This gives the user the ability to group and sort failures over a complete set of regression tests so that similar failures can be identified, which saves time and resources chasing the same problems and gets to the source of the error that occurs first. This results analysis feature enables the users to reduce their debug time.

Figure 10-11 illustrates the results analysis flow.

**Figure 10-11. Results Analysis (Triage) Flow**



The triage database (*.tdb*) file is generated using the *triage dbfile [options]* command.

**Table 10-3** outlines the results analysis (triage) modes, how each is triggered, and who is responsible for the *triage dbfile* command.

**Table 10-3. Automated Results Analysis (Triage) Modes**

Has ucdbfile?	Has triagefile?	Has -noqueuetriage?	Type of triage	Triage command
No	--	--	Unspecified	No UCDB file specified
Yes	No	--	Post-process (DIY) triage	Part of user postScript
Yes	Yes	Yes	Incremental auto-triage	Generated by vrun
Yes	Yes	No	Queued auto-triage	Generated by vrun

As in the auto-merge case, both the *ucdbfile* and *triagefile* parameters must be specified for automated failure triage to take place. If both parameters are specified, the default is queued triage (unless the *-noqueuetriage* option is specified, in which case incremental triage is performed).

Even if auto-triage is enabled due to the presence of the *triagefile* attribute, the auto-triage algorithm can be disabled manually with the *-noautotriage* command-line option. This prevents the creation of *triageScript* Actions to load the UCDB into the triage database.

The default commands used in the auto-triage *triageScript* Action are as follows:

```
set triagefile {(%triagefile%)}

eval [list triage dbfile (%triageoptions%) -name $triagefile (%ucdbfiles%)]
```

This built-in script can be overridden simply by adding a *triageScript* element to the Runnable associated with the UCDB file or to some Runnable visible from that Runnable by inheritance. If a *triageScript* element is found in the RMDB file, that script is used to perform triage database insertion. If no *triageScript* element is found, then the above default script is used instead. The triage options, if any, should be placed in a parameter called *triageoptions*, which must also be visible from the Runnable associated with the UCDB file to be added.

In queued mode, the *vrun* process accumulates a list of UCDB files to be triaged and ensures that only one *triage dbfile* command is ever executing against a single target database at any one time. The files are clubbed according to the contents of the generated *triageScript* (including the *triageoptions* value), (see “[UCDB File “clubbing” in Automated Merge and Automated Triage](#)” on page 400), thus drawing a balance between minimizing the total number of *triage dbfile* operations and reducing the latency of the triage results. The list of UCDB files to be added to

the triage database is supplied by *vrun* in the *ucdbfiles* parameter. This list is constructed from the contents of the *ucdbfile* parameters specified by those *execScript* Actions that completed since the last *triage dbfile* insertion into this particular triage database file (in incremental mode, there is always exactly one file listed in the *ucdbfiles* parameter whereas, in queued mode, there can be one or more UCDB files in the *ucdbfiles* list).

Note that the auto-triage does nothing special based on the existence (or lack thereof) of the triage database file. If there might be pre-existing data in the database that is not intended to contribute to the current regression run's triage analysis, a *preScript* early in the regression suite should manually execute a *triage dbfile -clear* command to initialize the triage database.

In the case of queued auto-triage, several UCDB files can be queued for triaging in a single command. In this case, the Runnable from which the *triageScript* will be drawn is the Runnable associated with the last *execScript* to have contributed to the triage. Since test completion times can vary randomly, it is highly recommended that all *execScript* Actions that can be queued for loading into a single triage database file override the *triageScript* commands in exactly the same way. The user should not make any assumptions about the order in which the component UCDBs will become available.

## UCDB File “clubbing” in Automated Merge and Automated Triage

Under non-queued incremental auto-merge and auto-triage, each completed UCDB file results in a separate merge/triage operation to handle that specific file. Because of file locking, multiple concurrent merge and/or triage commands to the same target file can experience additional latency as the locking algorithm blocks all but one of the concurrent commands from accessing the target file at any given time. VRM supports queued versions of incremental auto-merge and auto-triage that avoid this file locking latency.

The basic idea is simple. As long as a merge or triage is running against a particular target file (a “merge file” in the case of auto-merge or a “triage database” in the case of auto-triage), subsequent requests to merge/triage an individual UCDB result in the UCDB file being queued up internal to the *vrun* application. When the current merge/triage process completes, the queued UCDB files are then used to create another merge/triage Action on-the-fly and that Action is launched. Subsequent UCDB files becoming available for merge/triage while the new process is still running are likewise queued up to await the completion of the new merge or triage process. Besides avoiding the processing and latency overhead of file locking collisions, if more than one leaf-level UCDB happens to be waiting in the queue, all queued files are merged and/or inserted into the triage database with a single command, thus reducing the number of merge/triage commands that must be executed.

By default, UCDB input files are clubbed according to the script that must be executed in order to add them to the target merge/triage file. The script commands to be used in the *mergeScript* or *triageScript* Action are either generated internally by the auto-merge/triage process or come

from the RMDB database RMDB file (if the user overrides the default *mergeScript/ triageScript* commands). These commands are parameter-expanded in the following manner:

1. Any (%*mergefile*%) or (%*triagefile*%) references are replaced with the current *mergefile* or *triagefile*, respectively.
2. Any “(%ucdbfiles%)” references are replaced with the single string *dummy.ucdb* (in order to allow Action scripts to match even if the UCDB file in question differs (which it always will).
3. All other parameter references are expanded as they would be when the Action script file is generated and launched.

The commands are then concatenated and run through an SHA-1 message digest generator. The digest string is used to identify unique Action scripts (see note below). Leaf-level UCDB files are only clubbed together if the commands that would be executed to perform the merge and/or triage operation are identical in all respects **except** for the leaf-level UCDB names (which by definition, are different). If the Action scripts defined for two or more UCDBs differ, or if parameter references (often used for command-line options) differ between the multiple Action scripts, the UCDBs in question are not clubbed together. Instead, VRM launches a merge/triage Action using the first of the queued UCDB files and any other queued UCDB files whose SHA-1 message digest value matches that of the first of the queued UCDB files. Any remaining files are re-queued in the order they were initially queued. When that merge/triage Action finishes, another is started using the first of the postponed UCDB files and any other UCDB files whose message digest matches that file. This ensures an equitable throughput of UCDBs without clubbing together files that cannot be merged and/or triaged with the same commands.

If the default auto-merge and auto-triage commands are **not** overridden in the RMDB file, and if there is only one defined value for both the *mergeoptions* and *triageoptions* parameters throughout the entire regression run, a minor performance increase can be realized by disabling the SHA-1 based clubbing algorithm. To do this, set the *MTI\_VRUN\_CLUB\_BY\_TARGET* environment variable. The value of the variable is irrelevant. If a variable by that name is set in the environment, only the target file name is used to determine how to club the UCDB files.

Note that while the use of a message digest algorithm such as SHA-1 does not guarantee that two script files will never hash to the same value, it is generally believed to be computationally infeasible to come up with more than one unique text string that all hash to the same message digest.

By default, there is no limit to how many UCDBs may be clubbed in a single merge/triage operation. The list of UCDB files is written to a text file in order to prevent the merge/triage command line from exceeding operating system limits.

The list of UCDB files is created in the same directory as the corresponding script file (*mergeScript.do* or *triageScript.do*). The default name for this file is either *mergeScript.files* or *triageScript.files*. The path to this list file is available as the parameter **mergelist** (for auto-

merge) or **triagelist** (for auto-triage). In addition, if either parameter is defined by the user in the RMDB file, those user-specified paths will be used instead.

## Automated Trend Analysis

This functionality facilitates the tracking of verification plan coverage over time, specifically it allows you to monitor the changes in coverage information for merge files collected over different regression runs.

This auto-trend feature is enabled when you enable auto-merge and there is a non-empty *trendfile* parameter. Refer to the section “[Enabling and/or Disabling Coverage Merge Flows](#)” for more information.

You can disable the auto-trend feature with the -noautotrend argument to the vrun command, or by forcing the **OkToTrend** user-definable procedure to always return a false value (0).

The trendScript element adds a record to the status event log in the form:

```
<timestamp> trend <trend-action> <trendfile> <mergefile>
```

for example:

```
trend nightly/test1/trendScript /tests/vrm/merge/tplanxml/VRMDATA/  
trend.ucdb /tests/vrm/merge/tplanxml/VRMDATA/merge.ucdb
```

The use of the -rerun argument to the vrun command enables [Failure Rerun Functionality](#). Where the most common use model is collecting extra debugging information in a second run. Both passes comprise a single regression run, thus launching trending commands at the end of the second pass only. The files marked for trending from the first pass together with those of the second pass would launch corresponding trendScript actions at the end of the second pass.

## Multi-level Auto-merge Example

The example makes use of the queued auto-merge functionality in *vrun*. The merge is divided into two levels, the second-level merge being responsible for merging two merge files produced by the first-level merge. This example also illustrates how unit-level and system-level coverage can be merged into a single UCDB using VRM.

See “[Complete Multi-level Auto-merge Example](#)” on page 408 for the example used in the following sections.

<b>Auto-merge Example Design .....</b>	<b>403</b>
<b>Auto-merge Example RMDB Database.....</b>	<b>403</b>
<b>First-level Merge in Detail .....</b>	<b>405</b>
<b>Avoiding Problems with Inheritance .....</b>	<b>406</b>
<b>Second-level Merge in Detail .....</b>	<b>408</b>
<b>Complete Multi-level Auto-merge Example .....</b>	<b>408</b>

## Auto-merge Example Design

The block under test is a Hamming parity generator module (*mlm\_par*), purposely written as a series of SystemVerilog *if* statements in order to achieve partial statement coverage (if it were written with continuous assign statements, every statement would be covered by every test). This block is tested under two testbenches. One (*mlm\_uni*) simply instantiates the block once, drives some sample values, and prints both the input value and the resulting Hamming parity code. The other testbench (*mlm\_sys*) instantiates two parity blocks as part of an encoder/decoder module pair that test the ability of the Hamming code to repair single-bit failures. The second testbench also has a self-checking mechanism built-in.

## Auto-merge Example RMDB Database

The Verification Run Manager DataBase (RMDB) file is the VRM database.

The example regression consists of two groups of tests, one containing three tests and one containing two tests, although the basic concept can easily be expanded to handle more groups, larger groups, or even more levels. The *nightly* Runnable is the top-level Runnable that is invoked in order to execute the entire regression run. The *unittest* Runnable contains two tests that exercise the Hamming parity block in isolation. The *systemtest* Runnable contains three tests that exercise the parity generator as part of the encoder/decoder pair. Both testbenches are compiled once (each) in the *preScript* of the *unittest* and *systemtest* Groups and the stimulus used for each test is selected with a *plusarg* option on the *vsim* command line.

There is one *execScript* shared by all tests and one *preScript/postScript* pair shared by the two main Groups (*unittest* and *systemtest*). All the scripts are contained in a base Runnable called

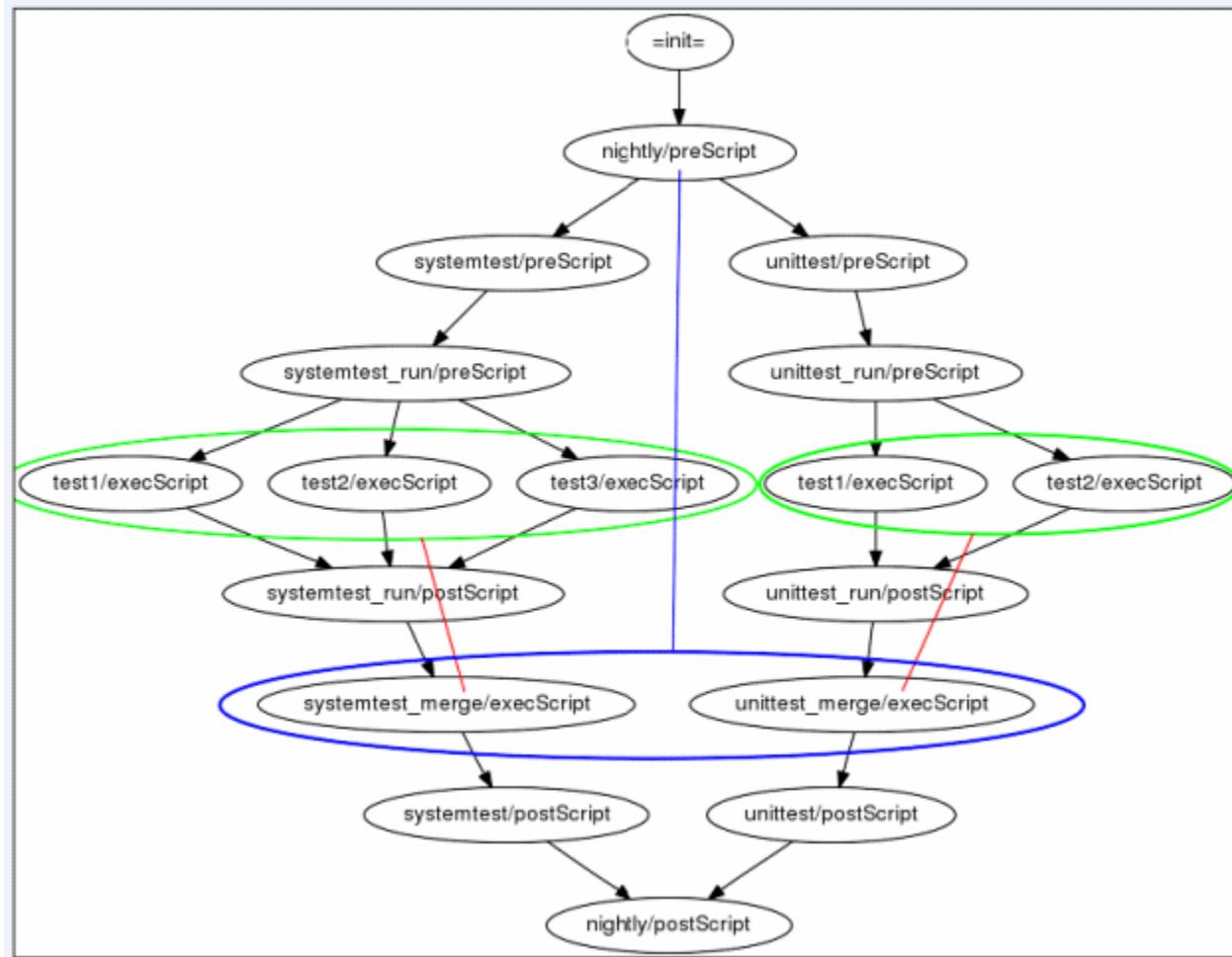
*firstlevel* (because the Groups that use this Runnable as a base Runnable are the two Groups that each do a first-level merge of UCDB files from the individual tests).

Because the Hamming parity blocks are at different hierarchical levels in the two testbenches, you can only merge the coverage from the two Groups of tests by using the *-strip <n>* and *-install <path>* options of *vcover merge*. The *-strip <n>* option removes *<n>* levels of hierarchy from instance and object names in the data files. The *-install <path>* option adds *<path>* as additional hierarchy on the front end of instance and object names in the data files. Refer to the *Questa SIM Reference Manual* for complete details on the *vcover merge* command.

Since the *mergefile* parameter inherits down the entire tree, you have to be careful how you configure the RMDB file for auto-merge. The basic approach is to configure three separate auto-merge targets. One auto-merge target will receive UCDBs from the *unittest* Group, one will receive UCDBs from the *systemtest* Group, and a third will receive the per-Group merge files as they are completed and merge them into a single top-level merge file.

Figure 10-12 illustrates the topology of the example RMDB database file.

**Figure 10-12. Topology of the Example RMDB Database File**



The two green circles represent the first-level merges. Since the auto-merge feature in *vrun* can handle multiple concurrent merge queues, two first-level merges can be running at any one time (one to the *unittest* merge target and one to the *systemtest* merge target). Each first-level merge file is written to the top-most Runnable comprising the Group which that intermediate merge file will represent.

Once all the merges for a given Group are complete, a sibling Task to the first-level merge Group copies the Group merge file to its own directory, applying the *-strip* and *-install* options where applicable. This step is represented by the two red arrows (lines) in the diagram.

These sibling Tasks form the leaves of the third auto-merge Group whose target is a merge file located in the *nightly* Runnable itself. As each Group merge file is completed, it is merged into the top-level merge file by this third auto-merge queue. This step is represented by the blue circle. The resulting top-level merge file is written into the working directory for the *nightly* Runnable.

The standard merges (that is, one or more UCDB files being merged as is into the target merge file) are all handled automatically by *vrun*. The *vrun* process makes sure that only one merge is running against a given target merge file at any given time. While a merge is running against a given target merge file, *vrun* will “batch-up” any completed UCDBs destined for that file and merge them into the target file in a single *vcover merge* invocation once the ongoing merge operation has completed. The only thing the user has to add to the configuration is the *vcover merge* commands for the non-standard merge operations (in this case, the strip/install step) and the appropriate *ucdbfile* and *mergefile* attributes for the three auto-merge Groups.

## First-level Merge in Detail

The two first-level merges are more or less identical, so the merge under *systemtest* is used as an example for purposes of this discussion. The *systemtest* Runnable defines a parameter called *group* whose value is *systemtest*. This is only needed because *systemtest* and *unittest* share many configuration elements and the *group* parameter allows these shared elements to be parameterized to match the Group in question.

The *systemtest* Runnable invokes, as members, the *systemtest\_run* and *systemtest\_merge* Runnables. The former is responsible for running the concurrent simulations defined in the system-level subtree while the latter is responsible for managing the strip/install process and serving as a leaf node for the second-level merge. Since the merge can only be done **after** all the regression tests have run, this Group is marked as sequential (meaning *systemtest\_merge* will only run after *systemtest\_run* completes).

The *systemtest\_run* Runnable defines parameters for the top-level module (*top*) and the testbench source file (*tbsrc*). It also, via the *firstlevel* base Runnable, defines the common merge file (*mergefile*) for the Group and a template (*ucdbfile*) for identifying the UCDB file associated with each test. As described in “[Coverage Merge Options](#)” on page 376, it is the presence of a non-empty *mergefile* parameter that triggers the auto-merge functionality.

Because the coverage from all the tests in the regression suite are merged into a single file as part of the regression process, the names assigned to the *TESTNAME* attribute in the test data record must be unique across the entire suite. The *ucdbfile* parameter uses the template `(%group%)_(%testname%)` to generate a testname that is based on the *group* parameter (*systemtest* or *unittest*) and the name of the test itself. This is necessary because two of the three Runnables that represent tests are reused by the two groups (an example of employing configuration reuse in order to save typing).

A *workpath* parameter is also defined. The value of this parameter is designed to point to the working directory for the top-most Runnable of the Group in question. This is needed because parameter references are expanded with respect to the Runnable on whose behalf the expansion is taking place (not the Runnable in which the parameter is defined). Because of inheritance, it is not possible to refer to the location of an ancestor Runnable's working directory using only parameter references.

As each test runs, it has a *ucdbfile* parameter that points to the UCDB file for the test and a *mergefile* parameter that points to the target merge file for the main Group in which that test is Running. For example, *test1* of the *systemtest* Group (which is actually a member of *systemtest\_run*) defines the following parameters (among others not mentioned):

- *ucdbfile* parameter:  
*systemtest\_test1.ucdb* (relative to the working directory for *test1*)
- *mergefile* parameter:  
*/path/to/VRMDATA/nightly/systemtest/systemtest\_run/systemtest.ucdb* (absolute)

When the test completes, the auto-merge algorithm schedules *systemtest\_test1.ucdb* to be merged into the target merge file for the *systemtest* sub-suite. The *postScript* for the *systemtest\_run* Group becomes dependent on the completion of the scheduled merge process so that, by the time *systemtest\_run* has completed, the target merge file for the *systemtest* Group is also complete.

Note that the value of the *mergefile* parameter can be pretty much anything that is constant for all tests in the subtree to which the merge file in question applies and that can be easily found by the second-level merge. In this case, an argument can be made for putting the target merge file in *nightly/systemtest* but, in the end, that is a matter of user preference (so long as the file can be located by both the first-level and second-level merge processes).

## Avoiding Problems with Inheritance

Inheritance solves a lot of problems in the general case. It allows a common parameter value to be entered into the RMDB database once and still be visible from large parts of the regression topology. It allows complex parameter values to themselves be parameterized using a simple parameter (such as a testname or a seed) defined in each test. Auto-merging is the one place where this flexible inheritance architecture works against us.

The auto-merge algorithm requires that *vrun* be able to locate both the UCDB file for any given test and the target merge file into which this UCDB is to be merged. This is done with the *ucdbfile* and *mergefile* parameters, respectively. For each test, the *ucdbfile* parameter points to the individual UCDB file produced by that test and the *mergefile* parameter points to the common target merge file.

The issue comes at the interface of a lower-level merge and a higher-level merge. Assume, for example, there are two sets of three tests each. [Table 10-4](#) shows the merges that need to be made to support multi-level merging for this example (see[Complete Multi-level Auto-merge Example](#)) (paths are ignored in this example):

**Table 10-4. Merges for Avoiding Inheritance Problems in Example**

Level	Merge File	Input UCDB File
1	group1.ucdb	group1_test1.ucdb group1_test2.ucdb group1_test3.ucdb
1	group2.ucdb	group2_test1.ucdb group2_test2.ucdb group2_test3.ucdb
2	final.ucdb	group1.ucdb group2.ucdb

Since the target merge file is common to all the tests under that Group, it is only natural that the path to that merge file is defined in the Group itself and inherited by all the tests under that Group. So the *group1* Runnable would define a *mergefile* value for the Group. However, the *group1* Runnable is also a leaf for the second-level merge. So it would like to define the *mergefile* parameter to point to the final target and the *ucdbfile* parameter to point to the results of the first-level merge for which it was responsible. Clearly these are conflicting requirements.

The solution is to split the *group1* Runnable into two parts. In the example above, a Group and a Task that are siblings in a higher-level Group (which serves only as a container to relate the two parts of the problem) are used. The user can also create one Group inside another in a nested division. The goal is simply to separate the top-most (Group) Runnable of the lower-level merge from the leaf Runnable of the higher-level merge.

In the example, the *systemtest\_run* Runnable represents the top-most Group of the first-level merge associated with the *systemtest* subtree. It is that Runnable which defines the *ucdbfile* and *mergefile* parameters for that subtree. Likewise, the *unittest\_run* Runnable defines the *ucdbfile* and *mergefile* parameters for the other first-level subtree. The *systemtest\_merge* Task (which is not under *systemtest\_run*) inherits its *ucdbfile* and *mergefile* parameters from the *nightly* Group that serves as the top-most Group for the second-level merge. Since *systemtest\_merge* (and *unittest\_merge* for the other first-level subtree) are Tasks, they are subject to the same post-execution analysis as any other UCDB-producing Task, which includes auto-merging. Therefore, the UCDB file “produced” by the *systemtest\_merge* Task is auto-merged into the

top-level merge file defined in the *nightly* Group as soon as the *execScript* for *systemtest\_merge* completes.

The only piece left is to cause the first-level merge files to appear to be UCDB files generated by the *execScript* Actions in the *unittest\_merge* and *systemtest\_merge* Runnables. In the case of the *unittest* subtree, the first-level merge file is to be used as is (without any strip/install options). Therefore, it is simply copied over by the *execScript* of the *unittest\_merge* Runnable. In the case of the *systemtest* subtree, the *execScript* of the *systemtest\_merge* Runnable performs the strip and install transformation, resulting in a new UCDB file with the same hierarchy as the unit tests. These two are then merged together by the auto-merge algorithm to produce the top-level merge file containing results from all five tests.

This is not the only architecture that will result in a properly auto-merged UCDB file. But by understanding how this example is constructed, the principle of multi-level auto-merge using *vrun* can be more clearly understood.

## Second-level Merge in Detail

The second-level merge is very similar to the two first-level merges. It is worth noting that the only two *execScript* Actions under the *nightly* subtree that are not also under one of the two main Groups (*unittest* and *systemtest*) are the *execScript* Actions under *unittest\_merge* and *systemtest\_merge*. This is intentional, since the two first-level merge files generated by those Actions are to be the only input files to the second-level merge. If other Task Runnables were defined at this level, their UCDBs would also participate in the second-level merge (which might also be intentional, under the right circumstances).

The *nightly* *Group* defines a *ucdbfile* parameter which, when expanded from the context of either of the two *execScript* Actions mentioned in the paragraph above, expands to point to the first-level merge file for the Group with which that *execScript* is associated (as a sibling, of course). The *nightly* Group also defines a *mergefile* parameter that points to the ultimate top-level merge target. Therefore, as each of the main test Groups completes, its first-level merge file is merged into the second-level merge file.

## Complete Multi-level Auto-merge Example

This section shows a complete multi-level auto-merge example.

**Figure 10-13. Multi-level Auto-merge**

```

<rmdb version="1.1">
  <!--
    Second-level merge: top-level Runnable and strip/install
  -->
  <runnable name="nightly" type="group">
    <parameters>
      <parameter name="blksrc">(%RMDBDIR%)/src/mlm_par.sv</parameter>
      <parameter name="ucdbfile">(%group%).ucdb</parameter>
      <parameter name="mergefile">(%DATADIR%)/nightly/merge.ucdb
        </parameter>
    </parameters>
    <members>
      <member>unittest</member>
      <member>systemtest</member>
    </members>
    <postScript>
      <command>coverage open merge.ucdb</command>
      <command>coverage attribute -test * -name TESTNAME -name
        TESTSTATUS</command>
    </postScript>
  </runnable>
  <runnable name="unittest" type="group" sequential="yes">
    <parameters>
      <parameter name="group">unittest</parameter>
    </parameters>
    <members>
      <member>unittest_run</member>
      <member>unittest_merge</member>
    </members>
  </runnable>
  <runnable name="unittest_merge" type="task">
    <execScript>
      <command>file copy ../../(%group%)_run/(%ucdbfile%
        (%ucdbfile%))</command>
    </execScript>
  </runnable>
  <runnable name="systemtest" type="group" sequential="yes">
    <parameters>
      <parameter name="group">systemtest</parameter>
    </parameters>
    <members>
      <member>systemtest_run</member>
      <member>systemtest_merge</member>
    </members>
  </runnable>
  <runnable name="systemtest_merge" type="task">
    <execScript>
      <command>vcover merge -strip 3 -install /mlm_uni (%ucdbfile%
        ../../(%group%)_run/(%ucdbfile%))</command>
    </execScript>
  </runnable>
  <!--
    First-level merge: unittest and systemtest hierarchies
  -->
  <runnable name="firstlevel" type="base">

```

## Post-execution Analysis

### Complete Multi-level Auto-merge Example

---

```
<parameters>
    <!-- this must be hard-coded because parameters are expanded
        from the POV of the Action -->
    <parameter name="workpath">(%DATADIR%)/nightly/(%group%
        /(%group%)_run</parameter>
    <parameter name="testname">(%group%)_(%pattern%)</parameter>
    <parameter name="ucdbfile">(%testname%).ucdb</parameter>
    <parameter name="mergefile">(%workpath%)/(%group%).ucdb
        </parameter>
</parameters>
<preScript launch="vsim">
    <command>file delete -force work</command>
    <command>file delete -force (%mergefile%)</command>
    <command>vlib work</command>
    <!-- forego coverage on testbenches in order to avoid merge
        warnings -->
    <command>vlog (%tbsrc%)</command>
    <command>vlog -coverAll (%blksrc%)</command>
    <command>vopt (%top%) -o (%top%)_opt</command>
</preScript>
<execScript launch="vsim">
    <command>vsim -c -coverage -lib (%workpath%)/work +(%pattern%
        (%top%))</command>
    <command>run -all</command>
    <command>coverage attribute -name TESTNAME -value
        (%testname%)</command>
    <command>coverage save (%ucdbfile%)</command>
</execScript>
<postScript launch="vsim" mintimeout="60">
    <command>coverage open (%mergefile%)</command>
    <command>coverage attribute -test * -name TESTNAME -name
        TESTSTATUS</command>
</postScript>
</runnable>
<runnable name="unittest_run" type="group" base="firstlevel">
    <parameters>
        <parameter name="top">mlm_uni</parameter>
        <parameter name="tbsrc">(%RMDBDIR%)/src/mlm_uni.sv</parameter>
    </parameters>
    <members>
        <member>test1</member>
        <member>test2</member>
    </members>
</runnable>
<runnable name="systemtest_run" type="group" base="firstlevel">
    <parameters>
        <parameter name="top">mlm_sys</parameter>
        <parameter name="tbsrc">(%RMDBDIR%)/src/mlm_sys.sv</parameter>
    </parameters>
    <members>
        <member>test1</member>
        <member>test2</member>
        <member>test3</member>
    </members>
</runnable>
<runnable name="test1" type="task">
    <parameters>
        <parameter name="pattern">short</parameter>

```

```
</parameters>
</runnable>
<runnable name="test2" type="task">
  <parameters>
    <parameter name="pattern">broad</parameter>
  </parameters>
</runnable>
<runnable name="test3" type="task">
  <parameters>
    <parameter name="pattern">shift</parameter>
  </parameters>
</runnable>
</rmdb>
```

# User-definable Procedures and Loading of Arbitrary TCL Code

---

VRM is designed to be a versatile scripting solution for a wide variety of regression flows and users. The basic architecture of nested Groups of Tasks following the *preScript*, *execScript*, *postScript* execution model is, itself, already quite extensible. Support for parametrized commands within those scripts further enhances that extensibility. However, the details involved in each step of the execution process are not quite as clear-cut. There are two approaches that must be balanced against one another as follows:

- VRM can provide only the basic execution model described above, leaving all the details such as pass/fail analysis, coverage merging, and directory cleanup to be implemented in the user-defined scripts.
- There is little value-added in a scripting mechanism where **all** the implementation details must be supplied by the user. And yet, the more of these details that are provided by the scripting mechanism for the sake of convenience, the less flexible that mechanism becomes for users whose environment cannot conform to the exact assumptions made when the scripting mechanism was first designed.

It is impossible to customize VRM for each environment it encounters. A solution whereby certain details involved in managing a regression suite are given a simple default behavior, one which should cover the bulk of common cases, and a mechanism is provided by which one or more of those algorithmic “chunks” can be overridden by the users to more closely fit their environment. The TCL language provides the necessary hooks to make such an implementation not only possible, but simple.

A “user-definable procedure” is a chunk of VRM functionality that has been isolated as a separate TCL procedure with a well-documented interface. One or more of these procedures can be overridden by user-defined procedures, written in TCL code either in a file passed to VRM via a command-line option or as part of the RMDB database. Arguments containing information relevant to each procedure are passed to the procedure when invoked and, in many cases, the procedure is expected to return the results of its analysis. These procedures are loaded into a namespace reserved for user-definable procedures so that there is little risk of inadvertently altering the internal algorithm of VRM. Exceptions returned from these user-definable procedures are trapped and reported in the due course of running the Tasks defined in the regression suite and should not affect the operation of VRM (other than to trigger an appropriate error message in the VRM log output).

In addition, VRM defines several utility procedures in the same namespace as the user-definable procedures. Their purpose is to save the user from having to reinvent certain complex operations that can prove useful when called from within a user-defined procedure. Note that these utility procedures can be overridden if necessary, but they are not called directly from VRM by default. Therefore, depending on which of the user-definable procedures are overridden, an override on a utility procedure can, in fact, have no effect on the behavior of VRM.

The following subsections list the user-definable procedures and utility procedures available in VRM, along with the input arguments and expected return value, if any, for each procedure. For the most part, the input arguments should be considered as read-only from the point of view of the user-defined procedure. There are some cases, however, where a user-defined routine may need to write some information to one or more of the arguments in the array. In those cases, the expected behavior is listed under the heading “Side effects” for the procedure in question. The default (built-in) version of most of these routines will emit an informational message when VRM is executed in *-debug* mode.

<b>Modifying VRM Behavior with User-definable Procedures .....</b>	<b>413</b>
<b>Loading User-supplied TCL Code Using usertcl into VRM .....</b>	<b>415</b>

## Modifying VRM Behavior with User-definable Procedures

There are certain aspects of the behavior of VRM that depend on assumptions about the verification environment. For some of these behavioral details, VRM defines an appropriate default that should work for the majority of environments and a mechanism whereby that behavior can be modified by the user at runtime. For each modifiable aspect, the default behavior is implemented in an isolated TCL procedure within VRM. To override this default behavior, the user creates a TCL code fragment that defines a procedure of the same name as the procedure to be overridden. This code fragment is written to conform to the same interface as the original function and is loaded into VRM at runtime. When VRM invokes the procedure, the user's version of that procedure is used instead of the built-in procedure. The procedure as defined by VRM is called a “user-definable” procedure. The procedure supplied by the user to override this procedure is called a “user-defined” procedure.

Input arguments are passed to each user-definable procedure via a named TCL associative array. The procedure receives the name of the array as its only formal argument and uses a TCL *upvar* command to create a local reference to the array. An associative array is used so that future additions to the list of data values supplied to any given procedure will not affect the ability of existing user-defined procedures (written before the new data values existed) to function as usual.

Some (but not all) user-definable procedures are expected to return the results of their analysis. In this case, the normal TCL *return* statement can be used. Some also have other side-effects (like altering the values in the associative array passed in by the caller). These side-effects are noted in detail below for each user-definable procedure.

An example of a user-definable procedure is the procedure called to determine whether the cumulative number of failures is sufficient to cancel the remaining jobs in the regression run. This decision is made in a user-definable procedure called [StopRunning Concept](#) (see [page 370](#)). The associative array passed as the input argument to this procedure contains the

following information (in addition to certain global values passed to all user-defined procedures):

```
$data(pass) := number of jobs which represent passed simulations
$data(fail) := number of jobs which represent failed simulations *plus*
                number of failed executions
$data(sims) := sum of array(pass) and array(fail), for convenience
```

The *StopRunning* procedure is expected to return a true value (1) if the number of failures is sufficient to cancel the remaining jobs or a false value (0) if the regression suite should continue running. The default internal procedure never cancels the regression. However, the user may wish to cancel the regression run if the cumulative number of failures ever exceeds the cumulative number of successes by five. In order to do this, create the following TCL code fragment and load it into VRM:

**Figure 10-14. Cancel Regression Run Code Fragment**

```
proc StopRunning {userdata} {
    upvar $userdata data
    if {$data(fail) > [expr {$data(pass) + 5}]} {
        return 1
    }
    return 0
}
```

In another example, define a *usertcl* element containing a simplistic implementation of a “delete-on-pass” policy (that is, the working directory for passing simulations is deleted upon completion of the simulation). Note that the text content of the *usertcl* element follows standard TCL syntax (any XML-specific characters such as < and & must be escaped). The format of the XML code to implement this override might look like the following:

**Figure 10-15. Delete-on-pass Implementation**

```
<usertcl name="delete-on-pass">
    proc OkToDelete {userdata} {
        upvar $userdata data
        lappend data(nodelete) "merge.ucdb" "postScript.log"
        return [expr {$data(passfail) == "pass"}]
    }
</usertcl>
```

The user-definable procedure *OkToDelete* analyzes the status of the Action in question and determines if the auto-deletion mechanism should be invoked. The statement *upvar \$userdata data* provides access to the incoming argument array and must be included. After that, the data values within the array can be accessed via the *data* array variable, as in *\$data(passfail)*. This user-defined override implements the “delete-on-pass” policy, returning a true (1) value only if the Action for which it was called passed. The resulting behavior is to delete all working directories and the files therein **except** for the files *merge.ucdb* (the coverage results) and *postScript.log* (the final report from the suite).

There are also utility procedures built into VRM and defined in the same namespace as that used for the user-definable procedures. These procedures can be called by the user to perform certain complex operations that are better defined as a unit. One such procedure is *GetUcdbStatus* (see “[GetUcdbStatus](#)” on page 561) that takes a path to a UCDB coverage file, opens that file, and returns the value of the *TESTSTATUS* attribute of the first Test Data record found therein. This utility procedure comes in handy for any user attempting to override the *AnalyzePassFail* (see [page 364](#)) user-definable procedure in a UCDB-based environment.

## Loading User-supplied TCL Code Using usertcl into VRM

In some cases, it is necessary to load user-defined TCL code into VRM (technically, into the TCL interpreter that is executing the VRM application). This TCL code can be specified in the RMDB database or in a separate file. Once specified, the code can be loaded via a command-line option or automatically at the start of the VRM run. Any TCL code so loaded becomes a global part of the VRM application. Procedures defined within the loaded code can be called from TCL-type parameter value expressions or from user-definable procedures.

If the name of a procedure defined in the loaded TCL code matches that of a user-definable procedure (see “[Catalog of User-Definable and Utility Procedures](#)” on page 499), the loaded procedure definition replaces (overrides) the default definition supplied as part of VRM.

Fragments of TCL code can be defined in the RMDB database using the *usertcl* element. The *usertcl* element must have a *name* attribute by which the element can be identified. An optional *base* attribute can be specified whose value contains the names of one or more base *usertcl* elements. The *usertcl* elements must be defined as top-level elements (that is, as child elements of the document (*rmdb*) element). The contents of a *usertcl* element consist of a single text string that is assumed to be valid TCL code. For example,

```
<usertcl name="debugmode">
  proc checkDebugMode {} {
    if {$::env(DEBUG) eq "yes"} {
      echo "Debug mode enabled"
      return 1
    }
    return 0
  }
</usertcl>
```

The procedure in this example can then be called from a TCL-type parameter value expression as follows:

```
<parameter name="dbgopts" type="tcl">[checkDebugMode] ? "+debug" :
  ""</parameter>
```

which is, in turn, referenced from an Action script command:

```
<command>vsim -c (%dbgopts%)
top</command>
```

A `usertcl` element can also refer to an external file rather than provide TCL content in-line with the XML element itself. In the event a `usertcl` element to be loaded specifies a non-empty `file` attribute, the value of that attribute is treated as a path to a file containing TCL code for that `usertcl` element and the contents of that file (if readable) is loaded in place of the contents of the `usertcl` element. Note that the contents of the `usertcl` element, if any, are ignored in this case, even if the referenced file does not exist. The file path is assumed to be either absolute or relative to the directory from which the RMDB database is read (that is, `RMDBDIR`). The `file` attribute is parameter-expanded so that the path-related predefined parameters can also be used to specify the value of the `file` attribute. For example,

```
<usertcl name="debugmode" file="debugmode.tcl">
    This string is ignored and the contents of the "debugmode.tcl" file
        is loaded instead.
</usertcl>
```

The `usertcl` element can specify one or more base `usertcl` elements via the optional `base` attribute. If a `usertcl` element is loaded for any reason, any base `usertcl` elements to which that element refers is also loaded. The sequence in which the base elements are loaded is the same as the sequence in which the elements base chain would be searched for inherited data items. In essence, the contents of the base `usertcl` elements are inherited by the referring element and treated as though they were defined directly in the referring element (by loading them whenever the referring element itself is loaded).

In many cases, the TCL code loaded provides callable utility functions and user-defined override procedures that are global in nature. That is, they are intended to customize VRM to a specific user's flow. In this case, it makes sense to load the associated TCL code every time VRM is invoked. If a given `usertcl` element is to be loaded automatically at the time VRM starts up, the name(s) of the `usertcl` elements to be loaded should be placed in the `loadtcl` attribute of the document (`rmdb`) element as follows:

```
<rmbd loadtcl="debugmode">
    <usertcl name="debugmode">
        ...auto-loaded TCL code...
    </usertcl>
</rmbd>
```

The `loadtcl` attribute can contain multiple tokens, separated by spaces and/or commas. Each token is assumed to be the name of a `usertcl` element defined at the top-level of the database. When VRM initializes its internal tables (but before the execution graph is expanded), the `usertcl` elements listed in the `loadtcl` attribute of the `rmbd` element are loaded in the order specified in the `loadtcl` attribute. Any procedures defined in the auto-loaded `usertcl` elements can be used in the `if`, `unless`, `repeat`, or `foreach` attributes of `runnable` elements used in the regression suite, since the execution graph is only expanded **after** the auto-loaded TCL fragments have been loaded.

User-supplied TCL code can also be loaded from the command line. The *-loadtcl* command-line option specifies a single TCL code fragment to be loaded. By default, the value supplied to the *-loadtcl* option is assumed to be the path to a file containing valid TCL code. If the value supplied to the *-loadtcl* option is a single token preceded by an at sign (@), as in @*debugmode*, the token is assumed to be the name of a *usertcl* element defined in the RMDB database and, if such a *usertcl* element exists, it is loaded. There can be any number of *-loadtcl* options specified in a single *vrun* command. In this case, the *-loadtcl* command-line options are processed in the order in which they appear on the command line. The *-loadtcl* options from the command line are processed immediately **after** loading any auto-loaded *usertcl* elements have been loaded. In this way, it is possible to provide an alternate definition for a given procedure and load it manually, overriding any definition supplied within the auto-loaded *usertcl* element.

Since the auto-loaded *usertcl* elements are defined in the RMDB database, the TCL code loaded is basically “global” to that particular database. Another RMDB database can have a completely different set of auto-loaded TCL code fragments.

Note that the term “loading” a TCL code fragment means that the contents of that code fragment are passed to the TCL interpreter for evaluation. VRM does not parse the TCL code. It cannot warn of potential procedure overrides. It also cannot predict whether the loaded code will complete successfully. Errors encountered during the “loading” of a TCL code fragment are supported as fatal errors.

Most behavioral tweaks made to VRM via this mechanism are global in nature. Common examples might include changing the pass/fail check to compare an Action script's results against a golden file (by default, the test status is read from the UCDB file, if any) or adding code to notify a 3rd-party tool upon completion of each Action. For these tweaks, or for user-callable procedure definitions that do not change the behavior of VRM, auto-loading is probably the best solution.

Some behavioral tweaks can depend on the context in which VRM is invoked. For example, a tweak intended to load UCDB files into the GUI Test Browser can only be useful when VRM is run from the Questa GUI. The preferred solution, in this case, is to construct a single “tweak” procedure that queries mode information supplied in a parameter or elsewhere. There can be cases, however, where it is simply easier to provide multiple versions of the tweak code and select one such version from the command line via the *-loadtcl* command-line option. Base inheritance can also prove useful in the case where multiple *usertcl* elements are selectively loaded but some user-defined procedures are common across two or more “modes” of behavior. In this case, the code common to multiple behavioral modes can be placed into a single *usertcl* element and referred to from the *usertcl* elements loaded for each unique behavioral mode.

For example, consider the following configuration:

```
<rmdb>
  <usertcl name="common">
    ...code needed for both modes...
  </usertcl>
  <usertcl name="batch" base="common">
    ...code needed for batch mode...
  </usertcl>
  <usertcl name="gui" base="common">
    ...code needed for GUI mode...
  </usertcl>
  ...
</rmdb>
```

The user invokes *vrun* with the option *-loadtcl @batch* when running in batch mode and with the option *-loadtcl @gui* when running in GUI mode (see “[Graphical User Interface \(GUI\)](#)” on page 139). The GUI mode “tweaks” can do things like load the UCDB files from completed simulations into the UCDB Browser as they become available. The batch mode “tweaks” can cause e-mails to be sent out upon completion of certain subsections of the regression suite. Still, this is just a contrived example. As has been mentioned before, it can well be easier and more clear to create a single override procedure that would test a parameter value or a TCL variable to determine which behavior is appropriate.

Note that the order in which base elements are processed is the same as the search order for inherited data objects (such as parameters). Each base element from a list of elements specified in a single *base* attribute is completely processed, including any base elements defined in that base element, before continuing to the next base element in the list. This is the logical outcome of the very definition of base inheritance that treats the contents of a base element as if it has been defined in the referring element.

Also note that in the case where VRM is searching for a data object (such as a parameter) matching certain properties (such as the parameter name), the first match usually satisfies the search and any base elements not consulted in the process will not be consulted at all. In the case of loading *usertcl* elements, there is no “matching” concept. The contents of the complete set of base *usertcl* elements are loaded when the referring element is loaded (in the same order as a network of base elements would have been searched had this been a search like operation).

A *usertcl* element is always loaded by name. The *name* attribute is mandatory on a *usertcl* element. If no *usertcl* element corresponding to a given name is found, an error is emitted to the VRM log output and the *vrun* execution is aborted. The following rules apply to the loading process itself:

1. The contents of all *usertcl* elements are loaded into a common namespace created to hold the user-definable procedure definitions (including the default definitions provided by VRM) in order to prevent the user-supplied TCL code from inadvertently colliding with existing *vrun*=/=vish code.

2. The contents of each *usertcl* element is loaded as an independent chunk, which means that TCL procedure definitions cannot span multiple *usertcl* elements.
3. Errors encountered during loading are reported in the VRM log output and the *vrun* execution is aborted.

In the event the TCL parser throws an error during the loading of a *usertcl* element, there is no way for VRM to control (or even predict) the possible side effects of the partially loaded TCL code fragment. Therefore, on detection of such a loading error, VRM emits an error message to its log output and aborts. The TCL error must be fixed or the offending *usertcl* element not be loaded in order to continue processing.

Note that the purpose of this functionality is to modify aspects of VRM behavior in order to fit the user's verification flow. For this reason, the user should consider any overrides made via user-supplied TCL code to be global to the specific *vrun* invocation (and, by inference, to a specific database). If, for any reason, this behavior needs to be conditional within the same *vrun* invocation, some form of conditional code should be included in the TCL procedures. Various data items related to the Action at hand, on which these conditional checks can be based, are supplied to each user-definable procedure.

**Important Note:** The contents of *usertcl* elements and/or TCL files loaded into VRM constitute raw TCL code. While Mentor Graphics recommend using these loaded code fragments only for the purpose of overriding the documented user-definable procedures, in reality these code fragments can execute any legal TCL commands and, knowing the proper namespace keywords, can conceivably access any global or namespace variables in VRM code or in the underlying *vish* or *vsim* applications (*vish* if *vrun* is called from a shell command line or *vsim* if *vrun* is executed from within a running *vsim* process). In addition, while VRM attempts to catch both parsing and runtime errors caused by the execution of the documented user-definable procedures, there are an infinite number of runtime bugs that VRM cannot hope to detect. For example, user-supplied code can abort a VRM run, bypassing the end-of-execution reporting and cleanup, simply by executing the TCL *exit* command. The users must exercise the same caution writing TCL code to be loaded into VRM that they would in writing TCL scripts to control their simulations.

## Loading Conditional User-defined TCL Code

The *usertcl* element, like the *method* and *Runnable* elements, supports *if* and *unless* attributes for conditional loading. The values of these attributes are parameter-expanded and then evaluated as TCL expressions. The rules are the same as with the other conditional elements as follows:

- If an *if* attribute is specified, it must evaluate to true.
- If an *unless* attribute is specified, it must evaluate to false (true and false being interpreted in the same way as with other TCL expressions).
- If either attribute is missing, that condition is not checked.

- If the attribute-based conditions associated with a given *usertcl* element are satisfied, the code within the element is loaded. If not, the contents of the *usertcl* element are ignored.

It is also possible to include the condition within the TCL contents of a *usertcl* element as follows:

```
<usertcl name="cond1">
if {$::env(MODE) eq "noisy"} {
    proc RunManager::User::ActionCompleted {userdata} {
        upvar $userdata data
        echo "ActionCompleted: '$data(ACTION)', UCDB is '$data(ucdbfile)'"
    }
}
</usertcl>
```

or even within the overridden procedure itself as follows:

```
<usertcl name="cond1">
proc RunManager::User::ActionCompleted {userdata} {
    upvar $userdata data
    if {$::env(MODE) eq "noisy"} {
        echo "ActionCompleted: '$data(ACTION)', UCDB is '$data(ucdbfile)'"
    }
}
</usertcl>
```

However, in the event a given *usertcl* element is loaded that refers to one or more base *usertcl* elements, the base *usertcl* elements are considered and loaded even if the referring *usertcl* element has a non-passing condition. This is somewhat counter-intuitive but stems from the fact that *usertcl* elements are loaded and not searched. When a *usertcl* element is identified for loading (either through the auto-load mechanism, a command-line option, or via the base inheritance chain from another loaded element), its base elements are also eligible for loading, even if the original element is not loaded.

For example, consider the following RMDB database fragment:

```
<rmdb loadtcl="pos posfail">
<usertcl name="pos" if="1">
    ...TCL code...
</usertcl>
<usertcl name="posfail" if="0" base="posbase">
    ...TCL code...
</usertcl>
<usertcl name="posbase">
    ...TCL code...
</usertcl>
</rmdb>
```

The “pos” and “posfail” *usertcl* elements are eligible for loading because they are named in the auto-load list (*loadtcl*) in the *rmdb* element. The “pos” *usertcl* element loads normally because, even though it has a conditional attribute, that attribute evaluates to true. The “posfail” *usertcl* element, on the other hand, has a conditional attribute that does not evaluate to true. Therefore,

the code contained within the “posfail” attribute does not get loaded. However, the base *usertcl* element “posbase” does get loaded by virtue of being a base element of a *usertcl* element otherwise eligible for loading.

Note that even though this seems counter-intuitive, it is the exact same result one would get if the conditional check were embedded in the TCL code itself rather than in an attribute on the *usertcl* element. The reason for this behavior stems from implementation and how the generic inheritance walker is used in conjunction with *usertcl* elements.

## Collate Pass/Fail UCDB Files for Merge and/or Triage

---

In the course of running a regression suite using VRM features, one ends up generating a large number of UCDB, WLF, and other log files that must be analyzed. Coverage information must be merged and/or ranked in order to get a handle on how well the entire regression suite tests the design. Error messages must be triaged in order to pinpoint the unique error messages resulting from that run. From the point of view of the VRM tool, it all boils down to collecting simulation collateral files, sorting them by test status, and making those file lists available to the user's post-regression analysis script.

This document lists several methods for accomplishing this purpose. Each has different characteristics. The choices that have been tested on VRM are shown in [Table 10-5](#), along with some of the pros and cons of each:

**Table 10-5. Methods of Collating Passed/Failed UCDB Files**

Method	Pro	Con
Fully-automated merge and triage	Both merge and triage automated	Regression must follow VRM conventions
Gather UCDBs from the event log	Minimum setup	Merge/triage done at end of run
Auto-merge (override <code>OkToMerge</code> )	Merging done on-the-fly	Triage done at end of run, some interaction with rerun functionality
Maintaining separate file lists	Files/lists can be used elsewhere	Merge/triage done at end of run

Because of the flexibility of VRM, many more combinations are possible. These three are meant to be more of a guide to customizing the process to meet the needs of the given user environment than any recommended best practice.

Some of these methods involve either overriding user-definable procedures or calling utility procedures within *vrun*. The following links point to the detailed description of each of these procedures:

- “[OkToMerge](#)” on page 513
- “[MergeOneUcdb](#)” on page 513
- “[FetchEventUcdbs](#)” on page 568

These and other user-space procedures are documented in “[Catalog of User-Definable and Utility Procedures](#)” on page 499.

## Basic Regression Suite Example

The following describes the basic regression suite being used as an example in this case:

**Figure 10-16. Basic Regression Suite**

```

<rmdb>
  <runnable name="nightly" type="group">
    <parameters>
      <parameter name="top">mlm_sys</parameter>
      <parameter name="tbsrc">(%RMDBDIR%)/src/mlm_sys.sv</parameter>
      <parameter name="blksrc">(%RMDBDIR%)/src/mlm_par.sv</parameter>
      <parameter name="ucdbfile">(%pattern%).ucdb</parameter>
    </parameters>
    <members>
      <member>test1</member>
      <member>test2</member>
      <member>test3</member>
      <member>test4</member>
    </members>
    <preScript launch="vsim">
      <command>file delete -force work</command>
      <command>vlib work</command>
      <command>vlog (%tbsrc%)</command><!-- no coverage on testbench
          in order to avoid merge warnings -->
      <command>vlog +cover (%blksrc%)</command>
      <command>vopt (%top%) -o (%top%)_opt</command>
    </preScript>
    <execScript launch="vsim">
      <command>vsim -c -coverage -lib (%DATADIR%)/nightly/work
          +(%pattern%) (%top%)_opt</command>
      <command>run -all</command>
      <command>coverage attribute -name TESTNAME -value
          (%pattern%)</command>
      <command>coverage save (%ucdbfile%)</command>
    </execScript>
  </runnable>
  <runnable name="test1" type="task">
    <parameters>
      <parameter name="pattern">short</parameter>
    </parameters>
  </runnable>
  <runnable name="test2" type="task">
    <parameters>
      <parameter name="pattern">broad</parameter>
    </parameters>
  </runnable>
  <runnable name="test3" type="task">
    <parameters>
      <parameter name="pattern">shift</parameter>
    </parameters>
  </runnable>
  <runnable name="test4" type="task">
    <parameters>
      <parameter name="pattern">error</parameter>
    </parameters>
  </runnable>

```

```
</rmdb>
```

There are four tests in one group. Each test runs a different stimulus pattern based on a *plusarg* on the *vsim* command line. The *plusarg* value also serves as a basename for the UCDB file (for convenience). The *error plusarg* induces a simulation error so that test fails. The other four tests pass. Each simulation adds the *plusarg* value into the UCDB as the *TESTNAME* attribute and saves off the UCDB file that is used by VRM to determine the pass/fail status of the test in question.

The following sections describe several ways in which coverage can be merged and error messages triaged. The parts of the configuration that have not been changed or which bear no relation to the merge or triage flows are not shown in the examples.

## Fully Automated Merge and Triage

This section is not so much about gathering and classifying UCDB files as making such gathering unnecessary. VRM supports full automation of coverage merge and failure triage across an entire regression suite. The user scripts have very little to do other than producing the final reports, as the following example illustrates:

```
<rmdb version="1.1">
  <runnable name="nightly" type="group">
    <parameters>
      ...
      <parameter name="ucdbfile">(%pattern%).ucdb</parameter>
      <parameter name="mergefile">(%DATADIR%)/merge.ucdb</parameter>
      <parameter name="triagefile">(%DATADIR%)/nightly.tdb</parameter>
    </parameters>
    ...
    <preScript launch="vsim">
      ...
      <!-- These two OPTIONAL commands are recommended for safety (see
          below) -->
      <command>xml2ucdb testplan.xml (%mergefile%)</command>
      <command>triage dbfile -name (%triagefile%) -clear</command>
    </preScript>
    <postScript launch="vsim" mintimeout="60">
      ...
      <postScript launch="vsim" mintimeout="60">
        <!-- Generate coverage REPORT -->
        <command>coverage open (%mergefile%)</command>
        <command>coverage attribute -test * -name TESTNAME -name
          TESTSTATUS</command>
        <!-- Generate triage REPORT -->
        <command>triage report -name (%triagefile%)</command>
      </postScript>
    </runnable>
    ...
  </rmdb>
```

In this example, the *ucdbfile* parameter is used to point to the UCDB file produced by each test. In this case, the UCDB file is local to the working directory for the test and is named for the test

itself (assume that the *TESTNAME* is properly set in the test data record as in the basic example above).

Auto-merging of coverage data is enabled by the presence of a non-empty value in the *mergefile* parameter. The *mergefile* parameter points to the target merge file into which the coverage data is to be merged. VRM, by default, batches UCDBs that finish at approximately the same time so that there is never more than one merge running concurrently against any one target merge file.

Auto-triage of error message is enabled by the presence of a non-empty value in the *triagefile* parameter. The *triagefile* parameter points to the triage database into which the messages of a given test are to be added. As with auto-merge, the auto-triage function batches UCDBs so that there is never more than one merge running concurrently against any one triage database file.

Both auto-merge and auto-triage call user-definable procedures to implement the medium-level functionality. In the case of coverage merge, the *OkToMerge* (see [page 369](#)) procedure is called and various data values (including the path to the UCDB file) are passed. If this procedure returns true, the *MergeOneUcdb* (see [page 370](#)) procedure is called to perform the auto-merge. The default *OkToMerge* implementation returns true only if a UCDB for the test in question exists and the test data record indicates that the test passed.

In the case of failure triage, the *OkToTriage* procedure is called, again passing the path to the UCDB file). If this procedure returns true, the *TriageOneUcdb* procedure is called to perform the auto-triage. The default *OkToTriage* implementation returns true only if a UCDB exists for the test in question and the test data record indicates that the test failed (triage is primarily a tool for analyzing failed tests).

The *xml2ucdb* command in the *preScript* is used to “initialize” the target merge file. If the target merge file does not exist when the first of the auto-merge tasks is launched, and if there is only one UCDB to merge, that UCDB is simply copied to the target merge file. Another approach is to “initialize” the target merge file by importing a testplan into the target merge file, which causes a UCDB to be generated at that location. Likewise, the triage database can be initialized at the start of the regression run by putting a *triage dbfile -clear* command in the *preScript*.

Both of these can be omitted if the *VRMDATA* directory is known to be empty. The triage utility creates the triage database if none exists and the auto-merge function of VRM will use the first UCDB to be merged to a given target merge file to initialize the merge file. However, care must be taken to delete these files prior to subsequent regression runs or the data for that run will be merged with data left-over from previous runs.

Besides initialization of the target files, the only thing left for the user-specified Action scripts to do is generate the final reports at the end of the regression run. The commands in the *postScript* Action show some example report commands.

## Using the Event Log to Gather UCDB Files

The event log records certain information for every Action launched by VRM. One of the bits of information recorded is the path to the UCDB file (if the *ucdbfile* parameter is defined for the *execScript* Action in question). The following additions to the basic example allow the *postScript* Action for the “nightly” Group to extract two lists of UCDB files from the event log: one for passing Actions and one for failing Actions. These lists are then passed to *vcover merge* and *triage dbfile*, respectively:

```
<rmdb>
  <runnable name="nightly" type="group">
    <parameters>
      ...
      <parameter name="passlist" type="tcl">[FetchEventUcdbs -script
        execScript -status pass]</parameter>
      <parameter name="faillist" type="tcl">[FetchEventUcdbs -script
        execScript -status fail]</parameter>
    </parameters>
    ...
    <postScript launch="vsim" mintimeout="60">
      <command>triage dbfile (%faillist%)</command>
      <command>triage report</command>
      <command>vcover merge ucdb (%passlist%)</command>
    </postScript>
  </runnable>
  ...
</rmdb>
```

A set of utility procedures is defined in the “User” namespace of *vrun* (see “[Event Log Access Procedures](#)” on page 567). The *FetchEventUcdbs* procedure searches the event log for completed Actions that match certain criteria and returns the paths of the UCDB files associated with the matching Actions.

In the above example, all simulations are contained in *execScript* Actions so it is filtered for that action type. However, since the *preScript* and *postScript* Actions do not define UCDB files in this example, no UCDB path is returned for those Actions. Therefore, the *-script* filter option can be omitted and get the same result.

The advantage of this approach is that a minimum of setup is required. The simple TCL-based parameters can be used any time a script command calls for a list of passing UCDBs or a list of failing UCDBs (and both can be used in the same command to get a list of all UCDBs or just eliminate the *-status* filter on the *FetchEventUcdbs* procedure call). The disadvantage is that both the *vcover merge* and the *triage dbfile* commands must be run after the tests have all completed.

## Override OkToMerge and Rely on Auto-merge

The auto-merge functionality build into *vrun* allows a UCDB file produced by an *execScript* Action to be merged into a target merge file merely by defining certain parameters in the Runnable where the *execScript* itself is defined.

*MergeOneUcdb* (see [page 370](#)) user-defined procedure initiates the auto-merge process. It is possible to manipulate the value of the parameter that contains the target merge file in such a way as to cause multiple target merge files to be created according to the criteria of the component tests. The following example illustrates one way of accomplishing this:

```

<rmdb loadtcl="split-merge">
    <usertcl name="split-merge">
        proc OkToMerge {userdata} {
            upvar $userdata data

            if {[! [string equal $data(ucdbfile) {}]]} {
                if {[string equal $data(passfail) pass]} {
                    set data(mergefile) [file join [file dirname
                        $data(mergefile)] passed.ucdb]
                } else {
                    set data(mergefile) [file join [file dirname
                        $data(mergefile)] failed.ucdb]
                }
                return 1 ;# merge UCDB if it exists
            }
            return 0;
        }
    </usertcl>
    <Runnable name="nightly" type="group">
        <parameters>
            ...
            <parameter name="ucdbfile">(%pattern%).ucdb</parameter>
            <parameter name="mergefile">(%DATADIR%)/merge.ucdb</parameter>
        </parameters>
        ...
        <postScript launch="vsim" mintimeout="60">
            <command>triage dbfile (%DATADIR%)/failed.ucdb</command>
            <command>triage report</command>
        </postScript>
    </Runnable>
    ...
</rmdb>
```

In this example, the *OkToMerge* (see [page 369](#)) user-defined procedure is overridden. If the *ucdbfile* parameter is defined (meaning the Action is expected to produce a UCDB file), the pass/fail status of the Action is checked. Depending on whether the Action passed or failed, the value of the *mergefile* parameter is altered. Since the *MergeOneUcdb* (see [page 370](#)) procedure receives the modified *mergefile* parameter, the auto-merge process will merge UCDBs from passing tests into the *passed.ucdb* file and UCDBs from failing tests into the *failed.ucdb*. The *postScript* Action need only pass the *failed.ucdb* merge file to the *triage dbfile* command.

The advantage of this approach is that both passing and failing UCDBs are merged on-the-fly, as other tests in the suite continue to run. Moreover, since queued merging supports multiple concurrent merge Actions as long as they are to different target merge files, passing and failing tests can be merged concurrently, resulting in further efficiency. The disadvantage is that by adding UCDBs to the merge files on-the-fly, collisions can occur if the failed test rerun feature is used in the same regression run.

Overall, this is probably the most efficient flow if both coverage merge and failure triage are involved.

## Maintain Separate Lists of UCDBs Internal to VRM

Another approach is to compile two text files, one with a list of UCDB files from failed tests and one with a list of UCDB files from passing tests. These list files can then be used in subsequent *vcover merge* and *triage dbfile* commands. The following is one way such an approach might be implemented:

```
<rmdb loadtcl="split-merge">
  <usertcl name="split-merge">
    proc OkToMerge {userdata} {
      upvar $userdata data

      if {[! [string equal $data(ucdbfile) {}]]} {
        if {[string equal $data(passfail) pass]} {
          set file [PathRelativeToScratch passed.log]
        } else {
          set file [PathRelativeToScratch failed.log]
        }

        AppendToFile $file [list $data(ucdbfile)]
      }
      return 0 ;# no automatic merge
    }
  </usertcl>
  <runnable name="nightly" type="group">
    <parameters>
      ...
      <parameter name="ucdbfile">(%pattern%).ucdb</parameter>
    </parameters>
    ...
    <postScript launch="vsim" mintimeout="60">
      <command>triage dbfile -inputs (%DATADIR%)/failed.log</command>
      <command>triage report</command>
      <command>vcover merge merge.ucdb -inputs
        (%DATADIR%)/passed.log</command>
    </postScript>
  </runnable>
  ...
</rmdb>
```

In this case, the *OkToMerge* (see [page 369](#)) procedure has been overridden. It always returns a false value that disables all automatic merging within the *vrun* process. Instead, *OkToMerge*

adds the path to the UCDB file for the test at hand into one of two external files. The utility function *AppendToFile* (see [page 554](#)) is used to add the UCDB to the end of the appropriate list file. The list files are then passed to the *vcover merge* and *triage dbfile* commands via their respective *-inputs* option.

The advantage of this approach is that it is easy to implement and the file lists can be used for other purposes, like reports or an e-mail notification. The disadvantage is that both the *vcover merge* and the *triage dbfile* commands must be run at the end of the regression run, after all the tests have completed.

A variation on this idea stores the lists in TCL variables as follows:

```

<rmdb loadtcl="split-merge">
  <usertcl name="split-merge">
    proc OkToMerge {userdata} {
      upvar $userdata data

      variable passlist
      variable faillist

      if {[! [string equal $data(ucdbfile) {}]} {
          if {[string equal $data(passfail) pass]} {
            lappend passlist $data(ucdbfile)
          } else {
            lappend faillist $data(ucdbfile)
          }
        }
      return 0 ;# no automatic merge
    }
  </usertcl>
  <runnable name="nightly" type="group">
    <parameters>
      ...
      <parameter name="ucdbfile">(%pattern%).ucdb</parameter>
      <parameter name="passlist" type="tcl">$passlist</parameter>
      <parameter name="faillist" type="tcl">$faillist</parameter>
    </parameters>
    ...
    <postScript launch="vsim" mintimeout="60">
      <command>triage dbfile (%faillist%)</command>
      <command>triage report</command>
      <command>vcover merge merge.ucdb (%passlist%)</command>
    </postScript>
  </runnable>
  ...
</rmdb>
```

In this case, two TCL-based parameters are used to retrieve the lists when the *postScript* needs them. The lists of UCDB files are added to the *vcover merge* and *triage dbfile* commands as appropriate. The advantage of this implementation over the prior one is that no extra files are created (making the performance marginally faster). It has the same disadvantage as before in that the *vcover merge* and *triage dbfile* commands must be run after all the tests have completed.

## Enabling Auto-Delete

Auto-delete is enabled by overriding the user-definable procedure *OkToDelete* (see below for a detailed description of this procedure). The *OkToDelete* procedure is called upon the completion of every Action (except, of course, for the *deleteScript* pseudo-Action). The decision of whether or not to delete a given directory is based on a veto system. Each *deleteScript* pseudo-Action carries a flag that indicates whether or not the directory associated with that pseudo-Action is safe to delete. That flag is set to *yes* by default (that is, the directory is safe to delete). If the *OkToDelete* procedure returns a false value for any of the Actions associated with the same directory as the *deleteScript* pseudo-Action, that flag is set to *no*.

For example, suppose the *execScript* Action for *nightly/test2* passed and the *OkToDelete* procedure returned a true value (indicating there was no objection at that point to deleting the directory) but the subsequent *mergeScript* failed, causing the *OkToDelete* procedure to return a false value for that Action (indicating that the status of the *mergeScript* was such that the directory should not be deleted). The flag on the *deleteScript* associated with that Runnable would be set to *no* by the false return from *OkToDelete* when called on behalf of the *mergeScript* Action and the directory is not deleted. The type of Action and the pass/fail status are available to the *OkToDelete* procedure so that the decision of whether to delete or preserve the directory may be based upon the results of the various Actions executed within that directory.

Following is an example of a typical *OkToDelete* procedure:

```
proc OkToDelete {userdata} {
    upvar $userdata data

    passed -
    skipped -
    killed -
    empty {return 1}
    default {return 0}
}
```

This *OkToDelete* procedure allows deletion if the Action on whose behalf it was called either passed, was skipped (that is, not executed because of a prior error), killed (that is, by an external termination request), or if the Action was empty (that is, had no commands defined). These are all safe completion results which would not normally require inspection of the logs and/or other collateral files. If the status is anything other than the four values listed, the *OkToDelete* procedure returns a false value that prohibits deletion (presumably so the user could inspect the logs and/or other collateral files to determine the cause of any failures).

## Protecting Select Files from Deletion

Following are the ways to protect one or more files from deletion:

- By passing a file or path glob to the *DoNotDelete* utility procedure.

- By including a file or path glob in the *nodelete* parameter in the RMDB file.
- By saving the file to a directory in which no Action is executed (the top-most (%DATADIR%) directory would be a convenient choice).

Directories that are directly associated with Runnables (such as (%DATADIR%)/*nightly*, (%DATADIR%)/*nightly/test1*, and so on), a special flag file is written to the directory in order to prevent its accidental deletion. The auto-delete algorithm will not traverse into these directories while analyzing a parent directory for deletion. They can, however, be deleted as part of the auto-delete operation associated with the Runnable for which they were originally created.

## Using the DoNotDelete Utility Procedure

Files and/or directories may be protected from auto-deletion by passing them to the *DoNotDelete* utility procedure. The *DoNotDelete* procedure may be called from any user-definable procedure. It is passed one or more absolute or relative path arguments that may include glob-style wildcards. If one or more of the path arguments is not absolute, those arguments are assumed to be relative to (%DATADIR%) or to the most recent absolute or relative path passed as the value of the *-dir* option in the same *DoNotDelete* invocation.

Table 10-6 shows various *DoNotDelete* commands and the resulting absolute-path string that would be expanded. In this case, the parameter reference “(%DATADIR%)” represents the path to the VRM *Data* directory, as specified by the *-vrldata* option.

**Table 10-6. DoNotDelete Commands**

Command	Resulting Absolute Path
DoNotDelete *.log	(%DATADIR%)/*.log
DoNotDelete nightly/ test1/*.log	(%DATADIR%)/nightly/ test1/*.log
DoNotDelete /tmp/*.log	/tmp/*.log
DoNotDelete *.log -dir / tmp *.ucdb	(%DATADIR%)/*.log / tmp/*.ucdb

It is recommended that the *CONTEXT* built-in parameter (passed to the *OkToDelete* procedure via the *userdata* array) be used when referring to files located within the working directory for the Action on whose behalf the *OkToDelete* procedure has been called.

If any of the resulting absolute-path strings contains glob-style wildcards, then the wildcards are expanded at the time the procedure is called and the resulting paths are placed in the internal *NoDelete* list. If a given path has no wildcards or the wildcards expand to an empty list, then the path itself is placed in the internal *NoDelete* list.

Note that the above description implies that files to be protected via a wildcard-laden argument to the *DoNotDelete* procedure must exist at the time the *DoNotDelete* command is invoked. However, individual files may still be protected from deletion, even when they do not yet exist,

by passing their names to the *DoNotDelete* procedure explicitly (that is, without wildcards). Since the *OkToDelete* is called after all the merge-related and triage-related user-defined procedures, it is generally safe to protect collateral files generated from those procedures using wildcard-laden paths.

Following is how the *OkToDelete* example shown above might be modified to protect all the log and UCDB files in the working directory for the Action on whose behalf it was called:

```
proc OkToDelete {userdata} {
    upvar $userdata data

    DoNotDelete -dir $data(CONTEXT) *.log *.ucdb

    switch -- $data(passfail) {
        passed -
        skipped -
        killed -
        empty   {return 1}
        default {return 0}
    }
}
```

Note that if the *mergefile* parameter is defined and a *mergeScript* is executed, or if the *triagefile* parameter is defined and a *triageScript* is executed, the *mergefile* and/or *triagefile* paths will be added to the internal *NoDelete* list. It does not matter whether the *mergeScript* and/or *triageScript* passed or failed (though the pass/fail status could have a bearing on whether the source UCDB/WLF files are deleted).

Refer to “[“DoNotDelete”](#) on page 556 for additional information on the *DoNotDelete* utility procedure.

#### Using the *nodelete* Parameter

Files and/or directories may also be protected from auto-deletion by including them in a *nodelete* parameter in the RMDB. By default, the *nodelete* parameter is only consulted by the *deleteScriptpseudo* Action.

At the time a *deleteScriptpseudo* Action is launched, the *nodelete* parameter is expanded from the point of view of the *deleteScript* pseudo-Action and the resulting value is treated as a space/comma separated list of path tokens, each of which may be an absolute or relative path including glob-style wildcards. Non-absolute paths are assumed to be relative to the working directory that is about to be deleted. The resulting absolute paths are passed to the *DoNotDelete* procedure prior to initiating the deletion process.

## Disabling Auto-Delete from the Command Line

Even if the RMDB has redefined the *OkToDelete* user-definable procedure in such a way as to enable auto-delete, it is still possible to block auto-delete from deleting anything during a given regression run. If the *-noautodelete* command-line option is specified at the start of a regression

run, auto-deletion is disabled globally for the entire regression suite. The *deleteScript* pseudo-Actions are still generated and the *OkToDelete* user-definable procedure are not called and the *deleteScript* pseudo-Actions are ignored when they come up for execution.

## Listing Actions from the Command Line

The *vrun* command line supports *-dumpgraph* and *-dumpgraphviz* to list the Actions which would be executed rather than actually executing them. Neither of these options list the *deleteScript* pseudo-Actions, as these are not real user-defined Actions. Also, the *deleteScript* pseudo-Actions are not listed as “skipped” (and, in fact, are not skipped) as a result of errors earlier in the execution graph. They will, however, be listed as unhandled if the *StopRunning* user-definable procedure returns true and terminates the regression run, or if the run is prematurely terminated by a user interrupt (using Ctrl-C).

## Deletion Process

As mentioned above, the *deleteScript* pseudo-Action is handled entirely within the *vrun* process responsible for managing the regression run. Once pulled off the run queue, if the flag on the *deleteScript* pseudo-Action indicates that there is no objection to the deletion, the *deleteScript* pseudo-Action is placed in a deletion queue (to ensure all scheduled deletions take place before the *vrun* process exits) and an internal deletion procedure is scheduled for execution after a fixed time (5 seconds). The delay is to ensure that the Action that just finished in the directory slated for deletion has actually exited and closed all its file handles. Once started, the deletion of a single Runnable directory runs until completion (without a timeout). Subdirectories within the directory being deleted are also deleted recursively unless they are marked with the special *.VRM* tag file (which indicates that the directory is associated with a member Runnable and is already subject to a separate but identical auto-deletion process).

Once the files have been deleted, if the directory associated with the *deleteScript* pseudo-Action contains nothing other than the *.VRM* flag file, then the directory is also deleted.

A known anomaly can crop up if the *VRM Data* directory is accessed via an NFS mount. In the event that the *vish* or *vsim* process executing the Action wrapper file times out and fails to exit in response to the *kill* message from the managing *vrun* process, it is possible several script and/or log files will still be opened by the wrapper process at the time the *vrun* process attempts to delete them. In this case, NFS deletes the file but creates a hidden file (hidden on Unix/Linux, at least) whose name begins with *.nfs* and continues with a string of what appear to be randomly generated files. These files are general harmless. They are tracking files that should eventually go away when the wrapper process finally closes. The auto-delete algorithm attempts to delete these *.nfs* files but ignores any errors that may occur. The only real problem is that they will sometimes prevent an *rm -rf VRMDATA* command from completing. The first solution is to issue the command again, as some NFS servers will not delete the files until they are actually accessed. If that does not work, it may be necessary to look for a *vish* or *vsim* process that has failed to exit. As a last resort, the directory containing the *.nfs* files can be moved to another

location on the same file system (that is, under the same mount point) where they should clear up on the next reboot.

### OKToDelete

Refer to “[OkToDelete](#)” on page 517 for information on the *OkToDelete* user-defined procedure.

## Auto-Clean

The auto-clean function is enabled by default and requires no user intervention. Auto-clean deletes the old content of each working directory before the first script (*preScript* or *execScript*) runs in that directory. The purpose is to prevent the files from previous simulations from interfering with a new simulation (or providing bogus status and/or coverage data) when the same *VRM Data* directory is used for multiple regression runs.

For example, if a simulation that is supposed to generate a UCDB file fails in such a way that it does not write the UCDB file, it is possible *vrun* could mistake the old UCDB file for the current status. The auto-clean function can be disabled by including the *-noautoclean* option on the *vrun* command line.

The *-clean* command-line option causes *vrun* to completely delete the *VRM Data* directory prior to executing any Actions. This is a surefire way to prevent cross-regression interference but it also deletes any logs and history that may have been saved in the *VRM Data* directory.

## Auto-Delete and Auto-Triage

The auto-delete and auto-triage functions delete the merged UCDB file and the TDB file, respectively, when the first reference to each file occurs in a given regression run. This function can be disabled by including either the *-noautomergedelete* or *-noautotriageclean* options on the *vrun* command line (note that TDB files are “cleared” by issuing the triage dbfile *-clear* command).

A special corner-case exists when testplan auto-import is enabled and *-noautomergedelete* is specified. By default, the old merged UCDB file is deleted the first time it is referenced by auto-merge and, at that time, if testplan auto-import is enabled, the import is done directly to the path given for the merged UCDB file (and the initial test UCDB gets merged into the merged UCDB file rather than copied over). In the case where *-noautomergedelete* is specified, the testplan auto-import is blocked. The reason is that *vrun* assumes the merged UCDB file is left over from a previous regression run and already contains a copy of the testplan. If that is not the case, or if the testplan has changed, then *-noautomergedelete* should not be specified so that the merged UCDB file can be generated from scratch. After that, *-noautomergedelete* can be used to allow subsequent (and likely partial) regression runs to merge their coverage results into the existing merged UCDB file.

Note that auto-deletion of the merge/triage files does not take place at the start of a second pass initiated by the *-rerun* command-line option.



# Chapter 11 Reference

---

The vrun command provides command-line access to the Verification Run Manager.

<b>vrun Command</b> .....	<b>438</b>
<b>vrmcounter</b> .....	<b>459</b>
<b>Job Control Options Quick Reference</b> .....	<b>459</b>
<b>Message Verbosity and Debugging</b> .....	<b>460</b>
<b>Definitions of Terms</b> .....	<b>461</b>
<b>XML Terms</b> .....	<b>464</b>
<b>Typographical Conventions</b> .....	<b>465</b>

# vrun Command

The vrun command reads and records all command-line options before processing begins. In the case of options with the same semantic (that is, *-loadtcl*, *-G*, and so on), the semantic of the option can impose an order-based priority on the individual option values. Between options of different types and/or bareword arguments, there is no inherent order dependency.

## Syntax

### Execute named runnables (context chains)

```
vrun [-run] [<run_options>] <Runnable> ...
      <run_options>
      [-adhoc] [-apicmd<command>] [-ask] [-autotimeout]
      [-checkrmdb] [-clean | -realclean]
      [-dump] [-dumpgraph] [-dumgraphviz]
      [-exclude <Runnable>] [-exitcodes]
      [-faillog <filename>]
      [-g<name>=<value>] [-G<name>=<value>]
      [-historydb <path>] [-html] [-htmldir <path>]
      [-importelapsed <filename>] [-include <Runnable>] [-inctimeout <integer>] [-interactive]
      [-j <integer>]
      [-l <logfile>] [-loadtcl <tclfile>] [-loadtcl @<element>]
      [-maxmerge] [-maxrerun <count>] [-maxtriage] [-mergewait <sec>] [-minmerge <n>]
      [-mintimeout <[tQ:]tX>] [-mintriage <n>] [-mseed [<n> | random | last]]
      [-noautoclean] [-noautodelete] [-noautomerge] [-noautomergedelete]
          [-noautotimeout] [-noautotrend] [-noautotriage] [-noautotriageclean] [-nodtdvalidate]
          [-noexec] [-nogridmsg] [-nohistory] [-nolocalrerun] [-nolockmsg] [-noqueueemerge]
          [-noqueueutriage] [-noreusesseeds] [-normdbopts] [-nostderr] [-nosummary] [-notimeout]
          [-notplandele] [-noucdb] [-nousermsg]
      [-pipelaunch] [-project <projfile>] [-config <name>]]
      [-R<name>=<integer>] [-regex] [-rerun <args>] [-reusesseeds <status, ...>]
          [-rmdb <rmdbfile>] [-run] [-runlist <filename>] [-clear] [-runlog]
      [-savestdout] [-select <keyword(s)>] [-show [-actions] [-all]]
          [-showcmds] [-showparams]
      [-tail] [-tasklist <filename>] [-testname] [-timeoutmargin <percent>] [-triagewait <sec>]
      [-vrmdata <directory>]
      [-wrapperpath <path>]
```

### Invoke vrun in interactive (GUI) mode

```
vrun -gui [<gui_options>]
      <gui_options>
      [-do<dofile>] [-project <projfile> | -noproject]
```

**Display cumulative status of regressions executed in VRM Data directory**

```
vrun -status [<status_options>]  
<status_options>
```

```
[-all] [-columns<columnList>] [-covreport] [-covreportopts <argument_list>]  
[-cumulative] [-elapsed <filename>] [-errors] [-eventlog <logfile>]  
[-filter <expression>] [-full] [-hierarchical] [-host] [-html]  
[-htmldir <path>] [-nosummary] [-notruncate]  
[-reason] [-summary] [-tail] [-tcl] [-testname | -notestname] [-times]  
[-vrmdata <directory>] [-warnings]
```

**View (or tail) the log output of a specific action**

```
vrun -viewlog [<log_options>] <action> [<action> [...]]  
<log_options>
```

```
[-tail] [-vrmdata<directory>]
```

**Control ongoing (remote) regression run**

```
vrun (-pause | -continue | -suspend | -resume | -kill | -exit)  
[<action> [<action> [...]]]  
<control_options>
```

```
[-continue] [-dest<porthost>] [-exit] [-kill [<actions>]]  
[-pause] [-resume [<actions>]] [-suspend [<actions>]]  
[-testname | -notestname] [-vrmdata <directory>]
```

**Modify an active regression run**

```
vrun - modify [-vrmdata <path>] [-dest <porthost>] [-eventlog <name>] [<context>/  
][<keyword>=<value> ...]]
```

**Send predefined event (start, user, or done) to a listening vrun process**

```
vrun -send <event> [<event_options>] <action> [<action> [...]]  
<event_options>
```

```
[-code<integer>] [-dest <porthost>] [-message <string>] [-reply]
```

**Emit the designated information and immediately exit**

```
vrun [-help | -sample]
```

**Verbosity options (common to all modes)**

```
vrun [-debug[=<value>]] [-quiet] [-verbose]
```

## Arguments

### Execution Mode Options

- **-adhoc**

Bypasses member checking in context chains. See “[Adhoc Mode](#)” on page 108 for additional information.

- **-apicmd <command>**

Access RMDB data (use API commands), then exit. This option consumes all remaining options on the command line and sends the tokens as a command to the RMDB API, printing the result and exiting. See “[Access RMDB API via the vrun Command Line](#)” on page 138 for additional information.

- **-ask**

When you specify this switch, you are prompted for values of key parameters, based on attributes set in the RMDB database file. See “[Selecting Values for Embedded Options](#)” on page 115 for additional information.

- **-autotimeout**

Enables the auto-timeout functionality (see “[Timeouts](#)” for additional information).

- **-checkrmdb**

Perform checks on RMDB file.

- **-clean | -realclean**

**-clean** — Delete most of the contents of the VRMDATA directory prior to the regression run, only retaining:

- logs/ subdirectory and contents
- the VRMDATA directory itself

**-realclean** — Deletes the complete VRMDATA directory prior to the regression run.

Note that use of this option will recover any seeds to reuse them in the current run; use **-noreuseseeds** to disable this behavior.

Use the **-normdbopts** option with either **-clean** or **-realclean** to prevent execution of a new regression.

- **-dump**

Displays Runnables/Actions that are executed. It is similar to the **-show** option but is unformatted.

- **-dumpgraph**

Dumps an execution graph after analysis, which is human readable. This option blocks execution by default. When combined with the **-run** option, execution continues after requested data is displayed. See “[Message Verbosity and Debugging](#)” on page 460 for additional information.

- **-dumpgraphviz**

Dumps an execution graph after analysis, which is in graphviz format. This option blocks execution by default. When combined with the *-run* option, execution continues after requested data is displayed. See “[Message Verbosity and Debugging](#)” on page 460 for additional information.

- **-exclude <Runnable>**

Excludes a specified Runnable (context chain) from execution.

<Runnable> — the name of the runnable or the complete context chain to be excluded (see “[Context Chains](#)” on page 104).

See [Including and Excluding Context Subtrees](#) for usage details when combining overlapping *-include* and *-exclude* options.

- **-exitcodes**

Instructs **vrun** to report a bit-wise error code reporting on pass/fail status of the actions that are part of the regression run. Refer to the section “[Exit Code Status](#)” on page 70 for more information.

- **-faillog <filename>**

Writes failed task names to the specified log file. This option may include embedded environment variables.

- **-g<name>=<value>**

Defines a default value for a parameter.

<name> — the name of the Runnable whose parameter is to be changed.

<value> — the value to be assigned to that parameter. Parameters can be overridden with values containing an equal sign (=).

For additional information, see “[Override Parameter Values from Command Line](#)” on page 132.

- **-G<name>=<value>**

Defines an override value for a parameter.

<name> — the name of the Runnable whose parameter is to be changed.

<value> — the value to be assigned to that parameter. Parameters can be overridden with values containing an equal sign (=).

For additional information, see “[Override Parameter Values from Command Line](#)” on page 132.

- **-historydb**

Specifies an alternate location for the history database, normally saved to VRMDATA/logs/db. When you use this argument, the history database is stored in both the new location and the default location. You can prevent the creation of the database in the default location with the [-nohistory](#) argument.

- **-html**  
Generate the HTML status report on completion. For additional information, see “[Creating the HTML-based vrun Status Report](#)” on page 72.
- **-htmldir <path>**  
Set the HTML report path (HTML mode only). This option may include embedded environment variables. For additional information, see “[Creating the HTML-based vrun Status Report](#)” on page 72.
- **-importelapsed <filename>**  
Imports data from a file, <filename>, generated by the -elapsed argument to a vrun -status run. Refer to the section “[Timeouts](#)” for more information.
- **-include <Runnable>**  
Includes a given runnable (context chain) for execution (see “[Context Chains](#)” on page 104). See [Including and Excluding Context Subtrees](#) for usage details when combining overlapping *-include* and *-exclude* options.
- **-inctimeout <integer>**  
This option specifies an additional timeout value added to the execution timeout for each UCDB in the merge/triage queue.  
  
    <integer> — default is 120 seconds.  
  
The timeout used for queued auto-merge and auto-triage Actions is dependent on the number of UCDBs being merged and/or triaged. The incremental timeout per UCDB can be controlled globally by the *-inctimeout* command-line option to **vrun** or on a per-Action basis by adding an *inctimeout* attribute to the *mergeScript* and/or *triageScript* element in the RMDB.  
  
• **-interactive**  
(optional) Runs all Actions in interactive mode.
- **-j <integer>**  
Limits the number of concurrently running Actions. If the limit you specify is less than or equal to zero, then no limit is imposed on the number of processes that can be running at any one time.  
  
    <integer> — default for all platforms is 0 (unlimited)  
  
See “[Limit Concurrently Running Processes](#)” on page 92 for details. Also see “[Named Execution Queues](#)” on page 333.
- **-loadtcl <tclfile>**  
Loads file containing user extensions (TCL). Note that the *-loadtcl* option cannot appear in the *-rerun* option. This option may include embedded environment variables. See “[Loading User-supplied TCL Code Using usertcl into VRM](#)” on page 415 for additional information. Also see “[Environment Variables](#)” on page 489.

- **-loadtcl @<element>**  
Loads *usertcl* element containing user extensions (TCL).
- **-maxmerge**  
Maximum number of UCDBs included in an auto-merge pass.
- **-maxrerun <count>**  
Maximum number of immediate rerun passes. The default is 10.
- **-maxtriage**  
Maximum number of UCDBs included in an auto-triage pass.
- **-mergewait <sec>**  
Postpones beginning the merge operation until after <sec> seconds, where the default is 30 seconds.  
  
**<sec>**  
An integer defining the number of seconds to wait, where the default is 30.
- **-minmerge <n>**  
Postpones beginning the merge operation until at least <n> UCDBs have been queued for processing. The command will a set amount of time, as defined by the -mergewait option.
- **-mintimeout [[<tQ>:<tX>]**  
Specifies default minimum timeout(s) (in seconds). Default setting for *tQ* is 60 seconds and the default for *tX* is 300 seconds. For additional information, see “[Timeouts](#)” on page 311.  
If you set *tX* to 0, you will disable timeouts. However, you should instead use the -notimeout option.
- **-mintriage <n>**  
Postpones beginning the triage operation until at least <n> UCDBs have been queued for processing. The command will a set amount of time, as defined by the -triagewait option.
- **-mseed [<n> | random | last]**  
Specify a global master seed. The value will be used to initialize the random number generator in the TCL interpreter (the interpreter which executes the user-defined procedures will be used for all random number generation) immediately prior to expanding an RMDB file.  
  
**<n>**  
An integer value, where the default is zero (0).  
**random**  
Indicates that a random integer will be used as a global master seed.  
**last**

Indicates that the master seed used in the previous regression run is used. If a previous seed cannot be found, the default seed will be zero (0).

When you launch a VRM GUI session based on a vrun command with this argument, the master seed value will persist for all newly-generated VRM configurations.

- **-noautoclean**  
Disable automatic clean on working directories.
- **-noautodelete**  
Disable auto-delete algorithm.
- **-noautomerge**  
Disables auto-merge algorithm.
- **-noautomergedelete**  
Do not delete old merge file before auto-merge (see “[Automatic Deletion of Merge/Triage Files](#)” on page 394 for additional information).
- **-noautotrend**  
Disables auto-trend functionality (see “[trendScript](#)” for additional information).
- **-noautotriage**  
Disables auto-triage algorithm (see “[Automatic Deletion of Merge/Triage Files](#)” on page 394 for additional information).
- **-noautotriageclean**  
Do not delete the old Triage DataBase (TDB) file before auto-delete (see “[Automatic Deletion of Merge/Triage Files](#)” on page 394).
- **-nodtdvalidate**  
Do not validate RMDB file against Document Type Definition (DTD) in the release directory. The default is 0 (that is, validate). See “[Validation of the RMDB Database](#)” on page 63 for additional information.
- **-noexec**  
Generates files, but does not launch the scripts (see “[Dry Run of the RMDB Database](#)” on page 62 for additional information).
- **-nogridcheck**  
Disables pro-active grid checks. See the section “[Pro-active Grid Status Checks](#)” for additional information.
- **-nogridmsg**  
Disables messages when missing or failed grid jobs are re-queued by the pro-active grid checker.

- **-nohistory**  
Prevents the creation of the history database in the default location (VRMDATA/logs/db) when using the [-historydb](#) argument.
- **-nolocalrerun**  
Disables the vrun default of automatically re-queuing individual failed Actions.
- **-nolockmsg**  
Do not emit message for VRMDATA directory lock.
- **-nomergererun**  
Disables the automatic rerun for Actions where the role attribute is set to mergeScript or triagesScript, as defined in the section “[Spoofing Default Behavior of Action Scripts](#)”.
- **-noqueuemerge**  
Disables queued auto-merge activities (see “[Enabling and/or Disabling Coverage Merge Flows](#)” on page 396).
- **-noqueuetriage**  
Disables queued auto-triage activities (see “[Automated Results Analysis](#)” on page 398 for additional information).
- **-noreuseseeds**  
Do not use original seed values when rerunning test. See “[Reuse of Random Seeds in Automated Rerun](#)” on page 358 for additional information.
- **-normdbopts**  
Do not parse command-line options embedded in the RMDB file.
- **-nostderr**  
Ignore non-empty *stderr* files when analyzing the pass/fail status of any Action script. Useful for instances where scripts emit messages to *stderr* even though they pass. For fine control of this behavior, add the **usestderr** attribute to the Action element ([action Attributes](#)).
- **-nosummary**  
Suppresses the summary at the end of a regression run.
- **-notimeout**  
Do not check for Action script timeouts. For additional information, see “[Timeouts](#)” on page 311.
- **-notplandelete**  
Prevents a testplan from being deleted from any merged UCDB files when using the [-noautomergedelete](#) argument.

- **-noucdb**  
Instructs the vrun command to not read unnecessary UCDB files.
- **-nousermsg**  
Suppresses the reporting of user messages, from Action scripts, to the vrun log output.
- **-pipelaunch**  
Use pipes instead of background processes. Causes child processes to be launched via a pipe for better error handling at the expense of consuming one file descriptor for each concurrent Action.
- **-project <projfile> [-config <name>]**  
Read command-line options from a project file. This option may include embedded environment variables.  
  
    **-config <name>** — Configuration name.
- **-R<name>=<integer>**  
Overrides repeat count for a Task or Group. This option can also be used to override a parameterized repeat count, in which case the value passed to the *-R* option has precedence over the parameter-expanded value (even if overridden by the *-G* option). See “[Override Repeat Counts on Runnable Nodes](#)” on page 92 for additional information.
- **-realclean**  
Delete all of the contents of the VRMDATA directory prior to the regression run  
  
You can retain some information in the VRMDATA directory with the *-clean* option.
- **-regex**  
Expands context chains as regular expressions. See “[Using Regular Expressions in Context Chains](#)” on page 107 for usage details.
- **-rerun <args>**  
Reruns (a second pass) with specified arguments. Note that the *-rmdb*, *-vrmdata*, and *-loadtcl* options cannot appear in the *<args>* value of the *-rerun* option.  
  
See “[Failure Rerun Functionality](#)” on page 346 for additional information. See also, “[Fully Automatic \(One-command\) Mode](#)” on page 357 for additional information.
- **-reuseseeds <status>[,<status> ...]**  
Enables the selective reuse of seeds according to job status. This argument will be overridden if you specify *-noreuseseeds* on the same command line.  
  
This option accepts a comma-separated list of status values, which include:

```
all (default)
passed
failed
```

- **-rmdb <rmdbfile>**

Specify path to RMDB database file. The default file is *./default.rmdb*. Note that the *-rmdb* option cannot appear in the *-rerun* option. This option may include embedded environment variables.

- **-run**

Executes selected Runnables. This is the default mode of operation for *vrun* command.

By default, the Runnables selected on the command line are executed. Options that request data output (that is, *-dump*, *-dumpgraph*, *-dumpgraphviz*, *-runlist*, and *-show*) also block the default execution of the selected Runnables. When combined with the explicit *-run* option, execution continues after the requested data is displayed.

See “[Executing the Selected Runnables](#)” on page 113 for additional information.

- **-runlist <filename> [-clear]**

Reads/writes a runlist of selected runnables. The *-runlist* option blocks execution by default. When combined with the *-run* option, execution continues after requested data is displayed.

<filename> — a file path. This file stores the current include/exclude context paths. This file can also be edited and viewed by the user. The file can be read and written by the user, but only *-include* and *-exclude* options, one per line, are supported. Anything else in the file is discarded and an error is displayed.

-clear — Clear existing runlist contents. Note also that if this option is not included on the command line, then any contents already in the runlist is also a part of the regression run. See “[Build a Runlist of Runnables to Execute](#)” on page 112 for additional information.

The *-runlist* option allows you to build up a list of files to run via repeated *vrun* command invocations. Use the *vrun -runlist <filename>...* command several times to include or exclude subtrees of the regression, each time seeing the list of what was selected as a result, until satisfied with the final list. Then use the *vrun -runlist <filename> -run* command to run the list.

This option may include embedded environment variables.

See “[Build a Runlist of Runnables to Execute](#)” on page 112 for information to show what is selected in the current runlist without actually changing the contents of the list.

- **-runlog**

Save log output to a file. The *-runlog* option causes *vrun* to write a copy of the log output (normally written to the console) in a file under *VRMDATA/logs* whose base name matches the Status Event Log for that regression run and whose extension is *.log*. The log output file is only created if/when a regression starts. This file is used by the VRM GUI to monitor the output of regression runs that were not started by that particular GUI session.

- **-savestdout**

This switch instructs vrun to create a .stdout file for Action scripts. The information in this file is nearly identical to the information in the .log file. When you set an Action's **gridtype** attribute to a non-blank value, this switch is enabled by default.

- **-select <keyword(s)>**

Includes Runnables from previous run by status. See “[Semi-automatic \(Two-command\) Mode](#)” on page 354 for additional information.

See “[Selecting Runnables Based on Results of Previous Run](#)” on page 102 for further usage details of the **-select** option.

- **-show [-actions] [-all]**

Displays Runnables/Actions that will be executed. The **-show** option blocks execution by default. When combined with the **-run** option, execution continues after requested data is displayed.

See “[Showing which Runnables are Selected](#)” on page 100 for information on what will run if the **-show** option is not used.

See “[View RMDB Contents](#)” on page 110 for information on the list that determines what is displayed when the **-show** option is given.

**-actions** — Shows Actions instead of Runnables. See [Show Actions that will Run](#) for additional information.

**-all** — Shows all selected Runnables. See “[View RMDB Contents](#)” on page 110 for additional information.

- **-showcmds**

Same as the **-noexec** option and in addition, the **-showcmds** option displays the commands that would run (see “[Dry Run of the RMDB Database](#)” on page 62 for additional information).

- **-showparams**

Same as the **-showcmds** option and in addition, the **-showparams** option displays expanded parameter values (see “[Dry Run of the RMDB Database](#)” on page 62 for additional information).

- **-showtimeouts**

Displays the queue name and queuing and execution timeout values for each Action at the time the Action is queued, internal to vrun. Useful for analyzing the timeout computation. Refer to the section “[Timeouts](#)” for more information.

- **-simsample**

Produces a sample RMDB file, which includes a simulation

- **-tail**

Print log output of each script on standard output.

- **-tasklist <filename>**  
Reads list of tasks to be executed from file. Note that the *-tasklist* option may include embedded environment variables.
- **-testname <testname>**  
This option alters messages emitted by vrun to refer to testnames instead of Action context strings when you have defined a testname for the Action.
- **-timeoutmargin <percent>**  
Specifies the default maximum elapsed time margin for the auto-timeout calculation (see “[Timeouts](#)” for more information).
- **-vrldata <directory>**  
Specify root of VRM Data directory tree. Note that the *-vrldata* option may include embedded environment variables.
- **-wrapperpath <path>**  
Specify location of the vsim/vish executables to launch user scripts. Note that the *-wrapperpath* option may include embedded environment variables. Controls the value of VSIMDIR, one of the [Predefined Parameters](#).

#### GUI Mode Options

- **-do <dofile>**  
Execute the commands in *<dofile>*. Note that the *-do* option may include embedded environment variables.
- **-project <projfile>**  
Load the specified project file. Note that the *-project* option may include embedded environment variables.
- **-noproject**  
This option is useful when you supply all the necessary information on the command line. It allows you to turn the GUI into a single-session tool that forgets any information about the session upon exit. Specifically, it suppresses the following behaviors:
  - Loading of the default project file, default.vrm, if any exists.
  - Loading of and prompting for a project file at the beginning of the session.
  - Prompt to save the project data into a file when closing the session.

If you specify *-noproject* and *-project* on the same command line you will receive a warning and *-project* will be accepted.

Note that you can manually load or save project information through menu selection at any time.

### Status Mode Options

- **-status**

Displays the status of the latest regression run (even if the process is still underway). The “latest regression run” is determined by looking for the event log with the latest date. This means that in order to report the status of a previously initiated run, the **-vrmdata** option must point to the same directory as for the run whose status is under consideration. The status output attempts to report the pass/fail status of each Action as follows:

- Actions for which a *start* event exists with no corresponding *done* event are listed as Running.
- Actions that do not list a UCDB file in the *done* event are listed as simply Pass or Fail.
- Action for which a UCDB is defined (and can still be found) are listed according to the status in the UCDB test data record.

Actions that are not yet eligible for launch, or have become eligible but have not yet been launched, are not listed in the status report.

- **-all**

Reports the status of all known Actions including those that are still pending.

- **-columns {<column>[/[+|-]<width>]}...**

Customize status report columns.

Specifies the order and appearance of data columns in the status report, where you can specify multiple columns in a space or comma separated list.

You can control the width and alignment with the optional syntax:

/ — indicates you are supplying width and alignment information.

[+|-] — indicates the alignment, where “+” indicates right-alignment and “-” indicates left-alignment. “+” is the default if you specify neither. This setting is honored in HTML reports.

<width> — indicates the width of the column, in characters. This value is not honored in HTML reports, as your browser will determine the width.

See “[Creating the Text-based vrun Status Report](#)” on page 71 for further information and [Table 3-3](#) on page 82 for the full list of columns.

Example:

```
vrun -status -columns action/-20,hostname/+18,RMDB:seed/10
```

- **-covreport**

Generates a merged HTML-format coverage report at the same time as a HTML status report. This report is based upon the merged UCDB files referenced in the regression run, also shown in the Merged Coverage box of the Verification Run Manager Status Report.

The status report will contain links to this report. This option only applies when specified with -status and -html.

- **-covreportopts <argument\_list>**

Allows you to control details of the HTML-format coverage report. Valid arguments are the same as those to the vcover report -html option set, as defined in the *Questa SIM Reference Manual*. If you specify more than one argument, or any spaces at all, for argument\_list, be sure to enclose all the arguments in double quotes (" "). This option only applies when specified with -status, -html and -covreport. For example:

```
vrun -status -html -covreport
      -covreportopts "-htmldir mycoverage.report"
```

- **-cumulative**

Include the latest status of all runs in the VRMDATA directory.

Provides a cumulative report on all event logs in the VRMDATA directory. By default, only status of the immediately previous (or current) run is shown.

- **-elapsed <filename>**

Writes the elapsed time of passed actions to an external file <filename>.

- **-errors**

The *-errors* option causes the status report to show only errors. This option can be used together with the *-warnings* option to show both errors and warnings.

- **-eventlog <logfile>**

Read status from a designated Status Event Log. When the *-eventlog* is used with the existing *-status* option, it reports the status of a particular regression run as opposed to the latest regression run in the VRMDATA directory. The <filename> argument can contain the absolute or relative pathname of the log file to be opened, or it can contain the log file base name only (in which case the name is interpreted as being within the current VRMDATA directory). Note that the *-eventlog* option may include embedded environment variables.

- **-filter <expression>**

The *-filter <expression>* option allows the user to specify a filter expression (to the report output) that is used to select a subset of the Actions executed. See [Table 3-3](#) on page 82 for the full list of fields and value types.

- **-full**

Displays all available data columns for each event. See [Table 3-3](#) on page 82 for the full list of columns.

- **-hierarchical**

Display Action hierarchy, based on the hierarchy of the RMDB file (HTML mode only).

- **-host**

Displays the following data: date, hostname, seed, time, and username.

- **-html**

Generate report in HTML. For additional information, see “[Creating the HTML-based vrun Status Report](#)” on page 72.

- **-htmldir <path>**

Set HTML report path (HTML mode only). Note that the *-htmldir* option may include embedded environment variables. Any intermediate directories that do not exist are created as the report is generated. If the specified directory already exists, the HTML report files are written to that directory but any files already existing in that directory is preserved (unless explicitly overwritten by files from the current HTML report). For additional information, see “[Creating the HTML-based vrun Status Report](#)” on page 72.

- **-nosummary**

Omits the summary counts from the test-based status report.

- **-notruncate**

Do not truncate long Action context string.

Overrides the default truncation of a long Action context string. In this case, the complete value is shown in the column and, if the data value exceeds the column width, the following column starts in the appropriate place on a new line.

- **-reason**

Adding the *-reason* option to the *vrun -status* command-line causes the status report to include both the offending message and its simulation time. See “[Creating the Text-based vrun Status Report](#)” on page 71 for information on when a warning, error, or fatal message changes the value of the *TESTSTATUS* attribute in the test data record of the UCDB file.

- **-summary**

The *-summary* option causes the Action list to be omitted and only the pass/fail summary information is shown. The *-summary* option is ignored if the *-tail* option is specified, since the *-tail* option is designed to be used to follow a running regression’s status in real-time, in which case the status summary is not available until the regression finishes.

- **-tail**

Follow and report status on the ongoing regression run. When following VRM status events with the command *vrun -status -tail*, Crtl-C terminates the status following.

Reports the status events from a running *vrun* process as they occur. See “[Advanced Tasks](#)” on page 73 for additional information.

- -tcl

Generates the status report in TCL format. For example:

```
{nightly/preScript Empty 1352414455}
{nightly/directed/preScript Empty 1352414455}
{nightly/directed/dirtest1/execScript Ok 1352414470}
{nightly/directed/dirtest1/mergeScript Passed 1352414474}
{nightly/directed/dirtest2/execScript Ok 1352414472}
{nightly/directed/dirtest3/execScript Ok 1352414472}
{nightly/directed/postScript Empty 1352414476}
```

Where the format is: (<runnable>/<script> <status> <timestamp>

- [-testname | -notestname]

Forces the status reports to be in test-centric or action-centric mode. By default, vrun -status determines the mode based on your RMDB.

- -times

Displays elapsed CPU times in addition to date and time.

- -triagewait <sec>

Postpones beginning the triage operation until after <sec> seconds, where the default is 30 seconds.

<sec>

An integer defining the number of seconds to wait, where the default is 30.

- -vrldata <directory>

Specify root of *VRM Data* directory. Note that the *-vrldata* option may include embedded environment variables.

- -warnings

The *-warnings* option causes the status report to show only warnings. This option can be used together with the *-errors* option to show both errors and warnings.

## Viewlog Mode Options

- -viewlog <action>

The *vrun -viewlog <action>* command can be used to view the log output of any Action after the fact.

The *-viewlog* command accepts absolute paths, paths relative to the current directory, paths relative to *VRMDATA*, and Action context paths (in that order) when specifying which log file to view.

- -tail

Adding the *-tail* option to a regression run causes the Action log output to be printed to the *vrun stdout* in real time. The *-tail* is used to follow the log output as it is written.

- [-testname | -notestname]  
Forces the viewlog to be in test-centric or action-centric mode. By default, vrun -viewlog determines the mode based on your RMDB.
- -vrldata <directory>  
Specify root of VRM Data directory tree. Note that the -vrldata option may include embedded environment variables.

### Control/Abort Mode Options

- -continue  
Resume launching of new jobs.
- -dest <porthost>  
Monitor daemon socket (*port@host*). Specifies the destination (*port@host*) for the master VRM instance.
- -exit  
Terminate regression run (not running jobs).
- -kill [<actions>]  
Terminate regression run and running jobs.  
Note that during a VRM regression run, Ctrl-C terminates the run.
- -pause  
Suspends launching of new jobs.
- -resume [<actions>]  
Resume suspended jobs and launch new jobs.
- -suspend [<actions>]  
Suspend running jobs, releasing any licenses, to allow for the launching of new jobs.
- [-testname | -notestname]  
Forces the control mode to be in test-centric or action-centric mode. By default, the control mode executable will determine the mode based on your RMDB.
- -vrldata <directory>  
Specify root of VRM Data directory tree.

### Modification Options

The -modify command-line option group is mutually exclusive with the other mode options, such as -status and -send. It is not possible to launch a new regression and to control an existing regression with the same vrun command. Refer to the section [Regression Run Modification](#) for more information.

- **-vrldata <path>**  
Specifies the location of the regression run to modify. Use this argument if your VRMDATA directory is located elsewhere than the default location. The default value is the VRMDATA directory of the current run directory.
- **-dest <porthost>**  
Specifies the porthost location of the regression run you wish to modify. By default, vrun will find this value through analysis of the related event log, but you can specify the port if you already know the value.
- **-eventlog <name>**  
Specifies the name of the eventlog for the regression run you wish to modify. Use this argument for when you have multiple regressions running from the same VRMDATA directory.
- **<context>/**  
For Operational Attributes associated with named queues, this argument should be the name of the specific named queue, where any following **<keyword>=<value>** pairs apply only to the specific named queue.  
  
For Parameters, this argument should be the complete or partial context chain of one or more Action scripts.  
  
If you do not specify this argument, the **<keyword>=<value>** pairs are modified globally.
- **<keyword>=<value>**  
Specifies the attribute or parameter you wish to override. Refer to the [Parameter and Attribute Modification](#) for information about what you can override.

#### Notification Mode Options

- **-code <integer>**  
Report status code (0 => OK). Specifies a pass/fail return status. Default value is 0. In the case of the default, this means that 0 = OK; otherwise, the action has failed.  
  
The **-code** option allows the calling Action to return a status (integer and/or message string) to the invoking VRM.
- **-dest <port@host>**  
Monitor daemon socket (**port@host**). Specifies the destination (**port@host**) for the master VRM instance.  
  
The **-dest** option is mandatory in the notification mode. The value is normally hard-coded into the wrapper script by invoking VRM instance when a given Action is launched (the invoking VRM opens a socket for listening on the specified port).  
  
The bareword (non-option) arguments specify one or more Actions to which the event applies. Each argument must correspond to an Action that was launched by the VRM instance to which the event message is directed and which has not yet completed. Any

number of Actions can be included on one command line (assuming they all share the same status); however, only one *done* event per Action is accepted.

- **-message <string>**

Report message string. Specifies a status message (a string) that is printed to the invoking VRM log output.

The *-message* option allows the calling Action to return a status (integer and/or message string) to the invoking VRM.

- **-reply**

Report message reply on *stdout*.

- **-send {done | start | user}**

Notification mode is enabled by the *-send* command-line option. Sends a notification to communicate changes in job status to a VRM process. Generally, this command is not needed as the command is called from the same wrapper script used to launch the job in question.

- **done**

Sent once the user script completes and returns to the wrapper.

For *done* events, if the *-code* value is zero (0) and the *-message* value is blank (or not specified), then the invoking VRM assumes the Action passed normally.

If the *-code* value is zero (0) and a *-message* string has been specified, then the message is emitted to the invoking the VRM log output.

If the *-code* value is non-zero, then the Action is assumed to have failed. An error message is emitted to the invoking the VRM log output and the appropriate pass/fail handling is invoked.

In the case of a failure, the optional *-message* string is added to the error message emitted by the invoking VRM.

The following example command transmits a *done* message on behalf of a successful *execScript* Action in the *nightly/test1* Runnable:

```
vrun -send done -code 0 nightly/test1/execScript
```

- **start**

Sent just as the user script begins to execute.

For *start* events, both the *-message* and the *-code* options are ignored.

- **user**

Sent upon demand by the user script to communicate informational message.

For *user* events, the value of the *-message* option is used in an informational message emitted to the *vrun* log output and the *-code* option is ignored.

## Help Options

- **-help**

Print this help message and exit.

Note that it is required that the *-help* option be the first argument on the command line. If the *-help* option appears anywhere else, then the help text is internal to *vish* (which describes the *vsim* options) and the *vish* help is displayed instead of the *vrun* help. For example,

```
vrun -help // displays the vrun help  
vrun -noexec -help // displays the vsim help
```

- **-sample**

Prints a sample RMDB file on *stdout*.

## Verbose Options

Verbose options (common to all modes)

By default, if no verbose option is specified, then VRM runs the requested Actions, reports *start* and *done* events for each, reports relevant errors and warnings, and emits a summary of the run when complete. If some Actions are not run due to earlier errors and/or premature termination, then that fact is reported but the offending Actions are not listed.

See “[Message Verbosity and Debugging](#)” on page 460 for additional information.

- **-debug[=<value>]**

Print low-level information as *vrun* progresses.

<value>

A comma-separated list of values, defined as follows:

- *usertcl* — emits a message before and after any usertcl procedure is executed.

If you specify *-debug* without the optional “=<value>” option, then details about the internal flow of the VRM are emitted. This mode is mainly for use by developers and field support in locating problems with the VRM tool or its interpretation of the RMDB database. Debug-level messages are generally prefixed with the name of the TCL procedure reporting the message.

Debug mode, if specified, overrides verbose mode.

- **-quiet**

Suppress information messages during the run.

If the *-quiet* option is specified, then the VRM reports relevant error/warning and emits a summary of the run when complete. The *start* and *done* events are not reported.

- **-verbose**

Print additional information as *vrun* progresses.

If the *-verbose* option is specified, then additional user-level information, including the actual launch commands, about the progress of each Action is emitted. In addition, if some Actions are not run due to earlier errors and/or premature termination, then the offending Actions are also listed. These messages are in addition to those reported in default mode.

## Description

You can abbreviate VRM command-line options as long as enough of the option name is specified to unambiguously identify a single option keyword. For example:

**vrun -run -ad -as nightly**

shows enough of the *-adhoc* and *-ask* options for *vrun* to determine your intent.

## vrmcounter

This command set controls the counter configuration in the VRM Cockpit window and the counters in the VRM Results window status bar.

### Usage

```
vrmcounter load <filename>
```

```
vrmcounter save <filename>
```

```
vrmcounter show
```

### Arguments

- vrmcounter load <filename>  
Loads the vrmcounter Configuration File from the specified location.
- vrmcounter save <filename>  
Saves the existing vrmcounter configuration to the specified location.
- vrmcounter show  
Displays the current vrmcounter configuration.

### Description

Refer to the section “[Counter Control in the VRM GUI](#)” for more information.

## Job Control Options Quick Reference

The following *vrun* command-line options are used to control a remote *vrun* process and/or the jobs running under that process. The *vrun* process receiving one of these options connects to a remote *vrun* process via its status event socket and issued a command of the same name as the command-line option (that is, *vrun-exit* issues an *exit* command via the socket to the remote *vrun* process).

The remote *vrun* process may be specified with the *-dest* command-line option; otherwise, the *port@host* is determined by examining the most recent Status Event Log in the specified *VRMDATA* directory.

**Table 11-1. vrun Job Control Options Quick Reference**

Option	Effect on Existing Job	Effect on Regression Run	Comments
-exit	None	Run is terminated	
-pause	None	New job launching is paused	
-continue	None	New job launching is resumed	

**Table 11-1. vrun Job Control Options Quick Reference (cont.)**

Option	Effect on Existing Job	Effect on Regression Run	Comments
-kill	Kill all existing jobs	Run is terminated	Also issued for Ctrl-C or if StopRunning returns true
-kill <actions>	Kill selected jobs	No effect	Also issued on Action timeout or expiration of wall-clock run limit
-suspend	Suspend all existing jobs	New job launching is paused	
-suspend <action>	Suspend selected jobs	No effect	
-resume	Resume all existing jobs	New job launching is resumed	
-resume <action>	Resume selected jobs	No effect	

The operations specified in the “Effect on existing jobs” column are implemented differently for jobs launched locally and jobs launched on the grid. The <gridtype> is specified in the *method* element and the procedures themselves can be overridden via an appropriate *usertcl* element.

**Table 11-2. Jobs Launched Locally vs. Jobs Launched on Grid**

Operation	Local Job	Grid Job
Kill	Sends KILL signal	Calls <gridtype>KillJob
Suspend	Sends STOP signal	Calls <gridtype>StopJob
Resume	Sends CONT signal	Calls <gridtype>ResumeJob

## Message Verbosity and Debugging

All messages are produced on the standard output of the VRM process or in the transcript of the invoking *vsim* session.

It is recommended that verbose mode be used when debugging the configuration of a regression suite.

It is recommended that the default mode be used when running as an automated (nightly) regression run.

As an additional configuration debug tool, the *-dumpgraph* option dumps the contents of the execution graph upon generation. This information is useful as it shows how the Actions depend on each other, as well as the results of any repeat expansion. Each Action in the graph is listed in alphabetical order along with a list of other Actions that must complete before that Action is

eligible to run. A variation on this option is the *-dumpgraphviz* option that causes VRM to emit a data file suitable for use as input to the Graphviz *dot* program (Graphviz is an Open Source graph drawing package). With either options, VRM exits immediately after producing the dump output.

In some complex regression runs (involving regular expression selection of top-level Runnables, repeating Tasks and Groups, and/or conditional membership), it can be difficult to know exactly what will run once the regression is kicked off. In order to get an expanded list of the Runnables that will be executed and in what context, the *-show* option to *vrun* can be used to display a list of selected Runnables after expansion of the context chains on the command line.

See “[Showing which Runnables are Selected](#)” on page 100 for additional information.

## Definitions of Terms

The following terms are used throughout this manual to refer to specific objects with the VRM environment.

### Action

A single script that represents one step in the regression suite. Leaf-level Runnables map to a single *execScript* Action. Group Runnables map to two Actions: a *preScript* Action that executes prior to the members of the group, and a *postScript* Action that executes after the members of the group have been executed.

### Calling Chain

The nesting history leading to the execution of a given Action (see “[Context Chains](#)” on page 104 for more details).

### Collateral Files

In the context of this document, collateral files refer to the files produced as a result of the execution of one or more Actions from a regression suite. The term sometimes also includes script files generated by the *vrun* application in the process of executing those Actions.

### Configuration

A pre-canned command-line template for launching regression runs. Each configuration is associated with a single RMDB file and generally contains a pattern of Runnables from the RMDB that are selected for execution and a list of command-line options to be passed to the *vrun* application.

### DTD

Document Type Definition.

## Group

A non-leaf node in the regression suite representing a group of Task or nested Group nodes.

## GUI Session

The period of time starting from a single invocation of the Graphical User Interface (GUI) and ending when that invocation is closed.

## Merge File

A UCDB file that contains the merged results of one or more regression tests and possibly one or more testplans.

## Parameter

A name/value pair owned by a Runnable and defined in the RMDB database or on the command line. A parameter is nothing more than a name/value pair that allows multiple Tasks to share the same execution script (or multiple Groups to share the same Task) by providing unique per-Group or per-Task values to be used in the scripts.

## Parameter Reference

A unique combination of characters (in this case, “(%somename%)”) used to parameterize a script command or a parameter value.

## Project File

A file containing one or more pre-canned configurations. The preferred extension of a project file is *.vrm* (which is why it is also referred to as a VRM File).

## Regression Test

One atomic simulation run represented by a Task Runnable in the RMDB database and resulting in, at most, one UCDB coverage file.

## Regression Run

A single batch-mode instance of the *vrun* application in which part or all of a regression suite is executed.

## Results

A term used to refer to the pass/fail status of the Actions executed as part of the regression suite (and sometimes inaccurately used to refer to the contents of the *VRMDATA* directory).

## Results Analysis

Results Analysis is a Mentor Graphics product. The command and the database that is created are called triage. This means that when referring to the process, Results Analysis is used; when using commands, it refers to creating a triage database for results analysis (see “[Automated Results Analysis](#)” on page 398).

## RMDB

Verification Run Manager DataBase (RMDB) file (see [Example 3-1](#) on page 60). The RMDB file is the VRM database.

## RMDB File

An XML file containing descriptions of the various tests and other steps that make up a regression suite. The preferred extension of an RMDB file is *.rmdb*.

## Runnable

A single node in the hierarchy topology of a regression suite. A Runnable may represent a compile step, a single simulation (that is, a single test), or a group of other Runnables.

## Status Event Log

A file generated by the *vrun* application in which every status change is recorded. In theory, this file contains the complete history of a single regression run.

## Task

A leaf-level node in the regression suite, typically associated with a simulation run.

## Target Merge File

A merge file that has been designated as the UCDB file into which the coverage results of a given regression test are to be merged.

## Test-level UCDB File

This term refers to a UCDB file generated from a single regression test. It contains a single testdata record for that test and no testplan.

## Triage

Triage is the command name and database that is created for results analysis (see “[Results Analysis](#)” on page 463 and “[Automated Results Analysis](#)” on page 398 for additional information).

## UCDB

Unified Coverage DataBase. UCDB API is an application programming interface for the Unified Coverage Database included in the Questa® SIM and ModelSim SE products. The UCDB and its API are completely independent of Questa and ModelSim, however UCDBs are easily created with these tools.

## VRM Data (or VRMDATA) Directory

The term used to refer to the directory tree under which a given regression run stores its script and collateral files, or to the top-most directory of that tree. This used to be called the *SCRATCH* Directory in versions prior to 10.1.

## Working Directory

A directory created and maintained by VRM and corresponding to a particular Task or Group of Tasks in a regression run. All Actions are executed in a working directory in order to prevent collateral files from multiple VRM Actions (or multiple regression runs) from colliding.

# XML Terms

The following terms are used to reference parts of the XML database format. The terms and their definitions are inherited from the XML specification.

## Attribute

A piece of meta-data attached to an element to describe more about the element and/or its behavior.

## Element

A semantic object in the RMDB database. An element can contain other nested elements, user data, or meta-data (information about the element itself). Each element has a unique name, based on the tag used to mark the boundaries of that element in the database file format.

## Tag

A string representing the start or the end (in some cases both) of a semantic *object* (called an *element*) in an XML file.

# Typographical Conventions

Certain typographical conventions are used to indicate when a word is being used for its VRM specific definition or simply as an English word. In general, the following rules apply:

- When referring to a node in the RMDB database or in the execution graph that the VRM uses to schedule jobs (that is, Runnable, Group, Task, Action, and so on), the word used is capitalized.
- Words used as attribute values, parameter names, or parameter values (for example, the “fred” Group) are enclosed in double quotes (" ").



# Chapter 12

## Communication Channels, Event Log, and Status Report

---

The communication of VRM relies on these various topics.

Action Script to VRM .....	467
Unattended Batch Mode .....	468
Communication Channels.....	469
Event Logging .....	471
Attach to a Running <i>vrun</i> Process via Event Log.....	473
Send Status Message from Wrapper/User Script.....	475

## Action Script to VRM

The *vrun* command always opens a TCP/IP socket on which it listens for status messages from the Action scripts it launched. This channel supports the following message types:

- Action script started
- Action script completed (done)
- User-defined log messages

This channel operates in one direction only. The “start” and “done” messages originate in the Action script wrapper and are implemented as calls to *vrun* with the *-send* command-line option. The *port@host* information is hard-coded into the wrapper script when the wrapper is generated. The connection is open only long enough to convey the status, at which time the socket is closed in order to prevent system resources in the *vrun* process from becoming exhausted when a large number of concurrent Actions are launched.

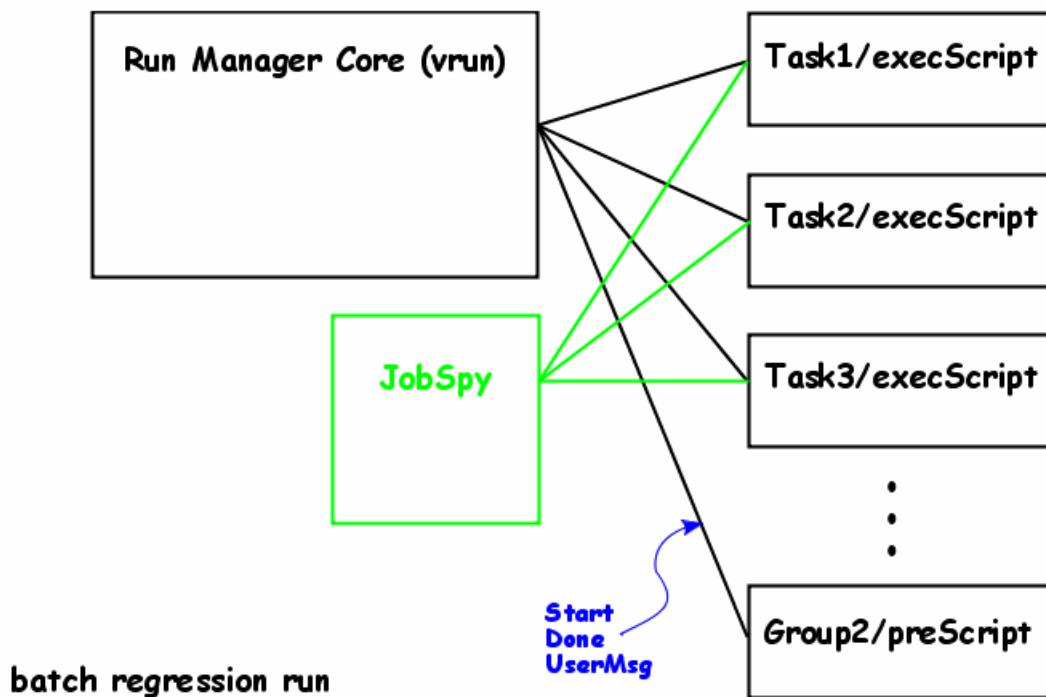
User-defined messages are typically initiated by the user's script either calling a TCL function in the Action script wrapper (for TCL scripts interpreted by *vsim*) or by calling *vrun* directly with the proper arguments (for scripts executed in a shell or other interpreter). In either case, these are one-way asynchronous messages that the Action script sends to the launching *vrun* process. The only use for these user-defined messages is to emit them to the VRM log output.

## Unattended Batch Mode

In this mode, the *vrun* command is executed from a shell or from the *vsim* transcript window. There is no interaction with the user once the process is kicked off. VRM receives status from the Action scripts and maintains internal tables representing the state of the regression suite. All status output from VRM is via the log output (*stdout*). The only way to stop the regression run is to kill the *vrun* process.

The only communication channel is the one-way status channel via which the *done* signal is sent. A parallel JobSpy process can also be set-up to monitor any simulations that can be kicked off as a result (in this case, assume that the three *execScript* Actions represent simulations. The JobSpy channel allows two-way communication between the monitor GUI and each simulation, but does not provide visibility into non-simulation processes launched as part of any Action script. [Figure 12-1](#) is a diagram illustrating the unattended batch regression run.

**Figure 12-1. Batch Regression Run**



# Communication Channels

The “core engine” of VRM is the *vrun* application. When managing a regression run, the *vrun* application is essentially a batch-mode process. It launches jobs and reports their status on a text-based log stream (via *stdout* or via the *vsim* transcript stream if launched from *vsim*). A single *vrun* process is responsible for a single regression run (although *vrun* can call additional instances of itself, depending on the user’s configuration, these child processes are also considered separate regression runs). The only direct control the user has over the regression process is through the RMDB database file and by way of command-line options specified at the time the *vrun* process is launched.

To provide runtime monitor and control functions, VRM defines three logical communication channels for communicating job progress and/or control. Each of these logical channels carries data in a single direction only as follows:

- The first logical channel carries status information from the running Action scripts (jobs) to the launching *vrun* process.
- The second logical channel carries status change events from the *vrun* process to the Runtime GUI.
- The third logical channel carries control commands from the Runtime GUI to the *vrun* process.

None of these channels make use of, or interfere with, the JobSpy communication mechanism (that is, JobSpy can be used independently of, and concurrently with VRM).

The term “Runtime GUI” refers to any process that is used to monitor or control the *vrun* batch process. In the most common case, this is the VRM window of the Questa GUI but other applications can also connect to the *vrun* process and make use of the same monitor/control channels by following the protocol described below.

Before launching any Action scripts, the *vrun* process opens a TCP/IP socket on which it listens passively for connections. The port number selected for this socket is published and all three logical channels use this same physical socket. The data content transferred via these connections depends on which logical channel is being invoked. The sections to follow describe the protocol of the communication.

<b>Basic Communication Model .....</b>	<b>469</b>
<b>Format of Messages from Action Script to vrun Core.....</b>	<b>470</b>
<b>Error Response .....</b>	<b>471</b>

## Basic Communication Model

The communication channel format follows a text-based model. Each “message” consists of one or more text records delimited by a newline character (that is, each message looks like a

single line of text). Messages to the *vrun* process always begin with a single-token keyword and contain a variable number of arguments. All keyword/argument tokens are passed in TCL list form (escaped as necessary). This helps reduce conversions from one format to another under various circumstances. For example, the TCL list can be passed as arguments to a TCL procedure, serialized for transmission over a socket, or encoded onto a *vrun* command-line for user-level access. All messages occur as the result of some event (that is, a state change in an Action (or possibly in VRM itself) or a user request).

## Format of Messages from Action Script to vrun Core

Status messages from an Action script to the *vrun* core follow a fixed format that looks as follows:

```
<event> <hostname> <status> <text-message> <action-path> [<action-path>  
[...]]
```

The “event” field should contain one of the strings from [Table 12-1](#).

**Table 12-1. Event Field Strings**

Event	Event/Description
start	Action script has started (sent before Actionscript starts).
user	User-defined text message from Action script.
done	Action script has completed (sent afterAction script completes).

Each properly functioning Action sends a single *start* message just before the user script starts running and a single *done* message just after the user script completes. These messages are sent from the TCL wrapper script that accompanies each Action. The user can send one or more optional *user* messages, each of which contains a message string that is relayed to the log output of the *vrun* process. A timeout can be specified from the time the Action is initially launched until the time the *start* message arrives and another timeout from the time the *start* message arrives until the *done* message arrives.

The “hostname” field should contain the name of the machine on which the Action script is running (the return value of the *info hostname* TCL command).

The “status” field contains a numeric value that is zero if the Action passed and non-zero if it failed (the actual value is generally determined by the Action script via the wrapper).

The “message” field is a free-form text message passed as a TCL-delimited string.

If any of these fields are empty or undefined for a given message, then a TCL empty string ({} ) appears in place of the missing value.

The remaining tokens in the message are interpreted as a list of Action script context paths to which the message applies. If the *vrun -send <event>* command is used to send the messages, then there is only one “action-path” token per message. But the receiving *vrun* process will accept any number of action-path tokens in a single message.

When a message is sent via this logical channel, the sender connects to the listening *vrun* application via the designated TCP/IP port, posts a single-line message to the resulting open socket, and then closes the socket. This “burst-mode” communication makes it less likely that system resources will be overwhelmed by a large number of concurrent Action scripts all starting or completing at the same time. In fact, the single-threaded nature of the *vrun* handling code implies that only one such event message can be transferred at any given time. For this reason, *vrun* performs as little processing as possible while the connection is open.

## Error Response

If an error is detected in a message received by *vsim* via the open event message socket, the single response “Eh?” will be returned. This is just a human feedback mechanism, as the messages in each case do not involve a handshake mechanism. The *vrun* application ignores messages with “Eh?” as their first token, in order to prevent a listening application from starting a feedback loop of error messages between itself and *vrun*.

## Event Logging

Every status event message generated by the *vrun* core is also logged in an event log. The event log is created in a directory called *logs* directly under the head of the VRMDATA directory for that regression run. The name of the log file is derived from the date and time when the regression run started, with the extension of *.rmev*. The file names are formatted such that a simple lexical sort yields the sequence in which the individual regression runs were executed. Any application that knows the path to the head of the VRMDATA directory can discover the event logs from all past regression runs involving that VRMDATA directory.

The event log is made up of the messages passed from the *vrun* core to the Runtime GUI. Each status event message occupies one line in the file. A sample event log file might look like [Example 12-2](#).

### Figure 12-2. Event Log File Sample

```
20090402T081950 begin 39096@myhost /path/to/simple.rmdb /path/to/VRMDATA
20090402T081950 eligible nightly/preScript
20090402T081950 launched nightly/preScript
20090402T081951 running nightly/preScript myhost
20090402T081951 done nightly/preScript passed myhost {} {} {}
20090402T081951 eligible nightly/test1/execScript
20090402T081951 eligible nightly/test2/execScript
20090402T081951 eligible nightly/test3/execScript
20090402T081951 launched nightly/test1/execScript
20090402T081951 launched nightly/test2/execScript
20090402T081951 launched nightly/test3/execScript
20090402T081952 running nightly/test2/execScript myhost
20090402T081952 running nightly/test1/execScript myhost
20090402T081952 running nightly/test3/execScript myhost
20090402T081953 done nightly/test2/execScript passed myhost
    nightly/test456/test456.ucdb ok 456
20090402T081953 done nightly/test1/execScript passed myhost
    nightly/test123/test123.ucdb ok 123
20090402T081953 done nightly/test3/execScript passed myhost
    nightly/test789/test789.ucdb ok 789
20090402T081953 eligible nightly/postScript
20090402T081953 launched nightly/postScript
20090402T081954 running nightly/postScript myhost
20090402T081954 done nightly/postScript passed myhost {} {} {}
20090402T081954 end
```

This event log shows that three leaf-level Tasks (instances of *execScript*) were initiated and that all three passed. Each was a simulation that resulted in a UCDB file and the user can construct the absolute paths to these UCDB files by prepending the *ucdb-file* field of each *done* message with the value of the *scratch-dir* from the *begin* message. Also, it is possible to find the RMDB database file used for this run by looking at the *ucdb-file* field of the *begin* message.

When the TCL wrapper scripts for each Action are generated by *vrun*, the *port@host* of the listening TCP/IP socket is hard-coded into the script. However, when external applications, such as the Runtime GUI, want to connect to a running *vrun* process in order to report ongoing status, the *port@host* field of the *begin* message should be consulted. Since the event logs are stored in a fixed location relative to the head of the directory, and since the latest event log file will always be the last file in a lexical sort of filenames, all the external application needs to know in order to locate the open TCP/IP port on the *vrun* process is the path of the VRMDATA directory in which the tests are being executed.

Moreover, since the event logs are always stored in a fixed location relative to the head of the VRMDATA directory, and since an event log is generated for every regression run, the cumulative event logs can be read in sequence as a single stream to find out what has run in that VRMDATA directory since the directory was created (assuming nobody deleted the log files without also deleting the entire VRMDATA directory). In order to develop a report on the current status of every test in the suite, the user needs to only read the event log files in sequence, keeping only the latest status event message for each Action.

In order to speed up the status reporting process in this case, at the end of each regression run, the *vrun* process checks the event log directory to see how many logs have accumulated. If more than one log file exists in *VRMDATA/logs*, a roll-up file will be created containing the cumulative status of all Actions to date. The name of the roll-up file follows the same pattern used for the event log files, save that the extension used is *.rmup*. The roll-up file contains a verbatim copy of the most recent status event message logged for each Action (only *running* and *done* events are considered). The event messages are stored in the roll-up file in an arbitrary order. In order for the roll-up file to appear like an event log, the most recent *begin* message from the set of event log files (or a surrogate, should none of the event logs files contain a *begin* message) will precede the Action-related status event messages. A roll-up file always ends with an *end* message bearing a timestamp based on when the roll-up log was created.

To generate a cumulative status report, the names of the files in the *VRMDATA/logs* directory should be lexically sorted. Then, all files prior to the latest roll-up (*.rmup*) file can be discarded (since the cumulative effect of the event log files prior to that are already reflected in the latest roll-up file). The files should then be read in sequence, recording each message in a hash keyed by the context chain string associated with each status event message (each message would replace any messages for that Action that were previously seen). The result can then be sorted as necessary and rendered into a report.

Note that the contents of the event log generated during a regression run are identical with the data emitted on the status channel. Internally, the event log is nothing more than a listener on the status channel whose socket handle just happens to be a file handle that is opened for writing any time *vrun* begins to execute a regression suite.

## Attach to a Running vrun Process via Event Log

Using the examples in this chapter you can learn how to attach to a process using its event log.

Looking at [Example 12-2](#) on page 472, the user can infer from the presence of an *end* message at the end of the file that this regression run has finished. However, if there is an incomplete file as shown in [Example 12-3](#), then the user can infer that the regression run from which this log file was emitted is (probably) still running. In order to verify that fact (and connect to the *vrun* process in order to receive further status event messages directly), the user can look at the *port@host* field of the *begin* message. This shows the host name and port number of the listening socket of the *vrun* process itself. If the user is able to successfully connect to that socket, there is a good chance that this regression run is still in progress (as opposed to having crashed without finishing the event log).

### Figure 12-3. Incomplete Event Log File

```
20090402T081950 begin 39096@myhost /path/to/simple.rmdb /path/to/VRMDATA
20090402T081950 eligible nightly/preScript
20090402T081950 launched nightly/preScript
20090402T081951 running nightly/preScript myhost
20090402T081951 done nightly/preScript passed myhost {} {} {}
20090402T081951 eligible nightly/test1/execScript
20090402T081951 eligible nightly/test2/execScript
20090402T081951 eligible nightly/test3/execScript
20090402T081951 launched nightly/test1/execScript
20090402T081951 launched nightly/test2/execScript
20090402T081951 launched nightly/test3/execScript
20090402T081952 running nightly/test2/execScript myhost
```

Once the connection is established, the listening application should issue the *status* command to the *vrun* instance. This will cause the connection to be added to the list of status event listeners, thus ensuring that any further event messages are sent to the listening application in addition to the event log. Therefore, it is not necessary for the listening application (probably a GUI or other status monitor) to continually check the event log file for updates.

Note that a significant interval of time can elapse between the time the listening application reads the event log (in *VRMDATA/logs*), discovers the lack of an *end* message, reads the *port@host* field from the *begin* message of the event log, opens the channel, requests to be added as an event listener, and the time new status events finally start arriving. In order to avoid missing events (which arrive asynchronously) during this interval, the *status* message also causes the *vrun* process to open the event log for reading and echo on the channel all the event messages generated up to that point.

In order to conserve network bandwidth, the *status* message can carry an optional *file-position* argument as follows:

```
status <file-position>
```

The *file-position* is a (long) integer representing the number of bytes of the event log file that were successfully read. If a non-zero *file-position* argument is given, the *vrun* process will skip that many bytes of the event log before reading it back over the communication channel to the listening application. Since those bytes were already read from the event log file before the connection was opened and the status was requested, the end result would be that only messages added since the listening application finished reading the event log will be read back over the communication channel, thus eliminating redundancy. Since the *status* command is handled in the same thread as the Action status messages, the protocol still prevents asynchronous status event messages from being dropped.

# Send Status Message from Wrapper/User Script

The *vrun* command line supports multiple modes. One is the default “run” mode in which *vrun* launches new jobs. Another is a “send” mode in which *vrun* simply sends a message to a specified port on a specified host and exits. This “send” mode is used by the wrapper script to send the *start* and *done* messages and can also be used within the user's script to send informational messages that appear in the log output stream of the *vrun* process. The following command-line options and usage are included in the output of the *-help* option of the *vrun* application:

```
vrun -send <event> [<options>] <action> [<action> ...]
  Send pre-defined event (start, user, or done) to a listening vrun process
    -code <integer>      Report status code (0 => OK)
    -dest <porthost>     Monitor daemon socket (port@host), required
    -message <string>    Report message string
```

The value of the *event* parameter (which must be one of the predefined choices listed above: start, user, or done) is used as the message keyword in the serialized message. The *port@host* of a listening *vrun* process must be supplied. The *-status* and *-message* options are optional.

In addition, there are two modes that can be used to monitor and/or control a running *vrun* process.

The *vrun-kill* command is as follows:

```
vrun -kill
  Terminate ongoing regression run
    -vrldata <directory>  Specify root of VRMDATA directory tree
```

This can be used to terminate a running *vrun* regression run. The necessary *port@host* is read from the most recent event log so the target *vrun* process of this command is the process currently running in the VRMDATA directory defined by the *-vrldata* option (or in \$cwd/VRMDATA by default). If the regression run was started using the default VRMDATA path, running the *vrun -kill* command in the same directory from which the original regression run was started should be enough to terminate the regression run.

The *vrun-status -tail* command is as follows:

```
vrun -status -tail
  Follow the progress of another vrun process
    -vrldata <directory>  Specify root of VRMDATA directory tree
```

This carries out the algorithm described in the section on “[Attach to a Running vrun Process via Event Log](#)” on page 473. That is, the most recent event log is read (based on the current VRMDATA directory) and the *port@host* field of the *begin* message is used to connect to another running *vrun* process. The status report produced is similar to that described in “[Status Reports](#)” on page 68, save that the selection of columns is fixed and the status messages are reported in the order they occur.



# Chapter 13

## Controlling Grid Jobs

---

The architecture of VRM is intentionally general. By default, Action scripts (*preScript*, *execScript*, *postScript*, and others) are launched as child processes on the local machine. The RMDB file can specify execution methods for all or a select group of Actions that cause the Action scripts to be launched in a user-specified manner. This can be used to launch remote processes on specific servers, to wrap a given Action script in another script, or to submit the Action script for execution on a compute grid or server farm. The user only needs to provide a template for the command necessary to launch a given script and *vrun* will follow the instructions set by the user when launching the scripts associated with that execution method.

However, the *vrun* application also provides for a rudimentary post-launch control over the resulting grid jobs in cases where such post-launch control is provided by the compute grid management tools. This is also intentionally general, as VRM should be able to integrate with regression flows utilizing various grid management systems, including those developed in-house by the user. The post-launch control operations understood by the *vrun* application are shown in [Table 13-1](#).

**Table 13-1. Post-launch Control Operations**

Operation	Per-job	Description
hold	No (global)	Suspend launching of new jobs without affecting currently running jobs.
suspend	Yes	Suspend execution of one or more running jobs (on platforms that support job suspension).
resume	Yes	Resume execution of one or more suspended jobs (on platforms that support job suspension).
kill	Yes	Terminate execution of one or more jobs, and possibly the <i>vrun</i> process itself.

The *resume* operation can be used to resume individual jobs or to countermand the effect of the *hold* operation. An individual job can only be the target of a *resume* operation if it was placed in suspended mode via a *suspend* operation sent to the same *vrun* process being asked to *resume* the job.

In order for these operations to function effectively on a given grid system, it is necessary to supply *vrun* with information about the grid system commands necessary to carry out these operations. For the grid systems, Platform Computing Corporation Load Sharing Facility (LSF), Univa Grid Engine (UGE), Oracle Grid Engine, previously known as Sun Grid Engine (SGE), and Realtime Design Automation's NetworkComputer (RTDA) this information is built-in to

the *vrun* application as supplied (although it can be overridden, if necessary). For other grid systems, this information is supplied by the user as described in the sections to follow.

Refer to “[Integration with Compute Grid Systems](#)” on page 94 and “[Aggregating Multiple Actions in a Single Job](#)” on page 336 for additional information.

<b>Specifying the Type of Grid System</b> .....	<b>478</b>
<b>Specifying the Job Control Commands</b> .....	<b>480</b>
<b>Controlling Jobs Remotely</b> .....	<b>484</b>
<b>Running with JobSpy</b> .....	<b>487</b>

## Specifying the Type of Grid System

In order for *vrun* to know which Action scripts are associated with which grid management system, it is necessary for a grid type string to be specified in the RMDB file. Since a job queued to a compute grid must make use of an execution method, the *method* element associated with said execution method is the best place to specify the grid type. A *gridtype* attribute on the *method* element can be set to a non-empty string to specify the set of grid management commands to be used when controlling the resulting grid job. Except for the two built-in grid types (*lsf*, *uge*, *sge*, and *rtda*), this string can be any arbitrary (legal) identifier. The string is used later as the basis for calling one or more user-definable procedures that control the jobs after launch.

An example of a simple one-test regression set-up for use with the LSF grid management system is shown [Example 13-1](#).

**Figure 13-1. LSF Grid Management System Regression Sample**

```
<rmdb>
  <runnable name="nightly" type="group">
    <method gridtype="lsf">
      <command>bsub -J (%RUNNABLE%) -oo (%TASKDIR%)/(%SCRIPT%).bat.o%J
          -eo (%TASKDIR%)/(%SCRIPT%).bat.e%J (%WRAPPER%)</command>
    </method>
    <members>
      <member>test1</member>
    </members>
    <execScript>
      <command>echo {Hello World from (%INSTANCE%)}</command>
    </execScript>
  </runnable>
  <runnable name="test1" type="task"/>
</rmdb>
```

In this example, the unconditional (and anonymous) *method* command specified in the “nightly” Runnable is selected to supply the execution method command for the single Action in the regression suite. Since the value of the *gridtype* attribute in the *method* element associated with Action’s execution method is set to *lsf*, and since the string *lsf* is understood to indicate the

Load Sharing Facility (LSF) grid management system, the utilities and commands supplied with the LSF system is used to control this job once it has been launched. The launch itself uses the command specified in the *command* element specified under the selected *method* element.

## Detecting Failed Grid Jobs

For the following instances, the actions will be re-launched if you enabled local rerun.

- If an execution method, containing a valid gridtype attribute, fails with a non-zero exit code, the method will be reported as a launch failure.
- If a job launched by vrun is reported as having failed or is not included in the status report at all, it will be flagged as a queuing failure.

Refer to the section “[Local \(Immediate\) Rerun](#)” for more information.

## Specifying the Job Control Commands

The grid type values *lsf*, *uge*, *sge*, and *rtda* are predefined and built-in to the *vrun* application. This section assumes some other grid management system is in use. In the following example, the grid type in the listed *method* element is set to the arbitrary string *fred*:

```
<method gridtype="fred">
    <command>fredsub (%WRAPPER%) </command>
</method>
```

To launch an Action under this execution method, the *fredsub* command is used along with the name of the script containing the command to launch the Action's wrapper script. In order to suspend, resume, or kill the resulting job once launched, the user needs to supply user-definable procedures for each operation. The names of these procedures are based on the string given in the *gridtype* attribute of the *method* element associated with the execution method originally used to launch the job.

The grid type string is converted to “title case” (that is, the first letter is converted to upper-case and the remainder of the string is converted to lower-case) and combined with the name of the operation to be performed. The user-definable procedures can be defined as shown in the section “[Grid Management Procedures](#)”

In all cases, the standard data value hash (see “[Modifying VRM Behavior with User-definable Procedures](#)” on page 413) is passed containing the context path of the Action to which the operation applies. Each of these procedures start from an Action context path and applies only to the job(s) directly associated with that Action. If multiple jobs can be associated with a single Action, then the *<gridtype>GetJobId* procedure should return a list of job IDs and the other procedures should be prepared to iterate over that list.

In the example above, the following procedures can be provided to control jobs submitted by the *fredsub* command:

- FredGetJobId
- FredGetStderr
- FredKillJob
- FredStopJob
- FredResumeJob

If any of these procedures is not defined, requests for that operation will yield a warning and the request is ignored. For example, if *FredStopJob* and *FredResumeJob* are not defined, then the commands *vrun -suspend* and/or *vrun -resume* will result in a warning for any jobs associated with Action scripts launched via an execution method whose grid type is set to *fred*. If a given procedure is defined but, when called, throws an error, that error is caught and reported by the *vrun* process as a warning.

Because of the way the value of the *gridtype* attribute is used to map to the user-definable procedures, the user is advised not to be too creative with case combinations of the *gridtype* values. In other words, be advised that the *gridtype* strings *uge*, *UGE*, *Uge*, *uGe*, and *ugE* all point to the same set of user-definable procedures.

It is possible that one or more jobs might not respond to a given control operation for some reason, or that one or more jobs may have terminated and been removed from the grid's list of pending jobs by the time *vrun* issues the command to carry out the requested operation. The *vrun* process does not consider this an error. Valid “done” messages from Action scripts continue to be processed regardless of the state of the job. The suspend, resume, and kill operations are considered advisory and are provided only as a convenience to VRM users.

<b>Pro-active Grid Status Checks .....</b>	<b>481</b>
<b>Locally Launched Jobs .....</b>	<b>482</b>
<b>Predefined User-definable Procedures .....</b>	<b>483</b>

## Pro-active Grid Status Checks

A pro-active grid status check, enabled by default, verifies that all Actions submitted to a grid management system, which have not yet completed, are still listed by the grid management system as either pending or running. This allows VRM to detect anomalous behavior in the set of submitted jobs without waiting for a timeout.

You can disable pro-active checks by specifying the *-nogridcheck* option on the *vrun* command line.

A pro-active check is only done when VRM is not otherwise engaged in launching new Actions or processing messages from previously launched Actions. The frequency of the checks is limited to once every 300 seconds (5 minutes). In addition, a blackout period of 60 seconds is imposed after any Action is launched. The blackout period prevents VRM from checking the status of a grid job before the grid management system has entered a newly-launched job into its pending queue.

At the start of each pro-active check, a list of currently pending or running Actions is compiled. Only those Actions which were launched via an execution method element that carries a non-blank *gridtype* attribute are considered. A list of *gridtype* values is also determined at this time. For performance reasons, VRM may not check all the pending (or running) Actions at one time. However, each time it gathers a list of Actions to be checked, it will check every Action on the list before updating the list. This ensures that every Action launched from every named queue eventually gets checked, assuming VRM is not too busy with higher priority activities.

For each *gridtype* found during the scanning of eligible Actions, the appropriate grid status command is executed (for example, **bjobs** for an LSF grid or **qstat** for an SGE/UGE grid). The pro-active check can handle multiple grid management systems in a single run. If the grid status command for a given grid management system returns an error, Actions submitted to that grid

management system will not be checked until the next pro-active grid status check. If a given grid status command fails 5 times in succession, a warning is issued to the *vrun* log output. Actions will not be flagged as missing by the pro-active check unless the grid status command for the *gridtype* associated with that Action has returned a clean queue status report.

The grid status command and interpretation are defined in the section “[Grid Management Procedures](#)”.

Once the current grid status is obtained from the appropriate grid status command(s), each Action is checked against the current grid status. If the grid status lists an Action as having failed, VRM will report the error as a launch error (failed/launch) and the error message will indicate “grid error” as the reason. If the grid status does not list an Action at all, VRM assumes the Action was dropped by the grid and will report the Action as having timed out (either timeout/queued or timeout/executing, depending on the most recent state of the Action). In either case, the default re-run mechanisms will be triggered in the same way as for any other error.

## Locally Launched Jobs

Actions for which no execution method is defined, or for which the selected execution method does not specify a grid type, are assumed (for job-control purposes) to have been locally launched. If the *vrun* process is able to determine the process identifier (pid) for the resulting child process, any job-control operations is applied to those jobs via the POSIX-standard *kill* command. The following signals are sent to the child process via the *kill* command in order to control the process:

**Table 13-2. Signals Sent Via the Kill Command**

Signals Sent	Action
STOP	Suspend the process.
CONT	Resume the suspended process.
KILL	Terminate the process.

In the case of an execution method that posts the Action script to a remote machine for execution (for example, via the *rsh* command), the process identifier received by the launching *vrun* process is that of the *rsh* command and not that of the launched Action script. In cases such as this, it cannot be possible to control the actual job associated with this Action. If this is a problem, Mentor Graphics suggest creating a set of user-definable procedures that can capture the remote hostname and remote pid and pointing the execution method to this set of procedures via the *gridtype* attribute.

## Predefined User-definable Procedures

There are several user-definable procedures are already defined by default in the *vrun* application.

You can see a list of them in the section “[Grid Management Procedures](#)”.

Like all user-definable procedures, these can be overridden by user-supplied TCL fragments (see “[Modifying VRM Behavior with User-definable Procedures](#)” on page 413). In both cases, the *KillJob*, *StopJob*, and *ResumeJob* procedures all call the associated *GetJobId* procedure first, which scans the standard output file for a given message pattern in order to convert the Action context path into a grid system job ID, and the resulting job ID is passed to the appropriate utility from the grid management software. It is assumed that the grid management utilities are available for execution from the launching *vrun* process (likely the case, unless the launch itself is not worked properly). User-defined grid systems can pattern their user-definable procedures after these built-in sets.

## Controlling Jobs Remotely

Once the *vrun* application starts launching Action scripts, the only way to communicate with the *vrun* process is via the TCP/IP socket opened for communication with the wrapper scripts. The *port@host* on which any given *vrun* process is listening is published in the log output of that process (usually as the first message, except in *-debug* mode). Any process can connect to this socket and send one of the following messages to the running *vrun* process:

**Table 13-3. vrun Messages**

Message	Action
suspend [<action> [...]]	Suspend the jobs associated with one or more Actions.
resume [<action> [...]]	Resume the suspended jobs associated with one or more Actions.
kill [<action> [...]]	Terminate the jobs associated with one or more Actions.
stop	Stop launching jobs (no effect on already running jobs).
exit	Stop launching jobs and exit (no effect on already running jobs, except that their done message may fail to connect).

The *suspend*, *resume*, and *kill* messages can optionally include one or more Action context paths.

- If any Action context paths are specified, the operation implied in the message applies only to those jobs associated with the named Actions (not to any other jobs or to the *vrun* process).
- If no Action context paths are specified, the operation implied in the message is assumed to apply globally (that is, it applies to all currently running jobs and to the *vrun* process itself).

For example, if a bare *kill* message is received by a running *vrun* process, it sends the necessary job control command/signal to kill every currently running job and then the *vrun* process itself will exit. If a bare *suspend* message is received by a running *vrun* process, it sends the necessary job control command/signal to suspend every currently running job and will then cease to launch any new Action scripts until a *resume* message is received or the *vrun* process is aborted.

If a *vrun* process has stopped launching new Action scripts as the result of a previous *suspend* or *stop* message, only a bare *resume* message or a *start* message causes the *vrun* process to resume launching jobs. That is, stopping the *vrun* process and all running jobs via a bare *suspend* message and then subsequently resuming only a select set of those jobs via a *resume* message will still leave the *vrun* launch algorithm in suspended mode because the selective *resume*

message does not apply to the *vrun* process itself. Likewise, while a *start* message causes a stopped *vrun* process to resume its effort to launch new Action scripts, it will not cause the execution of jobs suspended by a previous *suspend* message to be resumed. A bare *resume* command will resume both the *vrun* process itself and any suspended jobs.

The optional Action context paths passed as part of the *suspend*, *resume*, and *kill* messages are considered regular expressions that are partially matched against the list of pending Actions according to the following rules:

1. If a context path starts with a slash (/), the path only matches at the start of a pending Action. That is, while *nightly* matches any Action under the *nightly* group, regardless of whether the *nightly* group is named as a top-level Runnable, the context path /*nightly* only matches Actions under the top-level Runnable called *nightly*.
2. If the final component of a context path is not a script name (that is, \**Script*), a /\**Script* component is added to the context path.

For example, if the following Actions are currently running:

```
nightly/test1/execScript
nightly/test2/execScript
nightly/test3/execScript
```

The message *suspend nightly* suspends all three Actions while the message *suspend nightly/test1*, or simply *suspend test1* suspends only the first Action and leaves the other two running. Since a suspend message has no effect on Actions that are already suspended, a subsequent *suspend nightly* would then suspend the other two Actions, as would *suspend test[23]* (using a regular expression).

<b>Command Line Control from a Second vrun Process .....</b>	<b>485</b>
<b>Adjust Timeout Values to Account for Time Suspended.....</b>	<b>487</b>

## Command Line Control from a Second vrun Process

The *vrun* application has a mode via which it simply passes a message to another *vrun* command and then quits. This is a convenience feature designed to make it easier to control a running *vrun* process once launched.

This mode is activated by any of the command-line options shown in [Table 13-4](#).

**Table 13-4. Regression Run Control Command Options**

Options	Action
-suspend [<action> [...]]	Suspend the jobs associated with one or more Actions and, possibly, the vrun Action-launch process.

**Table 13-4. Regression Run Control Command Options (cont.)**

Options	Action
-resume [<action> [...]]	Resume the suspended jobs associated with one or more Actions and, possibly, the <i>vrun</i> Action launch process.
-kill [<action> [...]]	Terminate the jobs associated with one or more Actions and, possibly, the <i>vrun</i> process itself.
-stop	Stop launching jobs (no effect on already running jobs).
-start	Resume launching jobs (if stopped).
-exit	Stop launching jobs and exit (no effect on already running jobs, except that their done message may fail to connect).

The content and meaning of the Action context arguments is the same as in [Table 13-3](#) on page 484. It is an error to specify more than one of these mode options in the same *vrun* command.

These command-line options are pretty much an exact match to the TCP/IP messages accepted by the actively running *vrun* process. When the *vrun* application detects one of these options on the command line, it formulates and sends a corresponding TCP/IP message to the another *vrun* process (which is in the process of launching and monitoring jobs).

The *port@host* of the target *vrun* process is determined in one of two ways. If the *-dest <port@host>* command-line option is specified on the same command line, the message is sent to the specified port on the specified host. Otherwise, the most recent Status Event Log from the VRMDATA directory is opened and, if a begin event record is found, then the *port@host* from that event record is used. The directory can be specified via the *-vrldata* command-line option or *vrun* will look for the log under a directory called VRMDATA in the current directory.

The most efficient use of these options occurs when the default VRMDATA directory is being used and there is another shell window open in the same directory as that from which the target *vrun* command was originally launched. In that case, something as simple as the following command:

#### **vrun -kill**

can be used to completely terminate a *vrun* process and all the currently running jobs that were launched from that process. Since *vrun* uses asynchronous TCP/IP communications, the user can also temporarily suspend the running *vrun* process, execute the *vrun -kill* command, and then resume the original *vrun* process (at which time it should pick up the *kill* message and abort cleanly).

## Adjust Timeout Values to Account for Time Suspended

When an Action script is suspended, any non-zero timeout is temporarily suspended. For example, if an Action script launched with an execution timeout of 90 seconds is suspended after 30 seconds of runtime, in theory that Action still has 60 seconds of runtime remaining. If the suspension lasts longer than that, the Action will not timeout. Instead, when the Action script resumes execution, its timeout is adjusted such that it will not timeout until 60 seconds after the time it resumes execution. In other words, the Action script is always allowed to execute for the time specified in the execution timeout (*timeoutQ*) without penalty.

If a *start* message arrives for an Action that is in suspended mode, its timeout is set to the execution timeout value (*timeoutX*), but the Action will continue to be marked as suspended pending a command to resume the Action. A suspend command has no effect on the timeout of an Action that is already marked as suspended. Likewise, a resume command has no effect on the timeout of an Action that is not marked as suspended.

### Unsupported Environments

VRM does not support suspend, resume, or kill operations on locally executed Actions under the Windows environment. If grid management is being used to queue Actions onto a server grid and the appropriate grid management utilities are available to be called from within the *vrun* process on the Windows host, the suspend, resume, and kill operations should work as described above.

The suspend, resume, and kill operations do not actually affect running exec-mode Action scripts (those with the *launch* attribute of the Action script element set to anything other than *vsim*) that have been launched on the local machine. The reason is that the wrapper in an exec-mode script launches the user's script under a shell using the TCL *exec* command and the resulting shell process runs independently of the *vish*-based wrapper.

A request to suspend, resume, or kill the job associated with an exec-mode Action will result in the suspension, resumption, or termination of the *vish* process that is executing the wrapper, but the signal will not propagate to the child process (this is by design in POSIX-compliant systems). If the Action is launched via a grid management system, as described above, there should be no problem controlling the job after it has been launched.

## Running with JobSpy

There is nothing special required to run VRM with JobSpy. Simply set the *JOBSPY\_DAEMON* environment variable before invoking VRM. Make sure that any exec-mode scripts preserve the environment from the invoking process (for example, by using the *-f* option with */bin/csh*). Execute *vrun* specifying some set of Tasks that represent Questa simulations and as the simulations run, they should show up in the JobSpy job list.



# Chapter 14

## Environment Variables and Built-in Parameters

Reference information about environment variables and built-in parameters for VRM.

<b>Environment Variables .....</b>	<b>489</b>
<b>Predefined Parameters .....</b>	<b>492</b>
<b>Parameters Recognized/Defined by VRM.....</b>	<b>493</b>
<b>Parameterized Element Attributes.....</b>	<b>495</b>
<b>Relative File Rules .....</b>	<b>496</b>

## Environment Variables

The following environment variables are recognized and used by VRM:

**Table 14-1. VRM Environment Variables**

Variable	Value	Description
MTI_VRUN_BACKTRACE	(none)	Causes TCL stack backtrace to be printed to log output for VRM errors.
MTI_VRUN_CLUB_BY_TARGET	(none)	Causes vrun to use only the mergefile/triagefile path to club UCDB files for auto-merge/triage (see “UCDB File “clubbing” in Automated Merge and Automated Triage” on page 400).
MTI_VRUN_DB	path	Pointer to VRM RMDB database.
MTI_VRUN_GVPATH	(see below)	Specified where to search for Graphviz libraries.
MTI_VRUN_MIN_TIMEOUT	integer	Increases the global minimum timeout (only if value > current default)
MTI_VRUN_SEMCHECK	(none)	Enables semantic checking of loaded rmdb.

**Table 14-1. VRM Environment Variables (cont.)**

Variable	Value	Description
MTI_VRUN_SEND	(see below)	Command used by vish-mode Action scripts to send user-defined messages.
MTI_VRUN_USE_VCOVER	(none)	Causes vrun to call vcover stat -tcl -attronly to fetch UCDB test data records instead of using the UCDB API.
MTI_VRUN_USERTCL_PATH		Causes vrun to honor this environment variable that specifies a search path for files supplied to the -loadtcl command-line options or the file attribute of a usertcl element in RMDB.

If the *MTI\_VRUN\_GVPATH* is not specified, then the GUI searches the directory where the *vrun* executable is read and the */usr/lib64* and */usr/lib* directories for a directory *graphviz/tcl* that contains loadable Graphviz libraries. This search is done on an as-needed basis so the user only needs to set this variable when using the GUI's built-in Graphviz window to view the RMDB database contents.

The *MTI\_VRUN\_SEND* environment variable is set by the *vish*-mode wrapper (used for Action scripts whose launch attribute is set to *exec* or to some command shell path). It contains the basic command (minus the *-message* option) needed to send a user-defined message to the launching *vrun* process. See the section on “[Sending User-defined Messages from an Action Script](#)” on page 310 for details.

## Environment Issues

Unless the Action scripts specifies otherwise, the environment under which the Action script commands execute is a combination of settings inherited from the environment of the shell from which *vrun* is launched and whatever *vrun* (and the *vish* shell that executes it) happened to set. In particular, the *MODEL\_TECH* environment variable is set to point to the Questa release from which *vrun* was executed. If the intention is that the Action script commands are to be taken from the same Questa release as *vrun*, this is a perfect situation. However, if the intention is to use the *vrun* from one Questa release to execute commands from another Questa release (or from some third-party verification platform), care must be taken that the environment is properly set-up.

Note that using *vrun* from one Questa release to launch commands from another Questa release requires that *exec* type scripts be used exclusively. The remainder of this section assumes all Action scripts have their launch attribute set to *exec*.

The most robust solution is to use a “shebang” line like the following (on each script, without the *-f* option):

```
#!/bin/csh
```

This causes the shell running each script to load the user's *.cshrc* file before executing any commands. In this file, the environment variables needed by the script can be set properly (especially *PATH*, that governs which Questa release will supply the command executables). The penalty is a slight decrease in performance due to the *.cshrc* file. Other shells can be used with the appropriate setup file for that shell.

Note that the setup programs usually **define** environment variables but, in this case, the setup script (that is, *.cshrc*) needs to also **undefine** one variable. The *MODEL\_TECH\_TCL* environment variable is sometimes set by the *vrun* process and may need to be un-set in order for the Action script commands to run properly.

Another solution is to hard-code the Questa (or 3rd-party tool) release path onto the Action script commands. [Example 14-1](#) launches a simulation using a specific version of Questa, no matter which Questa release is used to supply the *vrun* executable itself.

### Figure 14-1. Hard-code Questa or 3rd Party Tool Release Path in Script

```
<rmdb>
  <runnable name="nightly" type="group">
    <parameters>
      <parameter name="platform"
        type="tcl">$::env(PLATFORM) </parameter>
      <parameter name="MTI">/path/to/release/modeltech</parameter>
      <parameter name="Q">(%MTI%)/(%platform%)</parameter>
    </parameters>
    <members>
      <member>test1</member>
    </members>
  </runnable>
  <runnable name="test1" type="task">
    <execScript launch="exec">
      <command>#!/bin/csh -f</command>
      <command>rm -rf work</command>
      <command>(%Q%)/vlib work</command>
      <command>(%Q%)/vlog (%RMDBDIR%)/top.sv</command>
      <command>(%Q%)/vsim top -c -do "run -all; quit -f"</command>
    </execScript>
  </runnable>
</rmdb>
```

In this case, it does not matter whether *vrun* was executed from */path/to/release/modeltech* or some other location. The explicit paths on the Action script commands will override any setting provided by the VRM process itself.

Another solution is to set the *PATH* to point to the location of the executables to be used for the Action script commands and then invoke *vrun* by its absolute path. This should work, since

**Predefined Parameters**

neither the *vrun* application nor the underlying *vish* interpreter change the *PATH* environment variable. This approach is more risky, but does not affect performance.

Note that all this is moot if the Actions are launched on a server grid and the grid management software does not preserve the environment when launching jobs.

## Predefined Parameters

The following parameters are built-in to VRM and are always defined within their respective context. These parameters can be overridden via the *-G* command-line option. However, due to the nature of the data contained in these parameters, they should only be overridden by someone who understands the tool very well.

Refer to the section “[Predefined Parameter Usage](#)” for additional information.

**Note**

 VRM predefined parameters are case-sensitive and always upper-case.

**Table 14-2. Predefined Parameters**

Parameter	Context	Meaning
CONTEXT	Runnable	List of ancestor Runnables showing the complete path.
DATADIR	Global	Root directory of VRM Data tree.
INSTANCE	Runnable	Name of the Task or Group being executed. (This is the same as RUNNABLE for non-repeating Tasks/Groups or RUNNABLE~ITERATION for repeating Tasks/Groups.)
ITERATION	Runnable	Repeat index (1..N) or foreach token of the Task being executed.
JOBSIZE	Method	Number of actions clubbed for the current job.
MODEL_TECH	Global	Directory in which the <i>vrun</i> (and other Questa executables) resides.
RMDBDIR	Global	Directory from which the RMDB file was read.
RMDBFILE	Global	Name of the current active RMDB file.
RUNNABLE	Runnable	Name of the Runnable corresponding to the Task/Group.
SCRIPT	Action	Name of the Action script (preScript, execScript, or postScript).
TASKDIR	Runnable	Working directory in which the Task/Group is being executed.

**Table 14-2. Predefined Parameters (cont.)**

Parameter	Context	Meaning
VRUNDIR	Global	Directory from which vrun was invoked.
VSIMDIR	Global	Reflects the value controlled by the vrun -wrapperpath option
WRAPPER	Method	Name of the shell script containing the Action script. <sup>1</sup>
ucdbfiles	Group	List of UCDB files to be merged/triaged. Refer to the section “ <a href="#">Enabling and/or Disabling Coverage Merge Flows</a> ” for more information.
CFGFILE		Deprecated. Replace with RMDBFILE.
CONFDIR		Deprecated. Replace with RMDBDIR.

1. Since the primary purpose of the execution method functionality is to allow a job to be launched onto a grid system or a remote machine, this script name can be used in the grid launch command where the name of the script to be launched is normally placed.

## Parameters Recognized/Defined by VRM

The parameters in this section receive special treatment when defined in the RMDB database.

**Note**

 Special parameters, recognized by VRM, are case-sensitive and always lower-case.

---

**Table 14-3. Special Named Parameters**

Parameters	VRM Functionality	Default	Description
faillog	failure rerun		Path of the file in which a list of failed Actions should be written. Refer to the section “ <a href="#">Global (Parameter-based) Method</a> ” for more information.
mergefile	auto-merge		Path/name of the UCDB file into which the current Action's coverage will be merged
mergelist	queued auto-merge	(%TASKDIR%)/mergeScript.files	Path to file containing list of UCDBs to merge.
mergeoptions	auto-merge, testplan auto-import		String containing command-line options to vcover merge command

**Table 14-3. Special Named Parameters (cont.)**

Parameters	VRM Functionality	Default	Description
nodelete	auto-delete		List of the file globs to preserve from auto-deletion. Refer to the section “ <a href="#">Protecting Select Files from Deletion</a> ” for more information.
tplanfile	testplan auto-import		Name (and path) of the testplan XML file. Alternatively, you can specify the name (and path) of a UCDB file, which will be imported into the auto-merge functionality without calling the xml2ucdb command. Refer to the section “ <a href="#">Testplan Auto-import</a> ” for more information.
tplanoptions	testplan auto-import		Set of additional options to be used with the xml2ucdb command upon importing the testplan. Refer to the section “ <a href="#">Testplan Auto-import</a> ” for more information.
tplanucdb	testplan auto-import		Path/name to the output UCDB file. You can not alter this value and it is defined in the context of its own Action.
trendfile	auto-trend		Path/name of the target trend UCDB file. Refer to the section “ <a href="#">trendScript</a> ” for more information.
trendoptions	auto-trend		String containing command line options to the vcover merge -trend command.
triagefile	auto-triage		Path/name of the TDB file into which the current Action's messages will be imported

**Table 14-3. Special Named Parameters (cont.)**

Parameters	VRM Functionality	Default	Description
triagelist	queued auto-triage	(%TASKDIR%)/ triageScript.files	Path to file containing list of UCDBs to triage
triageoptions	auto-triage		String containing command-line options to triage dbfile command
ucdbfile	auto-merge/triage, testplan auto-import		Path/name of the UCDB file associated with the current Action

Predefined parameters (those defined by VRM) are discussed in “[Predefined Parameter Usage](#)” on page 266.

## Parameterized Element Attributes

Most element attributes in the XML database are used verbatim.

A select few are parameter expanded before use. [Table 14-4](#) lists which ones and why.

**Table 14-4. Parameterized Element Attributes**

Element	Attribute	TCL	Justification
localfile	src		Source file for localfile copy/link.
localfile	name		Target file for localfile copy/link.
method	if	eval	Enables conditional execution methods.
method	unless	eval	Enables conditional execution methods.
Runnable	foreach		Allows parameterized iteration.
Runnable	if	eval	Enables conditional membership.
Runnable	repeat		Allows parameterized iteration.
Runnable	unless	eval	Enables conditional membership.
usertcl	file		Allows use of predefined paths.
usertcl	if	eval	Enables conditional loading.
usertcl	unless	eval	Enables conditional loading.
preScript, execScript, postScript	file		Allows use of predefined paths.

**Table 14-4. Parameterized Element Attributes (cont.)**

Element	Attribute	TCL	Justification
preScript execScript, postScript	ucdbfile		Points to UCDB file for Action.
preScript execScript, postScript	launch		Points to an executable

## Relative File Rules

There are a number of parameters and element attributes whose values represent files and/or paths. While absolute paths can be specified in any of these places, in general these parameters and attributes are specified as relative file/path names. The following basic rules explains how VRM resolves these relative paths:

1. Paths supplied on the command line are always assumed to be relative to the directory from which *vrun* was launched.
2. Paths within the RMDB file that represent source files are always assumed to be relative to the directory from which the RMDB was loaded.
3. Paths within the RMDB file that represent collateral files are always assumed to be relative to the working directory for the Action.
4. Certain paths (*mergefile/mergelist* only) have semantics that require that they be treated as relative to a specific location.
5. All paths defined within the RMDB file (parameters and/or attributes) are parameter-expanded (but not TCL expanded).

**Table 14-5** lists the various places where file/path names can be specified and how VRM interprets the specified value. The Expanded column indicates whether the value will first be parameter-expanded before being interpreted.

**Table 14-5. File/Pathname Specification Options**

Source	Expanded	Relative To
command-line option: -rmdb	No	\$cwd
command-line option: -faillog	No	\$cwd
command-line option: -loadtcl	No	\$cwd
command-line option: -vrmdata	No	\$cwd
parameter (file type)	Yes	RMDBDIR

**Table 14-5. File/Pathname Specification Options (cont.)**

Source	Expanded	Relative To
parameter: faillog	Yes	DATADIR
parameter: mergefile	Yes	DATADIR
parameter: mergelist	Yes	DATADIR
parameter: ucdbfile	Yes	TASKDIR
attribute: localfiles:name	Yes	TASKDIR
attribute: localfiles:src	Yes	RMDBDIR
attribute: usertcl:file	Yes	RMDBDIR
attribute: xxxScript:file	Yes	RMDBDIR



# Chapter 15

## Catalog of User-Definable and Utility Procedures

---

This chapter documents each of the user-definable procedures, and the low-level utility procedures available to the user-definable procedures.

Refer to “[User-definable Procedures and Loading of Arbitrary TCL Code](#)” on page 412 for details on how these procedures fit into the overall VRM flow.

<b>Catalog Format .....</b>	<b>499</b>
<b>User Definable Procedures .....</b>	<b>501</b>
<b>Utility Procedure Catalog .....</b>	<b>537</b>

## Catalog Format

Each user-definable procedure listed in this reference includes the following sections:

- Input arguments — The key/value pairs passed to the procedure in the data array.
- Returns — What the procedure should return to *vrun*, if anything.
- Default behavior — What the built-in version of this procedure does.
- Side effects — Other actions for which the user-definable procedure can be responsible.

Input arguments are passed to the user-definable procedure in a TCL array (which is an associative array of key/value pairs). This is done so that future additions to the data items passed will not affect existing user-defined override procedures. The procedure receives one argument, which is the name of this data array, and uses a TCL *upvar* command to gain access to the array (from the calling procedure's scope).

The return value is, in most cases, a simple string. It is recommended that all override procedures return some value (possibly a 0 or an empty string), even if the procedure is not listed as returning a value. In the case of an error, it is permissible to use the TCL *error* command to throw an error. The calling *vrun* process will catch the error and report it in the *vrun* log output (throwing an error should not cause the regression run itself to fail unless some required information is unavailable or corrupted).

In the case of the post-execution analysis (see “[Post-execution Analysis](#)” on page 361) chain of procedures (that is, *AnalyzePassFail*, *ActionCompleted*, *OkToMerge*, *MergeOneUcdb*, *OkToTriage*, *TriageOneUcdb*, and *OkToDelete* in that order), a single data array is created for the Action and shared among these procedures. That means that a user-definable procedure can

actually post additional data elements to the array and subsequent user-definable procedures in the chain can pick that data up from the array. For example, *AnalyzePassFail* can create an array element called “data(numerrors)” and *OkToMerge* can access this value as “\$data(numerrors)” and use it to determine whether or not to merge the UCDB in question.

## User Definable Procedures

---

The following groups of procedures are user definable and available for your use:

<b>Common Arguments Passed to All User Definable Procedures .....</b>	<b>501</b>
<b>Global Status Change Procedures .....</b>	<b>504</b>
<b>Action Status Change Procedures .....</b>	<b>506</b>
<b>Post-execution Processing Procedures.....</b>	<b>509</b>
<b>Regression Control Procedures .....</b>	<b>520</b>
<b>Grid Management Procedures .....</b>	<b>525</b>
<b>Logging Procedures.....</b>	<b>534</b>

## Common Arguments Passed to All User Definable Procedures

There are a number of common key/value pairs passed to every user-definable procedure in the data array. The values represent file names and/or paths that are global to the entire regression run.

Some of the values may be blank in some cases. For example, if an Action did not actually produce a UCDB file, then the *ucdbfile* keyword points to an empty string, as will the *ucdbstat* keyword.

These key/value pairs are shown in [Table 15-2](#). These key/value pairs can be found in the data array passed to each procedure, in addition to the values unique to each procedure.

The arguments in [Table 15-1](#) are passed to all the user-definable procedures, whether or not they have anything to do with a specific Action.

Note that the data argument keywords are case sensitive.

**Table 15-1. Common Arguments Passed to All User-definable Procedures**

Argument	Description
RMDBFILE	Path to the active RMDB file.
RMDBDIR	Directory from which the RMDB file was loaded.
DATADIR	Path to the root of the working (VRM Data) directory tree.
VRUNDIR	Directory from which vrun was executed (that is, \$cwd).

[Table 15-2](#) lists the common arguments passed to the action-related user-definable procedures.

**Table 15-2. Common Arguments Passed to Action- and Runnable-related Procedures**

Argument	Description
CONTEXT	Complete context chain of the active Runnables. (Runnable-related)
INSTANCE	Current Runnable. (Runnable-related)
ITERATION	Foreach token of Runnable template. (Runnable-related)
RUNNABLE	Name of Runnable element. (Runnable-related)
TASKDIR	Path to the working directory for the Action (note that the actual directory may not yet exist). (Runnable-related)
ACTION	Returns a unique string (the context chain) of the Action eligible to run, which you can use as the testname value. (Action-related)
SCRIPT	Name of script, such as preScript or any other script name. (Action-related)
TESTNAME	Name associated with Runnable, as defined by the optional <b>testname</b> parameter. (Action-related)

Table 15-3 lists the input arguments that are passed conditionally.

**Table 15-3. Input Arguments Passed Conditionally**

Argument	Description
ATTEMPT	Passed only if the Runnable is dynamically iterating.
HISTORY	Passed only if the Action is being rerun under the immediate rerun functionality.
\$Runnable.ITERATION	Passed only if the ITERATION of that Runnable in the context chain is non-blank.

Table 15-4 lists the data elements passed to the post-execution user-definable procedures (note that these values are dynamic and cannot appear as built-in parameters).

**Table 15-4. Data Elements Passed to the Post-execution Procedures**

Argument	Description
coverage	Total Coverage.
faillog	Path to the cumulative list of failed tests.

**Table 15-4. Data Elements Passed to the Post-execution Procedures (cont.)**

<b>Argument</b>	<b>Description</b>
passfail	Pass/fail flag. Any of the following: <b>passed</b> , <b>failed</b> , or <b>notrun</b> .  The value <b>notrun</b> is not valid subsequent to AnalyzePassFail.
randseed	Value of the SEED attribute in the UCDB test data record.
reason	Contains extended status values for the Action on whose behalf the procedures has been called,  This value is empty, except for timeouts, when <a href="#">AnalyzePassFail</a> is called, but contains the return value of AnalyzePassFail when subsequent post-execution analysis procedures are called.
status	Return status from the Action script.
tplancov	Testplan coverage.
ucdbfile	Path to UCDB file for the Action.
ucdbstat	Value of the TESTSTATUS attribute in the UCDB test data record.
message	Optional status message from the Action script.

## Global Status Change Procedures

The following procedures are called by *vrun* when the global status of the regression run changes:

<b>RegressionStarting</b> .....	<b>504</b>
<b>RegressionCompleted</b> .....	<b>504</b>
<b>ProcessIdleTasks</b> .....	<b>505</b>

### RegressionStarting

This procedure is executed after the execution graph and other VRM internal data structures have been initialized, but before the first round of Actions are initiated. This is a convenient place to put any global initialization code required for the other user-definable procedures to execute (it should not be confused with the *preScript* Action run as part of the top-level Group).

This procedure is called after any user-defined TCL code (defined in a *usertcl* element of the selected execution method or in a file loaded with the *-load* option) has been executed. Therefore, any variables or procedures defined in one of those code fragments should be visible to this user-definable procedure.

#### Input Arguments

See [Table 15-1](#) on page 501.

#### Returns

Nothing

#### Default Behavior

Nothing

#### Side Effects

Messages defined by this procedures are emitted, by default, when the regression run starts. You can suppress the messages with the *vrun -quiet* option.

### RegressionCompleted

This procedure is executed after all Actions resulting from the selected set of Runnables have either been executed or abandoned. It is a convenient place to emit status information to the log file, to populate GUI widgets when VRM is being executed from inside the *vsim* GUI, to notify third-party applications of the completion of the regression suite, or to kick off post-regression analysis scripts.

## Input Arguments

See [Table 15-1](#) on page 501.

## Returns

Nothing

## Default Behavior

Nothing

## Side Effects

Messages defined by this procedures are emitted, by default, when the regression run completes. You can suppress the messages with the `vrun -quiet` option.

# ProcessIdleTasks

The `vrun` process maintains an idle loop that runs either once a second or any time a status message is received from a running Action script. The idle loop is mostly responsible for launching eligible Actions and for detecting timeouts on running Actions. This procedure is called once for each pass through the idle loop. It was created as part of the auto-merge experimentation but is no longer used for anything. It can be used for any update process that must run continually throughout the regression run but cannot easily be tied to a positive event. Care should be taken when overriding this procedure as any lengthy processing can have a significant impact on VRM performance.

## Input Arguments

See [Table 15-1](#) on page 501.

## Returns

Nothing

## Default Behavior

Nothing

## Side Effects

None

## Action Status Change Procedures

The following procedures are called by *vrun* whenever the status of an Action changes:

ActionEligible.....	506
ActionLaunched.....	506
ActionStarted.....	507
ActionCompleted .....	507

### ActionEligible

This procedure is executed when the prerequisites of an Action are completed and the Action has been queued for execution. Note that this event does not imply that the Action has actually been launched (or queued to a grid), nor does it indicate that the VRMDATA directory for the Action has been created or prepared.

#### Input Arguments

See [Table 15-1](#) on page 501 to [Table 15-4](#) on page 502.

#### Returns

Nothing

#### Default Behavior

Message sent to the log output in *-verbose* mode.

#### Side Effects

None

### ActionLaunched

This procedure is executed when the Action has been launched or queued for execution on a grid system. The working directory for the Action will have been created and prepared by this point.

#### Input Arguments

See [Table 15-1](#) on page 501 to [Table 15-4](#) on page 502.

#### Returns

Nothing

## Default behavior

Message sent to the log output in *-verbose* mode.

## Side effects

None

# ActionStarted

This procedure is executed when the Action has begun execution. Since this event comes from the wrapper script, it indicates that the Action has successfully started and is actually running.

## Input Arguments

See [Table 15-1](#) on page 501 to [Table 15-4](#) on page 502.

## Returns

Nothing

## Default Behavior

Message sent to the log output.

## Side Effects

None

# ActionCompleted

This procedure is executed after each Action completes. Its purpose is to provide a post-completion hook to notify the Questa GUI (or 3rd-party utilities) of the completion of an Action. This procedure is also responsible for handling any collateral files (that is, log files, results files, and so on) that the user may wish to save outside of the VRM working directory tree.

The term “completed” implies any state change that results from *vrun* having deleted the action from its execution graph (including timeouts and skipped actions). In theory, there should be only one *ActionCompleted* event per Action. However, since an Action script can continue to run even after it times out or is otherwise disposed of by the *vrun* algorithm, it is possible for a second *ActionCompleted* event to be called for a given Action even after the Action has been declared complete.

It is also possible that both an *ActionStarted* and an *ActionCompleted* event can arrive after an Action has been declared complete. For example, the decision to dispose of a given Action can be made just after the Action script is launched, but before it actually began running. This can

occur more often when the Action is queued onto a heavily loaded compute grid. VRM makes no attempt to filter these duplicate events and the user should be careful when attempting to infer anything useful from them.

## Input Arguments

See [Table 15-1](#) on page 501 to [Table 15-4](#) on page 502.

The *ActionCompleted* user-definable procedure is unique in that it can be called for reasons other than the successful completion of the Action. In the case of a not complete event, the *passfail* data element will contain one of the keywords shown in [Table 15-5](#).

**Table 15-5. passfail Keywords**

passfail	Reason
dropped	Action not run (either StopRunning returned a true or due to a bug in vrun).
killed	Action was killed by user intervention.
skipped	Action was skipped due to earlier errors in dependent Actions.
timeout / queue	Action exceeded launch-to-start timeout.
timeout / exec	Action exceeded start-to-done timeout.
timeout	Action exceeded timeout.

## Returns

Nothing

## Default Behavior

Message sent to the log output (includes the path to the UCDB file in *-verbose* mode).

## Side Effects

This procedure is expected to copy and/or move files around as required by the user's regression environment. If the UCDB file is moved (to an archive directory, for example), then the new absolute path should be stored in the *ucdbfile* element of the data array. If any files are to be preserved after the auto-deletion step (assuming auto-deletion is enabled), then the paths to these files (either absolute or relative to the working directory of the Action in question) should be passed to the [DoNotDelete](#) (see [page 556](#)) utility procedure.

## Post-execution Processing Procedures

The following procedures are called in a specific sequence upon the completion of any Action:

AnalyzePassFail .....	509
GetUcdbCoverage .....	511
IsMergedUcdb .....	511
MergeRerun .....	512
OkToRerun.....	512
OkToMerge .....	513
MergeOneUcdb .....	513
OkToTrend.....	514
OkToTriage .....	515
TriageOneUcdb .....	516
OkToDelete.....	517

### AnalyzePassFail

After each Action completes this procedure will analyze the collateral files generated by the Action and determine if the Action passed or failed. By default, this procedure accomplishes this goal by analyzing the UCDB status, alternatively it can perform a comparison against golden simulation results.

#### Input Arguments

See [Table 15-1](#) on page 501 to [Table 15-4](#) on page 502.

#### Returns

This procedure returns a string beginning with either **passed**, if the Action ran successfully, or **failed**, if the Action encountered errors, potentially followed by a second string showing extended status information, as defined in [Table 3-4, “Possible Values for the Various Status Columns”](#). Anything else is interpreted by VRM as a failure.

#### Default Behavior

The default behavior of this procedure is to first examine the return status of the script.

- A non-zero return status is considered a failure.
- If the return status is zero and the *ucdbfile* value is blank (that is, no UCDB file is defined), the test is considered to have passed.

- If the return status is zero and a non-blank *ucdbfile* is defined, then the *ucdbstat* value is consulted.
  - If the *ucdbstat* field contains either “ok” or “warning,” then the procedure returns “passed.”
  - If any other *ucdbstat* value is found, then the procedure returns “failed.” A message is also sent to the log output in *-debug* mode.
- For tplanScript actions, the contents referenced by the tplanucdb parameter, if any, is used.

For cases where merged UCDB files are analyzed, the following rules apply:

- For **mergeScript** Actions, the contents of the *mergefile* parameter, if any, is used.
- For all other Actions, the contents of the *ucdbfile* parameter, if any, is used.
- If the *ucdbfile* or *mergefile* parameter points to a readable UCDB file, the **TESTSTATUS** attribute from the test-data record associated with that UCDB file is examined.
- If the UCDB file is a merged UCDB file, the Action is considered to have failed if the **TESTSTATUS** attribute contains a “Merge Error” value.
- If the UCDB file is a leaf-level UCDB file, the Action is considered to have failed if the **TESTSTATUS** attribute contains any value other than **OK** or **Warning**.

## Side Effects

If the *status* value is zero and the *ucdbfile* value is non-blank, then the global status counter corresponding to the *ucdbstat* value (completion status from the UCDB file) is incremented.

Actions that define *ucdbfile* in the RMDB but fail to produce a UCDB file will be counted under the “UCDB Error” category in the final status summary.

This procedure can copy and/or move files around as required by the user's regression environment. If the UCDB file is moved (to an archive directory, for example), then the new absolute path should be stored in the *ucdbfile* element of the data array. If any files are to be preserved during the auto-deletion step (assuming auto-deletion is enabled), then the paths to these files (either absolute or relative to the working directory of the Action in question) should be passed to the [DoNotDelete](#) (see page 556) utility procedure.

If the analyzed UCDB file is the result of testplan import, the only statuses that contribute to the global status counter are of TESTSTATUS: Error or Warning.

## GetUcdbCoverage

This procedure provides cached access to the coverage data in a UCDB file, fetching either total coverage or testplan coverage

### Input Arguments

**Table 15-6. Input Arguments for GetUcdbCoverage Procedure**

type	{total   testplan} Type of coverage.
ucdbfile	Path to UCDB file for the Action.

### Returns

This procedure returns the value of the selected coverage attribute from the test-data record associated with the specified file. An empty string is returned if the selected coverage attribute is not found.

Testplan coverage is only available for UCDB files that contain a testplan and at least one test.

## IsMergedUcdb

This procedure tests whether a given UCDB file is or is not a merged UCDB file.

### Input Arguments

**Table 15-7. Input Arguments for IsMergedUcdb Procedure**

ucdbfile	Path to UCDB file for the Action.
----------	-----------------------------------

### Returns

This procedure should return a “true” value (such as 1) if the UCDB file under examination exists, is readable, and contains a global MERGELEVEL attribute. Otherwise, a “false” value (such as 0) should be returned.

---

#### Note

-  In the absence of a merge error, the TESTSTATUS of the test-data record corresponding to a merged UCDB contains the highest severity of all the UCDBs represented in that merged UCDB. This value is not used by VRM (other than to check for a merge error) and is not reported in the status event log.
-

## MergeRerun

This procedure returns the state of the merge/triage auto-rerun function.

### Input Arguments

None

### Returns

This procedure returns a “true” value, by default. Otherwise, a “false” value if the `-nomergererun` option was specified on the `vrun` command line.

## OkToRerun

This procedure is called for each failing Action (whether the failure is a hard failure or a timeout). If the procedure returns a true value, then the failing Action is re-queued for another execution. If the procedure returns a false value, then the Action is not rerun. This procedure can also call the *OverrideRmdbParameter* procedure to alter parameter values (for the Action in question only) prior to the next execution.

### Input Arguments

See [Table 15-1](#) on page 501 to [Table 15-4](#) on page 502.

### Returns

True or false, depending on whether the Action in question should be re-queued for another execution.

### Default Behavior

Returns 1 (true) if the current failure is due to a queue timeout (that is, a grid job got dropped). Returns 1 (true) for any other failure reason, provided that the Action is an *execScript*, the *DEBUGMODE* parameter was defined prior to the failure, the *DEBUGMODE* parameter does not already contain the values true, yes, or 1, and the most recent failure reason has not occurred more than once in the current regression run.

Otherwise, returns 0 (false).

### Side Effects

If the failure is not due to a queue timeout and the *DEBUGMODE* parameter is set to false, no, or 0, a local (that is, effective for this Action only) built-in parameter called *DEBUGMODE* is defined and set to true, yes, or 1 (depending on the original value of the parameter) prior to the second or subsequent executions.

## OkToMerge

This procedure is executed after each Action completes, but only if a UCDB coverage file for the Action has been identified. Its purpose is to determine whether the circumstances related to the Action just completed warrant merging of the UCDB coverage data into a common merge file. If the user environment specifies that only the coverage data from passing simulations should be merged into the final UCDB coverage file, then this is the procedure where that policy is implemented.

### Input Arguments

See [Table 15-1](#) on page 501 to [Table 15-4](#) on page 502.

### Returns

This procedure returns a “true” value (1) if the UCDB coverage data for this Action should be merged with the common merge file. Otherwise, a “false” value (0) is returned.

### Default Behavior

By default, this procedure always returns a “true” value (1). If the *ucdbfile* value is non-blank and if the *passfail* value is “passed” (by default, only passing tests are merged). A message is also written to the log output in *-debug* mode. If the procedure returns a “false” value because the value of *ucdbfile* is blank, then this condition is returned to the log output in *-verbose* mode.

### Side Effects

If this procedure returns a “true” value, then VRM automatically passes the *ucdbfile* path to the [DoNotDelete](#) (see [page 556](#)) utility procedure. If the *ucdbfile* element of the data array is altered by this procedure, then the [MergeOneUcdb](#) (see [page 513](#)) user-definable procedure uses the modified value instead of what was found in the RMDB file. If either the *mergefile* or *mergelist* elements of the data array are set by this procedure, then those values are used by the [MergeOneUcdb](#) user-definable procedure rather than the values of the parameters of the same name in the RMDB file.

## MergeOneUcdb

This procedure is called if the OkToMerge procedure returns a “true” value. It causes the data in the UCDB coverage file for this Action to be merged into a common merge file.

Refer to [OkToMerge](#) for more information.

### Input Arguments

See [Table 15-1](#) on page 501 to [Table 15-4](#) on page 502.

The following arguments can be passed in as data elements (if they are set by the *OkToMerge* (see [page 513](#)) procedure.

**Table 15-8. Arguments Passed in as Data Elements set by OkToMerge**

Member	Description
mergefile	Path of the UCDB file into which the coverage for this Action is merged.
mergelist	Path of the text file to which the UCDB path for this Action is appended.

## Returns

Nothing

## Default Behavior

The *mergefile* and *mergelist* attributes are fetched from the RMDB, if they were not already defined by the *OkToMerge* procedure. If the resulting value of the *mergefile* data element is non-blank, then the merge algorithm (see below) is activated. If the resulting value of the *mergefile* data element is blank but the resulting value of the *mergelist* data element is non-blank, then the path to the UCDB file (as conveyed in the *ucdbfile* data element) is appended to the file indicated by the (absolute or relative) path indicated by the *mergelist* data element. The *mergefile* and *mergelist* attributes can be either absolute paths or paths relative to the head of the VRMDATA directory.

If the merge algorithm is activated, *vrun* checks for the presence of a UCDB file at the path indicated by the *mergefile* data element. If no such file exists, the file at the path indicated by the *ucdbfile* data element is simply copied to the location indicated by the *mergefile* data element. This avoids a first-file race condition not handled by *vcover merge* implementation. If the specified merge file already exists, then the *ucdbfile* path is passed to the auto-merge mechanism for merging.

## Side Effects

In incremental merge mode, an additional *mergeScript* Action can be added to the execution graph. In queued incremental merge mode, whether or not the completion of a given Action results in the generation of a *mergeScript* depends on the order of completion of the various Actions relative to each other and to the time required to complete a single merge. In that case, the appearance of *mergeScript* Actions can be somewhat unpredictable.

## OkToTrend

This procedure is called at the end of the regression. It is called for each *mergefile* marked for trending, using the context of the Task where the *trendfile* was first encountered. The *mergefile* is passed as the *ucdbfile* input argument for the procedure.

## Input Arguments

See [Table 15-1](#) on page 501 to [Table 15-4](#) on page 502, as well as the argument:

**Table 15-9. Arguments Passed in as Data Elements set by OkToTrend**

Member	Description
trendfile	Pathname of the target trend UCDB file. <sup>1</sup>

1. This argument must be independent of any task directory so that it is not subject to auto-clean functionality.

## Returns

This procedure returns a “true” value (1) if the *ucdbfile* should be trended in its corresponding *trendfile* parameter. Otherwise, a “false” value (0) is returned.

## Default Behavior

By default, this procedure returns a “true” value (1), unless the *ucdbstat* is a merge error.

## Side Effects

VRM creates a [trendScript](#) action if the procedure returns true.

## OkToTriage

This procedure is executed after each Action completes, but only if a UCDB coverage file for the Action has been identified. Its purpose is to determine whether the circumstances related to the Action just completed warrant adding the log messages from the Action to the triage database. If the user environment specifies that only the messages from failing simulations should be added to the triage database, then this is the procedure where that policy is implemented.

## Input Arguments

See [Table 15-1](#) on page 501 to [Table 15-4](#) on page 502.

## Returns

This procedure returns a “true” value (1) if the messages for this Action should be added to the triage database. Otherwise, a “false” value (0) is returned.

## Default Behavior

By default, this procedure returns a “true” value (1) if the *ucdbfile* value is non-blank and if the *passfail* value is “failed” (by default, only failing tests are triaged). A message is also written to the log output in *-debug* mode.

## Side Effects

If this procedure returns a “true” value, then VRM automatically passes the *ucdbfile* path to the [DoNotDelete](#) (see page 556) utility procedure. If the *ucdbfile* element of the data array is altered by this procedure, then the [TriageOneUcdb](#) user-definable procedure uses the modified value instead of what was found in the RMDB file. If the *triagefile* element of the data array is set by this procedure, then that value is used by the [TriageOneUcdb](#) user-definable procedure rather than the value of the parameter of the same name in the RMDB file.

## TriageOneUcdb

This procedure is called if the [OkToTriage](#) procedure returns a “true” value. It causes the data in the UCDB coverage file for this Action (and, by reference, the messages in the WLF file for this Action) to be loaded into the failure triage database.

Refer to [OkToTriage](#) for more information.

### Input Arguments

See [Table 15-1](#) on page 501 to [Table 15-4](#) on page 502.

The following argument can be passed in as a data element (if set by the *OkToTriage* procedure).

**Table 15-10. Arguments Passed as Data Elements set by OkToTriage**

Member	Description
triagefile	Path of the triage database into which the message for this Action should be inserted.

### Returns

Nothing

### Default Behavior

The *triagefile* attribute is fetched from the RMDB, if it was not already defined by the *OkToTriage* procedure. If the resulting value of the *triagefile* data element is non-blank, the *ucdbfile* path is passed to the auto-triage mechanism for processing. The *triagefile* attribute can be either an absolute path or a path relative to the head of the VRMDATA directory.

## Side Effects

In incremental triage mode, an additional *triageScript* Action can be added to the execution graph. In queued incremental triage mode, whether or not the completion of a given Action results in the generation of a *triageScript* depends on the order of completion of the various

Actions relative to each other and to the time required to complete a single *triage dbfile* command. In that case, the appearance of *triageScript* Actions can be somewhat unpredictable.

## OkToDelete

This procedure is executed after each Action completes (it is not called for *deleteScript* pseudo-Actions). The purpose of this procedure is to determine if conditions are such that the working directory for the Action may be scheduled for deletion (for example, to implement a “delete-on-pass” disk-space conservation scheme).

Individual files that are not to be deleted should be registered by calling the *DoNotDelete* utility procedure (see “[DoNotDelete](#)” on page 556) if they have not already been registered by one of the other user-definable procedures.

Unlike some of the other post-processing user-definable procedures, *OkToDelete* can be called for Actions which, for some reason, have not actually executed (including Actions for which no script was defined). The reason for this is because the working directory may still need to be cleaned up even if the Action was skipped, killed, dropped, or otherwise came to an untimely end.

The *OkToDelete* procedure works on a veto basis. If multiple Actions execute in a single directory (for example, *preScript* and *postScript*; or *execScript*, *mergeScript*, and *triageScript*), all Actions must return a true value in order for a directory to be deleted. If the *OkToDelete* procedure returns a false value when called for any Action executed in a given directory, that directory is not scheduled for deletion.

### Input Arguments

See [Table 15-1](#) on page 501 to [Table 15-4](#) on page 502.

The *ucdbfile*, *ucdbstat*, and *randseed* elements is blank in cases where there is no UCDB file or the Action did not execute. The *faillog* element is blank for Actions other than *execScript*, Actions that did not execute, or if the *-faillog* command-line option is not specified. The *status* element is blank for Actions that did not execute (in fact, the *status* element is probably the best indication of whether *OkToDelete* is being called as the result of an Action completion event or some other reason).

The *passfail* element contains one of the keywords listed in [Table 15-11](#).

**Table 15-11. Keywords for passfail Element**

Passfail	Reason
dropped	Action was not run (either StopRunning returned true or due to a bug in vrun).
empty	Action was skipped because no commands were specified in the RMDB database file.

**Table 15-11. Keywords for passfail Element (cont.)**

Passfail	Reason
failed	Action failed to launch or executed and failed.
killed	Action was killed by user intervention.
noexec	Action was not executed due to the -noexec command-line option.
passed	Action executed and passed.
skipped	Action was skipped due to earlier errors in dependent Actions.
timeout	Action exceeded timeout (for any reason).

If a UCDB file was generated by the Action script, then the *ucdbstat* element will contain one of the keywords listed in the [Table 15-12](#):

**Table 15-12. Keywords for ucdbstat Element**

ucdbstat	Meaning
ok	Simulation passed
warning	Simulation passed with warnings
error	Simulation reported at least one error
fatal	Simulation reported a fatal error
missing	Simulation did not run
merge	An error occurred during merging
ucdb	The UCDB file could not be read

Note that unlike the *ActionCompleted* user-definable procedure, where a precise status is required for those Actions that were actually eligible to run, the *OkToDelete* user-definable procedure is mainly concerned with whether the status warrants keeping the working directory around. Therefore, additional “non-eligible” status types are represented and timeouts are not differentiated as to their cause.

## Returns

This procedure returns a TCL “true” value (1, true) if the working directory can be deleted. Otherwise, a “false” value (0, false) is returned.

## Default Behavior

By default, this procedure always returns a “false” (0) value. A message is also written to the log output in *-debug* mode.

## Side Effects

If this procedure returns a “true” value, the working directory for the current Action, along with its contents, may be scheduled for deletion. Certain collateral files can be protected from deletion by passing the absolute or relative path of those files to the [DoNotDelete](#) (see [page 556](#)) utility procedure. Files and/or directories outside of scope of the VRMDATA directory are not deleted. Directories that contain at least one protected file or directory will, themselves, be considered as protected (recursively).

## Regression Control Procedures

The following procedures allow the user to customize the behavior of key portions of the VRM regression algorithm:

<b>SelectRunnables</b> .....	<b>520</b>
<b>StopPropagating</b> .....	<b>521</b>
<b>StopRunning</b> .....	<b>522</b>
<b>GetNextIteration</b> .....	<b>523</b>

### SelectRunnables

This procedure allows the user to capture the list of Runnables selected from the command line or to customize how Runnables are selected. This procedure is called immediately after the runlist (if any) is read and the command-line arguments have been processed into an include/exclude list (but just before the include/exclude list is expanded into the Runnable selection (sparse) tree).

The list of selected Runnables is passed to this procedure in the *eiselect* data element as a list of lists (see [Table 15-13](#)). This procedure can modify that list as it sees fit. If the modified list is to be used in place of the original list, this procedure should return a true (1) value. Otherwise, if a false value is returned, the original include/exclude list is used as is.

The include/exclude list is basically a list of sub-hierarchies to include in the run and other sub-hierarchies to exclude. Each element in this list is itself a list composed of two members. The first member of each leaf-level list is a single character indicating the purpose of the list element. The second member of each leaf-level list is a string interpreted according to [Table 15-13](#).

**Table 15-13. Element Members for SelectRunnables**

If 1st member is:	Then 2nd member is:
I	A context string representing a Runnable to be included in the run.
E	A context string representing a Runnable to be excluded from the run.
T	An absolute path to a file containing a list of Runnables to be included in the run.
S	A comma/space separated list of keywords as is passed to the -select command-line option.

For example, a typical include/exclude list might be as follows:

```
{ {I nightly} {E nightly/test2} }
```

which would select for running the test hierarchy under the *nightly* Runnable, with the exception of the *nightly/test2* Runnable. Proper care must be taken to ensure that the list of lists is assembled properly before this procedure returns. If the *eiselect* data member has been modified and the modifications are to be used in building the execution graph, this procedure must return a true (1) value.

## Input Arguments

Also see [Table 15-1](#) on page 501.

**Table 15-14. SelectRunnables Input Arguments**

Member	Description
pass	1 if this procedure is called before the initial run; 2 if called before the failure rerun phase.
eiselect	The include/exclude list as described in <a href="#">Table 15-13</a> .

## Returns

True (1) if the *eiselect* list has been modified, false (0) otherwise.

## Default Behavior

Returns false always.

## Side Effects

If this procedure returns true (1), then the include/exclude list in the data array is used to build the execution graph instead of the original include/exclude list built from the command line options. This modified include/exclude list will also be copied out to the runlist, if one is enabled. It will also affect what is displayed as a result of the *-show* option or any of the graph dump options (that is, all downstream processing of selected Runnables will assume the include/exclude list returned from this procedure was build directly from the command-line options).

## StopPropagating

This procedure is executed after each Action completes (immediately prior to scheduling any dependent Actions for execution). Its purpose is to determine if dependent Actions should be abandoned based on the pass/fail status of the current Action. This is used where, for example, testing cannot continue if a critical compilation step failed. In order to change the VRM policy on when downstream actions should be skipped as a result of the failure of an earlier Action, this is the procedure to override.

## Input Arguments

See [Table 15-1](#) on page 501 to [Table 15-4](#) on page 502. In addition to the procedures listed in these tables, *StopPropagating* gets the additional argument listed in [Table 15-15](#).

**Table 15-15. StopPropagating Input Arguments**

Member	Description
onerror	Set to “abandon” if error propagates downstream.

## Returns

This procedure returns a “true” (1) value if dependent Actions should be abandoned. Otherwise, a “false” value (0) is returned.

## Default Behavior

By default, this procedure returns a “true” (1) value if the Action failed and the *onerror* element of the data array is set to “*abandon*”. When *vrun* builds the execution graph, the *onerror* flag is set to “*abandon*” for all *preScript* Actions and for the *execScript* Action of any Task member of a sequential Group. The net effect is that a failing *preScript* Action will cause all further processing for the associated Group to be skipped, and a failure in any member of a sequential Group will cause all Actions related to subsequent Runnable members of that Group (and the *postScript* Action) to be skipped. A message is also written to the log output in *-debug* mode.

## Side Effects

If this procedure returns a “true” value, processing of dependent Actions is abandoned. VRM emits a warning message for each Action abandoned and, in *-debug* mode, also lists the specific Actions that caused the Action in question to be skipped. The details of how a failing Action affects the execution of subsequent Actions is discussed in [“Execution Graph and Advanced Topology”](#) on page 255.

## StopRunning

This procedure is executed after each launched Action completes or experiences a timeout (immediately prior to scheduling any dependent Actions for execution). It is not called for Actions that are skipped, dropped, or not executed for some reason. Its purpose is to determine if there have been sufficient failures to warrant cancellation of the entire VRM regression run.

## Input Arguments

Also see [Table 15-1](#) on page 501.

**Table 15-16. StopRunning Input Arguments**

Member	Description
fail	Number of failed simulations plus number of failed Action launches.
pass	Number of passed simulations.
sims	Total of “pass” and “fail” (for convenience).

## Returns

This procedure returns a “true” (1) value if the remaining regression run should be abandoned. Otherwise, a “false” value (0) is returned.

## Default Behavior

By default, this procedure always returns a “false” (0) value. A message is written to the log output in *-debug* mode.

## Side Effects

If this procedure returns a “true” value, further processing of Actions is abandoned. VRM emits a warning message and list the Actions that have yet to be executed. Actions that may have been queued to a processing grid are not affected. Launched Actions completing after VRM terminates will not undergo post-processing.

## GetNextIteration

This procedure is called to retrieve the next iteration token to be used in place of the \* token in a dynamically repeating Runnable. The token returned can be any combination of alphanumeric characters, digits, and the special characters period (.), colon (:), underscore (\_), or hyphen (-). If an invalid token is returned, or if the returned token would result in a non-unique context path, then the procedure is called again until a valid and usable token is returned (or until 93 retries have failed). If this procedure returns an empty string, then the repeat loop for the Runnable in question is terminated.

Refer to “[Dynamically Repeating Runnables](#)” on page 233 for more information

## Input Arguments

See [Table 15-1](#) on page 501 and [Table 15-4](#) on page 502.

## Returns

One or more *foreach* token to be used as the iteration value for the current iteration of the Runnable in question.

Returns compound tokens when foreach tokens include characters other than the leading asterisk (\*), for example, “test1-1” and “test1-2” when the foreach token is “\*1”. This is to prevent the inadvertent generation of duplicate tokens.

## Default Behavior

Returns integers from 1 to N with a separate sequence for every value of *ITERATION*. The value of N is determined by expanding the *repcount* parameter from the point of view of the repeating Runnable. The procedure returns an empty string once N integer tokens have been generated or if the *repcount* parameter cannot be expanded.

## Side effects

None.

## Grid Management Procedures

The following user-definable procedures are called by the *vrun* process when attempting to control a grid job.

The procedure name is dynamically constructed by applying “title” capitalization to the string value found in the *gridtype* attribute of the method element used to launch the job, and then appending one of the following strings shown in [Table 15-17](#):

**Table 15-17. Postfix String and Behavior**

Postfix string	Behavior
<gridtype>SubmitOptions	Generate command-line options for submit command. Refer to the chapter “ <a href="#">RMDB Versions</a> ” for information on how this procedure differs between versions.
<gridtype>GetJobId	Retrieve the job ID for a specified Action (usually from the log output).
<gridtype>GetStderr	Fetch the contents of the stderr output file as a string.
<gridtype>KillJob	Kill the grid job.
<gridtype>CacheStatus	Enables collection of cache information.
<gridtype>GetCachedStatus	Fetches the status of a specific Action.
<gridtype>GetLastStatus	Fetches raw data of the grid status command.
<gridtype>StopJob	Suspend the job associated with a specified Action.
<gridtype>ResumeJob	Resume the suspended job associated with a specified Action.

For example, if the *gridtype* attribute of the execution method used to launch a given job was *uge*, then the *UgeKillJob* procedure is consulted when *vrun* wanted to kill the job while the *UgeGetStderr* procedure is called upon completion in order to determine if there were any error messages to be written to the log output stream.

The procedures listed below are those supported internally. Like any other user-definable procedure, they can be overridden to customize VRM behavior.

VRM supports submission of array-jobs to both LSF, UGE, SGE, and RTDA grid management systems.

## <gridtype>SubmitOptions

This procedure is called in order to fetch a list of command-line options which should be passed to the grid submission command of the following grid management systems:

- LsfSubmitOptions — IBM's Platform Load Sharing Facility (LSF)
- SgeSubmitOptions — Sun Grid Engine (SGE) server grid
- UgeSubmitOptions — Univa Grid Engine (UGE) grid
- RtadSubmitOptions — Realtime Design Automation NetworkComputer

There are two ways of adding command-line options to the grid submission command:

- Static — add options directly to the execution method command under the **method** element used to invoke the grid submission command.
- Dynamic — use procedure in case some of the options must be determined dynamically.

### Input Arguments

**Table 15-18. LsfSubmitOptions Input Arguments**

Member	Description
ACTION	Context chain of the Action just completed.
CONTEXT	List of lineal ancestor Runnables.
GRIDTYPE	The value of the gridtype attribute.
INSTANCE <sup>1</sup>	Name of the task being executed.
JOBTYPE <sup>2</sup>	One of “gridarr”, “clubbed”, or “nomulti”.
NUMJOBS	The number of Actions represented in the grid job to be launched.
RUNNABLE	Name of the Runnable element.
SCRIPT	Name of the script (such as preScript, execScript, postScript).
TASKDIR	Path to the working directory for the Action.
TESTNAME	Value of the testname parameter for the Action, if any.

1. The INSTANCE member has the same value as the RUNNABLE member for non-repeating Runnables or a <RUNNABLE>~<ITERATION> for repeating Runnables.
2. This procedure is called once per grid job. A single grid job may represent multiple Actions. If so, the JOBTYPE argument will be “gridarr” for a grid array job or “clubbed” for a job in which the Actions were clubbed internally by VRM. In either case, the NUMJOBS argument will contain the number of Actions included in the grid job.

As with all other user-definable procedures, the arguments are passed as a reference to a TCL array. In addition to the arguments listed above, the common arguments (such as VRMDATA) are also passed to this procedure.

Refer to the chapter “[RMDB Versions](#)” for information on how this procedure differs between versions.

## Returns

The procedure should return a list of command-line options to be used in the job submission command. How the return value is interpreted depends on the RMDB version setting.

- Version 1.0 — the return value is interpreted as a single string.
- Version 1.1 and beyond — the return value is interpreted as a TCL list with one command-line argument per list element.

Be sure to match the format of the returned data to the RMDB version.

The return value of this procedure is used to define a GRIDOPTS parameter when the execution method command is expanded.

## Default Behavior

By default this procedure returns arguments to assign a reasonable name to the job and to redirect the stdout and stderr of the job to appropriately-named files in the working directory associated with the Action.

## Side Effects

This procedure sets the MTI\_VRUN\_JOBIDX\_ENV environment variable to match the variable used to iterate clubbed or array jobs.

## <gridtype>GetJobId

This procedure is called in order to fetch the *JobId* of a job queued to the following grid management systems:

- LsfGetJobId — IBM's Platform Load Sharing Facility (LSF)
- SgeGetJobId — Sun Grid Engine (SGE) server grid
- UgeGetJobId — Univa Grid Engine (UGE) grid
- RtdaGetJobId — Realtime Design Automation NetworkComputer (RTDA) grid

## Input Arguments

See [Table 15-1](#) on page 501 to [Table 15-4](#) on page 502.

## Returns

The *JobId* string of the grid job associated with the specified Action, or a blank string if the *JobId* cannot be determined.

## Default Behavior

This procedure scans the *stdout* file of the batch job that queued the Action script (*VRMDATA/\$ACTION.stdout*) for a message pattern indicating the queuing of an:

- LSF job (^*Job <[0-9]+>* is submitted).
- SGE job (^*Your job [0-9]+ .\** has been submitted).
- UGE job (^*Your job [0-9]+ .\** has been submitted).
- RTDA job (^*Your job [0-9]+ .\** has been submitted).

If the pattern is found, then the *JobId* is extracted from the message and returned as a string. You must specify -savestdout to the vrun command to create this *.stdout* file.

## Side Effects

A warning message is emitted to the log output if a *JobID* is not identified from the *VRMDATA/\$ACTION.stdout* file contents.

## <gridtype>GetJobStatus

This procedure is called in order to fetch the current status of the job associated with an Action queued to the following grid management systems:

- LsfGetJobStatus — IBM's Platform Load Sharing Facility (LSF)
- SgeGetJobStatus — Sun Grid Engine (SGE) server grid
- UgeGetJobStatus — Univa Grid Engine (UGE) grid

## Input Arguments

**Table 15-19. GetJobStatus Input Arguments**

Member	Description
ACTION	Context chain of the Action just completed.
INSTANCE	Name of the task being executed.
ITERATION	Repeat index or foreach token of Runnable instance
RUNNABLE	Name of the Runnable element.

**Table 15-19. GetJobStatus Input Arguments (cont.)**

Member	Description
SCRIPT	Name of the script (such as preScript, execScript, postScript).
TASKDIR	Path to the working directory for the Action.

## Returns

One of the following strings denoting the current status of the grid job:

- ok — The job is listed as either pending, running, suspended, or done.
- error — The job is listed as having returned a non-zero status when executed by the grid management system.
- missing — The job is no longer listed in the queue status report or is listed as unknown or zombie.

## Default Behavior

This procedure checks the output of the bjobs queue status report for the job ID associated with the Action in question. The jobid is obtained by calling the <gridtype>GetJobId user-definable procedure. The bjobs command is only called once every 5 minutes and the output of the command is cached to minimize overhead.

## <gridtype>GetStderr

This procedure is called in order to fetch the *stderr* output of a job queued to the following grid management systems:

- LsfGetStderr — IBM's Platform Load Sharing Facility (LSF)
- SgeGetStderr — Sun Grid Engine (SGE) server grid
- UgeGetStderr — Univa Grid Engine (UGE) grid
- RtdaGetStderr — Realtime Design Automation NetworkComputer (RTDA) grid

## Input Arguments

See [Table 15-1](#) on page 501 to [Table 15-4](#) on page 502.

## Returns

Nothing

## Default Behavior

Searches for one or more files whose name matches the pattern *VRMDATA/\$ACTION.bat.e\** and returns (as a TCL list) the contents of the lexically last file in the list of matches.

## Side Effects

None

## <gridtype>KillJob

This procedure is called in order to kill a job queued to the following grid management systems:

- LsfKillJob — IBM's Platform Load Sharing Facility (LSF)
- SgeKillJob — Sun Grid Engine (SGE) server grid
- UgeKillJob — Univa Grid Engine (UGE) grid
- RtdaKillJob — Realtime Design Automation NetworkComputer (RTDA) grid

## Input Arguments

See [Table 15-1](#) on page 501 to [Table 15-4](#) on page 502.

## Returns

Nothing

## Default Behavior

Calls [`<gridtype>GetJobId`](#) to retrieve the *JobId* associated with the specified Action. If this *JobId* is non-blank (that is, if [`<gridtype>GetJobId`](#) successfully extracted the *JobId*), the *JobId* is passed to the *qdel* command.

## Side Effects

If the *JobID* is successfully retrieved, and if the *qdel* command exists on the path, then the grid job associated with the Action should be terminated and/or deleted from the grid queue. A warning message is emitted to the log output if a *JobID* is not identified from the *VRMDATA/\$ACTION.stdout* file contents. You must specify *-savestdout* to the *vrun* command to create this *.stdout* file.

## <gridtype>CacheStatus

This procedure is used to fetch, parse, and cache the current status of all jobs running on the grid.

This procedure is specific to the following grid management systems:

- LsfCacheStatus — IBM's Platform Load Sharing Facility (LSF)
- SgeCacheStatus— Sun Grid Engine (SGE) server grid
- UgeCacheStatus— Univa Grid Engine (UGE) grid

## Input Arguments

See [Table 15-1](#) on page 501.

## Returns

Returns the word “error” if the grid status command fails or if the output of the grid status command cannot be parsed. It returns the word “ok” if the current grid status was successfully cached.

## Default Behavior

Calls the grid status command appropriate to the grid type. For example, **bjobs** for LSF and **qstat** for SGE and UGE. It then parses the output, and caches the results in the form of a TCL array that maps job IDs to a status of ok (if the job is still pending or running or is reported as having completed successfully) or error (if the job is reported as having exited due to a failure).

## Side Effects

Executes the grid status command appropriate to the grid type.

## <gridtype>GetCachedStatus

This procedure is called in order to fetch the cached status of a specific Action. The <gridtype>CacheStatus procedure must have been called at least once prior to calling this procedure.

This procedure is specific to the following grid management systems:

- LsfGetCachedStatus — IBM's Platform Load Sharing Facility (LSF)
- SgeGetCachedStatus— Sun Grid Engine (SGE) server grid
- UgeGetCachedStatus— Univa Grid Engine (UGE) grid

## Input Arguments

See [Table 15-1](#) on page 501 and [Table 15-2](#) on page 502.

## Returns

Returns one of the following strings:

- ok — The job is listed as pending, running, or suspended.
- error — The job is listed as having failed.
- missing — The job is missing from the grid status report.
- unknown — The job ID cannot be determined.

## Default Behavior

Converts the Action context string to a job ID and returns the current status of said job as reflected in the cached output of the grid status command (which must first be updated using the <gridtype>CacheStatus user-definable procedure). Note that if the Action in question was not launched via an execution method whose method element contained a valid gridtype string, or if the Action was not successfully queued to a grid management system, this procedure will return the string “unknown”. The pro-active check algorithm is designed to only request the current status of Actions which it believes were actually queued to a grid management system.

## Side Effects

None

## <gridtype>GetLastStatus

This procedure is called in order to fetch the raw output of the grid status command for debug purposes. The <gridtype>CacheStatus procedure must have been called at least once prior to calling this procedure.

This procedure is specific to the following grid management systems:

- LsfGetLastStatus — IBM's Platform Load Sharing Facility (LSF)
- SgeGetLastStatus — Sun Grid Engine (SGE) server grid
- UgeGetLastStatus — Univa Grid Engine (UGE) grid

## Input Arguments

None.

## Returns

Returns the captured output of the grid status command as of the last time the <gridtype>CacheStatus procedure was invoked.

## **Default Behavior**

Fetches output from the grid status command.

## **Side Effects**

None

## Logging Procedures

These user-definable procedures allow you to override error/warning message generation and user message generation in the log file.

The **vrun** application invokes these procedures to emit the log output messages rather than emitting the messages directly. By replacing the appropriate procedures, you may intercept these commands and emit messages in the format of your choice.

<b>ProclaimPassFail</b> .....	<b>534</b>
<b>ProclaimUserMessage</b> .....	<b>535</b>

### ProclaimPassFail

This procedure is executed once the pass/fail status of each Action has been determined. The purpose of this procedure is to report the outcome of the Action to the **vrun** log output. It is called once per Action, regardless of whether the Action passed or failed. It is also called prior to the decision to re-run the Action so that initial failures are reported to log output, even if the Action is queued for re-execution.

#### Input Arguments

**Table 15-20. ProclaimPassFail Arguments**

Member	Description
ACTION	Context chain of Action completed
RUNNABLE	Name of Runnable element
ITERATION	Repeat index or foreach token of Runnable instance
INSTANCE	Current Runnable instance, such as RUNNABLE~ITERATION
TASKDIR	Path to the working directory for the Action
SCRIPT	Name of the script, such as preScript, execScript, postScript
passfail	Flag string indicating the outcome of the Action (passed or failed)
faillog	Path to the cumulative list of failed tests
nodelete	List of files not to be deleted (absolute or relative to the working directory)
randseed	Value of the SEED attribute in the UCDB test data record
status	Return status from the Action script

**Table 15-20. ProclaimPassFail Arguments (cont.)**

<b>Member</b>	<b>Description</b>
message	Optional status message from the Action script
ucdbfile	Path to UCDB file for the Action, if any
ucdbstat	Value of the TESTSTATUS attribute in the UCDB test data record

## Returns

Nothing.

## Default Behavior

The default behavior of this routine depends on the value of the *passfail* element of the “userdata” array.

If *passfail* has a value of “passed” and the value of the *message* element of the “userdata” array is not empty, the *message* value is assumed to be a warning message from the Action and the message string is reported along a pointer to the log file for the Action. If the value of the *message* element is an empty string, no log output is produced.

For all other cases, the Action is assumed to have failed. A failure message is emitted with the following additional information:

- The value of the *message* element of the “userdata” array, if non-empty.
- The value of the TESTSTATUS attribute in the UCDB test data record, if a UCDB file was found.
- A pointer to the log file for the Action.
- The contents of the stderr output emitted from the local launch command and/or the grid job

## Side Effects

None.

## ProclaimUserMessage

This procedure is executed in response to the receipt of a user-generated informational message from the Action script. The purpose of this procedure is to report the content of the message to the **vrun** log output. These messages are optional and asynchronous to the VRM flow (save that informational messages can only be received from an Action script that is currently executing).

## Input Arguments

**Table 15-21. ProclaimUserMessage Arguments**

Member	Description
ACTION	Context chain of Action completed
RUNNABLE	Name of Runnable element
ITERATION	Repeat index or foreach token of Runnable instance
INSTANCE	Current Runnable instance, such as RUNNABLE~ITERATION)
TASKDIR	Path to the working directory for the Action
SCRIPT	Name of the script, such as preScript, execScript, postScript
passfail	Contains the string “usermsg”
status	Hostname of the machine on which the originating Action is running
message	Text of the message originated from the Action script

## Returns

Nothing.

## Default Behavior

The default behavior of this procedure is to emit the received message on the **vrun** log output.

## Side Effects

None.

# Utility Procedure Catalog

---

The procedures in the linked sections listed below are not user-definable but are defined in the same namespace so that the various user-defined override procedures can make use of the lower-level facilities in the VRM code.

<b>System-level Utilities .....</b>	<b>538</b>
<b>Path-conversion Procedures .....</b>	<b>548</b>
<b>File and Directory Manipulation Procedures .....</b>	<b>552</b>
<b>Parameter Expansion Procedure .....</b>	<b>558</b>
<b>UCDB Access and Analysis Procedures.....</b>	<b>560</b>
<b>Event Log Access Procedures.....</b>	<b>567</b>
<b>Email-related Procedures .....</b>	<b>569</b>

## System-level Utilities

These are general utilities that do not fall under any of the other categories:

<b>testMode</b> .....	<b>538</b>
<b>ActionToTestname</b> .....	<b>539</b>
<b>SeedRandomGenerator</b> .....	<b>539</b>
<b>GetRandomValues</b> .....	<b>539</b>
<b>ExpandTestlist</b> .....	<b>540</b>
<b>GetTestlistTokens</b> .....	<b>541</b>
<b>GetMostRecentFile</b> .....	<b>542</b>
<b>GetMostRecentFileContents</b> .....	<b>543</b>
<b>GetStatusCounts</b> .....	<b>543</b>
<b>RightNow</b> .....	<b>545</b>
<b>GetNextToken</b> .....	<b>545</b>
<b>GetStderrForAction</b> .....	<b>546</b>
<b>TranslateExtStatus</b> .....	<b>546</b>
<b>TranslateRunStatus</b> .....	<b>547</b>

### testMode

This procedure determines whether the -testname option was specified to the vrun command.

#### Input Arguments

None

#### Returns

True if the -testname option was specified.

#### Example

```
if {[isDebug]} {  
    if {! [testMode] || [string equal $data(TESTNAME) {}]} {  
        logDebug "Debug message related to action '$data(ACTION)'"  
    } else {  
        logDebug "Debug message related to test '$data(TESTNAME)'"  
    }  
}
```

## ActionToTestname

This procedure returns the testname, if any, associated with an Action.

### Input Arguments

action — context chain for the Action in question. This does not support glob characters.

### Returns

The registered testname associated with the Action (or an empty string).

## SeedRandomGenerator

This procedure seeds the TCL random number generator. It is called automatically by VRM each time an RMDB file is expanded if the -mseed command-line option is passed to vrun.

### Input Arguments

seed — Integer value to use as the seed

### Returns

Nothing

### Side Effects

Alters the future random number stream by re-seeding the random number generator.

## GetRandomValues

Returns one or more random integers generated by the (potentially pre-seeded) TCL random number generator. If the optional prefix argument is specified, each generated integer is appended to this prefix string. The resulting values are returned as a TCL list.

### Input Arguments

count — Number of random values to generate.

prefix — String to prepend to each generated value.

### Returns

A list of tokens, each consisting of the optional prefix value and a generated random integer.

## Side Effects

Calling this procedure advances the random stream by one value. Therefore, calling this procedure from concurrent threads or auto-generated constructs (such as mergeScript Actions) could adversely affect the stability of the random number stream. Note that the random numbers generated by this procedure are not suitable for cryptographic purposes.

## Example

You can generate random seeds manually, without the use of a testlist file, by using the GetRandomValues utility procedure, which would be called and the result used in the foreach attribute of a repeating Runnable. The RMDB file would look something like this:

```
...
<runnable name="Simulate" foreach="(%tests%)">
  <parameters>
    <parameter name="tests" type="tcl">[GetRandomValues 3 randtest_]</parameter>
    <parameter name="testname" type="tcl">(%INSTANCE%)</parameter>
    <parameter name="seed" type="tcl">(%ITERATION%)</parameter>
  </parameters>
  <execScript>
    <command>...</command>
    <command>vsim ... -sv_seed (%seed%) ...</command>
    <command>...</command>
  </execScript>
</runnable>
...
```

The GetRandomValues procedure would generate three random seeds (perhaps “123”, “456”, and “789” for example) and append each to the prefix string “randtest\_”, returning the tokens “randtest\_123”, “randtest\_456”, and “randtest\_789”. These tokens would be used by vrun to expand the “Simulate” Runnable into three instances: “Simulate~randtest\_123”, “Simulate~randtest\_456”, and “Simulate~randtest\_789”. Within the Runnable, the value of the testname parameter is derived by stripping off the Runnable name from the Runnable instance name. The seed parameter, used in the vsim command, is derived by stripping the test prefix string from the value of the testname parameter.

## ExpandTestlist

This procedure reads a standard testlist file and returns a list of expanded "tests" (Runnable instances) based on the contents of the testlist file. The path passed to this procedure may contain wildcards which will be expanded by the TCL glob command and, of the matching files, the file with the most recent modification date will be read.

## Input Arguments

pattern — Path to one or more [Testlist Files](#) (possibly via glob wildcards).

args — Either, or both, of the following arguments, in any order:

- -debug — Causes the procedure to print each test specification from the file onto stdout.
- -seed — Causes the expanded test names to contain the generated seed value rather than a one-based integer instance number.

## Returns

A TCL list of expanded tests, each of which consists of a two-item list containing the expanded testname and a TCL list of key/value pairs. The structure would look something like this:

```
{ {test_1 {debug 0 ... seed 123}}  
  {test_2 {debug 0 ... seed 456}}  
  ...  
  {test_N {debug 1 ... seed 789}} }
```

## Side Effects

Calling this procedure can advance the random number sequence by one or more steps.

## GetTestlistTokens

This procedure reads a standard testlist file and returns a list of expanded testnames suitable for use in the foreach attribute of a repeating Runnable. The path passed to this procedure may contain wildcards which will be expanded by the TCL glob command and, of the matching files, the file with the most recent modification date will be read.

## Input Arguments

pattern — Path to one or more [Testlist Files](#) (possibly via glob wildcards).

## Returns

A flat list of expanded testnames. Unless otherwise specified by the testlist, the testnames will consist of the base name specified in the testlist and the generated seed value, separated by an underscore.

## Side Effects

Calling this procedure can advance the random number sequence by one or more steps.

## Example

You can expand a testlist manually from within the RMDB file, using a simple testlist file (in this case, called "mytests"):

```
randtest 3
```

**Note**

 In this case, the seeds could have been supplied directly from the testlist file but to keep things the same as in the earlier manual example, the seeds are omitted here, forcing vrun to generate three random seeds. For the sake of the example, we are assuming those randomly-generated seeds are “123”, “456”, and “789”, as unlikely that may be in practice.

---

The GetTestlistTokens utility procedure would be called and the result used in the foreach attribute of a repeating Runnable. The RMDB file would look something like this:

```
...
<Runnable name="Simulate" foreach="(%tests%)">
  <parameters>
    <parameter name="tests" type="tcl">[GetTestlistTokens mytests]</parameter>
    <parameter name="testname" value="(%ITERATION%)"/>
    <parameter name="seed" type="tcl">[lindex [split (%ITERATION%) "_"] 1]</parameter>
  </parameters>
  <execScript>
    <command>...</command>
    <command>vsim ... -sv_seed (%seed%) ...</command>
    <command>...</command>
  </execScript>
</Runnable>
...
...
```

The GetTestlistTokens procedure would read the testlist and return a number of testname tokens, based on the testlist contents. In this case, it would return the three tokens: “randtest\_123”, “randtest\_456”, and “randtest\_789”. The seed parameter, used in the vsim command, is derived by stripping the text prefix string from the value of the testname parameter.

## GetMostRecentFile

This procedure returns a path to the most recently modified file which matches the passed pattern. The path passed to this procedure may contain wildcards which will be expanded by the TCL glob command and, of the matching files, the file with the most recent modification date will be read.

### Input Arguments

pattern — Path to one or more testlist files (possibly via glob wildcards)

### Returns

A path to the most recent file which matches the passed pattern.

## GetMostRecentFileContents

This procedure returns the contents of the most recently modified file which matches the passed pattern. The path passed to this procedure may contain wildcards which will be expanded by the TCL glob command and, of the matching files, the file with the most recent modification date will be read.

### Input Arguments

`pattern` — Path to one or more testlist files (possibly via glob wildcards)

### Returns

The contents of the most recent file which matches the passed pattern, as a TCL list of lines.

## GetStatusCounts

This procedure returns the counters associated with a given categorization of Actions.

### Input Arguments

`<status_keyword>` — One of the following values:

- `prepared` — Counts whether the preparation of the set of scripts and other files for each action was successful or not. This does not include actions that are empty, skipped or dropped. The counters that make up this category are:
  - `ok` — Number of Actions whose scripts and other files were successfully prepared.
  - `ng` — Number of Actions which encountered an error while preparing scripts and other files.
- `launched` — Counts whether the launch was successful as actions are launched. This does not include actions that are empty, skipped, dropped, or that failed during the script-generation process. The counters that make up this category are:
  - `ok` — Number of Actions successfully launched.
  - `ng` — Number of Actions that failed to successfully launch.
- Actions that are queued to a grid system are considered launched when the grid submission command is successfully launched. All Actions are launched as background processes, which means errors can occur during the execution process which are not reported back to the launching vrun process. These errors are generally detected as queuing timeouts.
- `completed` — Counts whether the status code of actions, as they are reported as completed, was zero (success) or non-zero (failure). This does not include actions that

do not run (empty, skipped, dropped), failed to launch, or timed-out before completion. The counters that make up this category are:

- ok — Number of Actions that completed with a zero status.
- ng — Number of Actions that completed with a non-zero status.

This count is irrespective of whether a UCDB was generated and is independent of the pass/fail decision made by the AnalyzePassFail user-definable procedure.

- ucdb — Counts the UCDB TESTSTATUS values for the actions defined in the RMDB as generating a UCDB. This does not include actions that do not define a ucdbfile parameter. The counters that make up this category are:
  - ok — Number of Actions that passed without warning.
  - warning — Number of Actions that passed with one or more warnings.
  - error — Number of Actions that encountered an error.
  - fatal — Number of Actions that encountered a fatal error.
  - missing — Number of Actions reported as “missing” by the UCDB.
  - merge — Number of Actions that failed to properly merge (should only happen for a mergeScript Action).
  - unknown — Number of Actions with a valid UCDB but an invalid TESTSTATUS value.
  - ucdb — Number of Actions whose UCDB file was missing or unreadable.

An action can create a valid and error-free UCDB file and then fail in some other way (for example, non-zero status code or non-empty stderr file). If the extended status value returned by the AnalyzePassFail user-definable procedure does not match the TESTSTATUS read from the UCDB file, the TESTSTATUS value is neither recorded in the status event log nor counted in this categorization.

- all — Counts the total number of actions that return each of the raw status values, as defined in [Table 3-4 “Possible Values for the Various Status Columns”](#). The counters that make up this category are generated dynamically, therefore only those with a non-zero value will appear.
- passfail — Counts the passed and failed actions as they finish. this does not include actions that are empty. The counters that make up this category are:
  - pass — Number of Actions which passed without warning (UCDB and non-UCDB Actions)
  - warn — Number of Actions which passed with one or more warnings (UCDB Actions only)

- fail — Number of Actions which failed for reasons not associated with the UCDB file
- ucdb — Number of Actions which failed because of their UCDB TESTSTATUS value
- term — Number of Actions which were skipped, dropped, or killed

## Returns

A list of key/value pairs where the key in each case is the name of the counter and the value is the count.

## RightNow

This procedure returns the current date and time as a formatted string.

### Input Arguments

None

## Returns

Formatted time string (for example, “Sat 04 Apr 2009 07:21:14 PM PDT”).

## GetNextToken

This procedure takes a list of tokens from an RMDB parameter and returns the tokens from the list, one at a time.

### Input Arguments

**Table 15-22. GetNextToken Arguments**

Member	Description
parameter-name	Name of the parameter that holds the space-separated list of tokens.
action	Context string of the Action.
tag	(optional) A string to make token strings unique. The last-used token is stored in an array, indexed by a combination of the parameter name and the tag.  You may only want to use this when you call GetNextToken from multiple places in the RMDB with the same parameter name.

## Returns

Each token in the list, one at a time.

## GetStderrForAction

This procedure searches for and returns any stderr output from a specified Action. The stderr output might be used to analyze the pass/fail status of an Action or as part of a failure message on the **vrun** log output. The procedure takes, as input, the context string of the Action whose stderr output is to be fetched and returns the relevant contents.

Related to the [ProclaimPassFail](#) user-definable procedure.

## Input Arguments

**Table 15-23. GetStderrForAction Argument**

Argument	Description
action	Context chain of the Action

## Returns

The value returned is a TCL list containing either one or two elements.

If only one element is returned, that element represents the stderr output of the local launch command (including the stderr output of the user script if the script was executed on the local machine).

If two elements are returned, the second element represents the stderr output of the grid job under which the user script was executed (including the stderr output of the user script).

Returning the stderr output of the grid job requires that the user-definable procedures supporting the type of grid under which the user script was executed are written in such a way that the <gridtype>GetStderr procedure returns the grid job's stderr output.

Each of the elements returned in the TCL list are themselves TCL lists potentially containing multiple lines of stderr output. These lists should be iterated and/or joined before posting the resulting string to the **vrun** log output or to some other file or channel.

## TranslateExtStatus

Translates the **extstatus** value into a human-readable string during generation of text and HTML reports and when displaying status values in the GUI. You can call this procedure from any user-definable procedure but may not override its behavior.

## Input Arguments

**Table 15-24. TranslateExtStatus Argument**

Argument	Description
reason	Two-token status from the <b>reason</b> member of the userdata array.

## Returns

Human-readable version of the extended status, as shown in [Table 3-4, “Possible Values for the Various Status Columns”](#).

## TranslateRunStatus

Translates the **runstatus** value into a human-readable string during generation of text and HTML reports and when displaying status values in the GUI. You can call this procedure from any user-definable procedure but may not override its behavior.

## Input Arguments

**Table 15-25. TranslateRunStatus Argument**

Argument	Description
reason	Two-token status from the <b>reason</b> member of the userdata array.

## Returns

Human-readable version of the run status, as shown in [Table 3-4, “Possible Values for the Various Status Columns”](#).

## Path-conversion Procedures

The following procedures can be used to manipulate relative paths with respect to the various directory trees known and/or managed by VRM:

<b>PathRelativeToDatadir .....</b>	<b>548</b>
<b>PathRelativeToVrun .....</b>	<b>548</b>
<b>PathRelativeToRmdb .....</b>	<b>549</b>
<b>PathRelativeToQuesta.....</b>	<b>549</b>
<b>ActionToLogFile .....</b>	<b>550</b>
<b>ActionToLogFiles.....</b>	<b>550</b>

### PathRelativeToDatadir

This procedure combines a list of directory components into a single path, prepended with the absolute path of the top-most directory of the *VRM Data* area (path given by the *-vrldata* option), and normalizes the result for the local platform.

#### Input Arguments

**Table 15-26. PathRelativeToDatadir Input Arguments**

Argument	Description
args	Path components and/or lists of path components to be combined.

#### Returns

String containing the combined path.

### PathRelativeToVrun

This procedure combines a list of directory components into a single path, prepended with the absolute path of the directory from which *vrun* is launched, and normalizes the result for the local platform.

#### Input Arguments

**Table 15-27. PathRelativeToVrun Input Arguments**

Argument	Description
args	Path components and/or lists of path components to be combined.

## Returns

String containing the combined path.

## PathRelativeToRmdb

This procedure combines a list of directory components into a single path, prepended with the absolute path of the directory from which the RMDB database is loaded, and normalizes the result for the local platform.

## Input Arguments

**Table 15-28. PathRelativeToRmdb Input Arguments**

Argument	Description
args	Path components and/or lists of path components to be combined.

## Returns

String containing the combined path.

## PathRelativeToQuesta

This procedure combines a list of directory components into a single path, prepended with the absolute path of the directory from which the *vrun* application is loaded (which is also the directory in which other Questa utilities can be found), and normalizes the result for the local platform.

## Input Arguments

**Table 15-29. PathRelativeToQuesta Input Arguments**

Argument	Description
args	Path components and/or lists of path components to be combined.

## Returns

String containing the combined path.

## ActionToLogFile

This procedure converts an Action path into a directory path that refers to a file within the working directory for said Action. This is useful for locating the Action's log file should a user-definable procedure wish to emit a message suggesting that the user refer to the Action's log file for more details. The resulting absolute path is not checked against the directory for validation.

### Input Arguments

**Table 15-30. ActionToLogFile Input Arguments**

Argument	Description
ACTION	Action path (as found in the data(action) input argument of most user-definable procedures).
extension	Extension of the file of interest (for example, log, stdout, stderr, and so on).

### Returns

Absolute path name to a (possibly non-existent) file associated with the Action passed to the procedure.

## ActionToLogFiles

This procedure converts an Action path into one or more strings that refer to files within the working directory for said Action. This is useful for locating log files related to a given Action should a user-definable procedure wish to emit a message suggesting that the user refer to the Action's log file for details.

The difference between this procedure and the [ActionToLogFile](#) (see page 550) procedure is that this procedure assumes the resulting absolute “path” contains one or more “glob” characters. This path is then passed to the TCL *glob* function, resulting in a list of zero or more files that match the pattern. Even if the resulting file path contains no *glob* characters, the *glob* function will at least verify that a file exists at that path or an empty list is returned.

### Input Arguments

**Table 15-31. ActionToLogFiles Input Arguments**

Argument	Description
ACTION	Action path (as found in the data(action) input argument of most user-definable procedures).
extension	Extension of the file of interest (for example, log, stdout, stderr, and so on), possibly including one or more glob characters.

## Returns

List of zero or more absolute path names to the file(s) associated with the Action passed to the procedure.

## File and Directory Manipulation Procedures

These procedures support the creation, modification, and deletion of directories, files, and symbolic links from within a user-definable procedure.

By convention, there are only two directory trees where VRM has the authority to create, modify, or delete files:

1. *VRMDATA* directory tree.
2. Directory from which the *vrun* process is launched (that is, the current working directory or *\$cwd* of the *vrun* process).

This restriction is self-imposed and exists to prevent the user from accidentally requesting VRM to delete or overwrite sensitive source files. However, since user-definable procedures are written in pure TCL, there is nothing stopping the user from pressing the standard TCL commands into such service, if necessary.

<b>VerifySafeTarget</b> .....	<b>552</b>
<b>MakeDirectory</b> .....	<b>553</b>
<b>WriteToFile</b> .....	<b>553</b>
<b>AppendToFile</b> .....	<b>554</b>
<b>CopyFile</b> .....	<b>554</b>
<b>LinkToFile</b> .....	<b>555</b>
<b>GetFileLines</b> .....	<b>555</b>
<b>GetFileContents</b> .....	<b>555</b>
<b>DoNotDelete</b> .....	<b>556</b>

### VerifySafeTarget

This procedure verifies that the path in question is within the reach of VRM, defined as anywhere under the *VRMDATA* directory or anywhere under the directory from which the *vrun* process was originally launched. All other paths are off limits for file generation, modification, or deletion.

#### Input Arguments

**Table 15-32. VerifySafeTarget Input Arguments**

Argument	Description
target	Absolute path of target under consideration.

## Returns

True value (1) if the path is within either the `VRMDATA` directory or the directory under which `vrun` was originally launched. Throws an error if the path in question lies outside both of those areas.

## MakeDirectory

Creates a directory, if it does not already exist. Directory creation is recursive so parent directories that do not already exist are also silently created.

This procedure calls `VerifySafeTarget` (see [page 552](#)) to ensure that the directory to be created lies within a part of the directory tree to which VRM has write authority.

## Input Arguments

**Table 15-33. MakeDirectory Input Arguments**

Argument	Description
dir	Absolute path of the directory to be created.

## Returns

String containing the path passed as the `dir` argument. Throws an error if the path in question lies outside the reach of VRM reach or if the directory (or one of its parents) cannot be created.

## WriteToFile

Writes the specified lines to the specified file. Any existing content in the file is deleted. This procedure calls `VerifySafeTarget` to ensure that the file to be written lies within a part of the directory tree to which VRM has write authority.

## Input Arguments

**Table 15-34. WriteToFile Input Arguments**

Argument	Description
filename	Absolute path of the file to be written.
lines	List of lines to write to the specified file.

## Returns

Nothing. Throws an error if the path in question lies outside VRM's reach, if the file cannot be opened for writing, or if an error occurs during the write process.

## AppendToFile

Appends the specified lines to the specified file, preserving any existing content. This procedure calls *VerifySafeTarget* to ensure that the file to be written lies within a part of the directory tree to which VRM has write authority.

### Input Arguments

**Table 15-35. AppendToFile Input Arguments**

Argument	Description
filename	Absolute path of the file to be written.
lines	List of lines to write to the specified file.

### Returns

Nothing. Throws an error if the path in question lies outside the reach of VRM, if the file cannot be opened for appending, or if an error occurs during the write process.

## CopyFile

Copies the contents of one file to a file in another location, creating the target file if necessary and/or overwriting any existing content. If parent directories of the target path do not exist, they are created. This procedure calls *VerifySafeTarget* to ensure that the target file lies within a part of the directory tree to which VRM has write authority (the source path is not checked and can be any path for which the user has read access).

### Input Arguments

**Table 15-36. CopyFile Input Arguments**

Argument	Description
source	Absolute path to the source file.
target	Absolute path to the target file.

### Returns

Nothing. Throws an error if the target path lies outside the reach of VRM, if the source file cannot be opened for reading, if the target file cannot be opened for writing, or if an error occurs during the copy process.

## LinkToFile

Creates a symbolic link from an existing path to another location (supported only on platforms that support symbolic links). If parent directories of the target path do not exist, they are created, although the trailing component of the target path will always end up as a symbolic link. This procedure calls *VerifySafeTarget* to ensure that the directory in which the link is to be created lies within a part of the directory tree to which VRM has write authority (the source path is not checked and can be any path in the directory structure).

### Input Arguments

**Table 15-37. LinkToFile Input Arguments**

Argument	Description
source	Absolute path to the source file.
symbol	Absolute path of the link to be created.

### Returns

Nothing. Throws an error if the target path lies outside VRM's reach, if the target directory cannot be opened for writing, or if an error occurs during the link creation process.

## GetFileLines

Reads and returns the contents of the specified file, stripped of comments and blank lines, and returned as a list (one element per line).

### Input Arguments

**Table 15-38. GetFileLines Input Arguments**

Argument	Description
filename	Absolute path of file to be read.

### Returns

The non-comment text in the file, separated into a TCL list where each element of the list ended with a newline character (\n) in the original file. Throws an error if the specified file cannot be opened for reading or if an error occurs during the read process.

## GetFileContents

Reads and returns the contents of the specified file as a list (one element per line).

## Input Arguments

**Table 15-39. GetFileContents Input Arguments**

Argument	Description
filename	Absolute path of file to be read.

## Returns

The raw text in the file, separated into a TCL list where each element of the list ended with a newline character (\n) in the original file. Throws an error if the specified file cannot be opened for reading or if an error occurs during the read process.

## DoNotDelete

Adds one or more files and/or directories to an internal “no delete” list. Wildcards may be included in the file/directory arguments. Relative files and/or directories are first expanded using the most recent base directory, which may also be an absolute or relative path. The base directory starts out blank. If, after the initial expansion, the resulting path is still relative, it is expanded with respect to the top-mode level of the *VRM Data* directory. Following this, any wildcards in the resulting absolute paths are expanded, and the resulting paths are added to the “no delete” list. Only files that actually exist at the time the *DoNotDelete* procedure is called are added to the file; this means an entire directory of files may be added to the “no delete” list with a single wildcard argument. In the event the wildcard expansion results in an empty list, the expanded path itself is added to the “no delete” list (just in case the wildcard is a literal file/directory name or the path represents an explicit reference to a file that does not yet exist).

The contents of the “no delete” list are used to protect certain collateral files from auto-deletion upon the successful completion of the Action(s) for which they were generated. The “no delete” list is a hash of absolute paths, each of which represents a single protected file or directory. In addition, every directory associated with a Runnable context is marked with a special .VRM flag file and recursive file deletion will not traverse into these directories.

## Input Arguments

This procedure is passed one or more of the following arguments in any sequence:

**Table 15-40. DoNotDelete Input Arguments**

Argument	Description
-dir <base>	Base directory for expanding relative paths (can be blank if all “do-not-delete” paths are absolute).
<path>	Path to file or directory to be preserved (relative to base).

The *-dir* option sets a persistent base directory by which the other path arguments are interpreted. Once the base directory is set via the *-dir* option, all paths passed from that point onward are assumed to be relative to that base directory (except for absolute paths). Before the base directory has been set, passed paths are assumed to be relative to the top of the *VRM Data* directory.

For example, the following command (called from within a user-definable procedure) would mark as preserved the *merge.ucdb* file in the top-level of the *VRM Data* directory, all existing files with the extension *.log* in the working directory for the Action, and the *covhtmlreport* directory in the working directory for the top-level *nightly* Task:

```
DoNotDelete merge.ucdb -dir  
$data(TASKDIR) *.log $data(DATADIR)/nightly/covhtmlreport
```

This is not intended to be a typical example. For the most part, the paths passed to the *DoNotDelete* procedure would fall within the working directory for the Action under consideration as follows:

```
DoNotDelete -dir $data(taskdir)  
*.log *.ucdb *.dump
```

is a more typical command.

Note that while wildcards in the base directory argument are not explicitly disallowed, the behavior of this procedure in this case is neither guaranteed nor tested. It is recommended that any *glob* wildcard characters in the *base* path argument be properly escaped to avoid trouble.

## Returns

Nothing

## Parameter Expansion Procedure

The following procedures are used to support parameter expansion or set/change the parameter value from within a user-definable procedure:

<b>OverrideRmdbParameter</b> .....	<b>558</b>
<b>ExpandRmdbParameters</b> .....	<b>558</b>

### OverrideRmdbParameter

Allows a user-defined TCL procedure to set or change the value of a parameter. The change is effective only for the specified Action. The new value remains effective for the duration of the regression run or until another value is assigned by this same procedure.

#### Input Arguments

**Table 15-41. OverrideRmdbParameter Input Arguments**

Argument	Description
ACTION	Context chain of the Action on whose behalf any parameter references are being expanded.
key	Name of the parameter to be changed.
value	New value for the parameter.

#### Returns

Nothing.

### ExpandRmdbParameters

Expand the parameters in an arbitrary string from the point of view of a given Action.

#### Input Arguments

**Table 15-42. ExpandRmdbParameters Input Arguments**

Argument	Description
ACTION	Context chain of the Action on whose behalf any parameter references are being expanded. If you leave this argument blank, the API will globally search for, and expand, parameter values in the “line” argument.
line	Arbitrary text string, possibly containing parameter references.

## Returns

A text string corresponding to the string passed in the *line* argument, but with any parameter references resolved. Throws a error in the event a circular parameter loop is detected. Emits a warning message if one or more parameters have no definition (however, the reference to the undefined parameter is simply replaced with a blank string and the result is still returned).

## UCDB Access and Analysis Procedures

The following procedures simplify access to UCDB files from user-definable procedures:

<b>GetUcdbContextAttribute</b> . . . . .	<b>560</b>
<b>GetUcdbContextSummaryAttribute</b> . . . . .	<b>560</b>
<b>GetUcdbContextSeed</b> . . . . .	<b>561</b>
<b>GetUcdbContextStatus</b> . . . . .	<b>561</b>
<b>ConvertUcdbContextStatus</b> . . . . .	<b>562</b>
<b>IsCoverstore</b> . . . . .	<b>563</b>
<b>IsTplanUcdbContext</b> . . . . .	<b>563</b>
<b>IsSweepUcdbContext</b> . . . . .	<b>564</b>
<b>FindMaxTeststatus</b> . . . . .	<b>564</b>
<b>TranslateUcdbContextStatus</b> . . . . .	<b>564</b>
<b>CheckUcdbContextLock</b> . . . . .	<b>565</b>
<b>ValidateUcdbContext</b> . . . . .	<b>565</b>

### GetUcdbContextAttribute

Fetches the value of the specified attribute from a singleton test data record in the specified UCDB file. By default, this procedure makes use of a C-based TCL library that calls the UCDB API in read-streaming mode to fetch the test data records.

#### Input Arguments

**Table 15-43. GetUcdbContextAttribute Input Arguments**

Argument	Description
ucdbContextfile	Absolute path to the UCDB file of interest.
attrkey	Attribute keyword whose value is to be fetched.

#### Returns

The value of the specified attribute (as a string). A blank string is returned if the UCDB file contains multiple test data records. Throws an error if the UCDB file cannot be read or if *vcover* returned an error status. This utility procedure does cache its results.

### GetUcdbContextSummaryAttribute

Fetches a summary attribute value from a given UCDB file. By default, this procedure makes use of a C-based TCL library that calls the UCDB API in read-streaming mode to fetch the

summary records. The *GetUcdbSummaryAttribute* attribute gives the user access to the global summary attributes.

## Input Arguments

**Table 15-44. GetUcdbSummaryAttribute Input Arguments**

Argument	Description
ucdbfile	Absolute path to the UCDB file of interest.
attrkey	Attribute keyword whose value is to be fetched.

## Returns

The value of the specified attribute (as a string). A blank string is returned if the UCDB file contains multiple summary data records. Throws an error if the UCDB file cannot be read or if *vcover* returned an error status. This utility procedure does not cache its results.

## GetUcdbSeed

Fetches the value of the *SEED* attribute from a singleton test data record in the specified UCDB file. Uses *GetUcdbTestAttribute* to fetch the attribute value.

## Input Arguments

**Table 15-45. GetUcdbSeed Input Arguments**

Argument	Description
ucdbfile	Absolute path to the UCDB file of interest.

## Returns

The value of the *SEED* attribute (as a string), or an empty string if the *SEED* attribute does not exist, the UCDB has no test data records, or the UCDB has multiple test data records. Throws an error if the UCDB file cannot be read or if *vcover* returned an error status.

## GetUcdbStatus

Fetches the value of the *TESTSTATUS* attribute from a singleton test data record in the specified UCDB file. Uses *GetUcdbTestAttribute* to fetch the attribute value.

## Input Arguments

**Table 15-46. GetUcdbContext Input Arguments**

Argument	Description
ucdbfile	Absolute path to the UCDB file of interest.

## Returns

One of the following strings, according to the value of the *TESTSTATUS* attribute found in the UCDB file:

**Table 15-47. Strings Returned for GetUcdbContext**

String	Interpretation
ok	Simulation passed without warnings or errors.
warning	Simulation warnings were issued.
error	Simulation errors occurred.
fatal	Fatal errors occurred.
missing	Simulation was not run (possibly potential test data record).
merge	Coverage merge error occurred.
unknown	Unknown status value found in test data record.
ucdb	Cannot find/open UCDB file or cannot find test data record(s).

These strings represent the names of the status counter categories within VRM. The *TESTSTATUS* attribute stored in the UCDB test data record is an integer. This integer is converted to a string as its fetched from the UCDB in order to make debugging of VRM easier and in order to isolate to a single procedure any indexing problems that can arise as a result of the integers used for this UCDB attribute changing in any way (the TCL code used by VRM has no way to access the *enum* variable that holds the list of possible values, or to verify that the understanding of these values is correct).

## ConvertUcdbContext

This procedure records (counts) the UCDB status passed as the sole input argument and returns either the string “passed” or the string “failed,” according to the status.

## Input Arguments

**Table 15-48. ConvertUcdbContext Input Arguments**

Argument	Description
ucdbstat	Status string (one of the strings listed in the Returns section of the <a href="#">GetUcdbContext</a> procedure).

## Returns

This procedure returns the string “passed” if the input status is “ok” or “warning.” Otherwise, returns the string “failed.”

## IsCoverstore

This procedure determines if a given UCDB file is a coverstore.

## Input Arguments

**Table 15-49. IsCoverstore Input Arguments**

Argument	Description
ucdbfile	Path to the UCDB file or coverstore

## Returns

This procedure should return 1 if ucdbfile points to a valid coverstore.

## IsTplanUcldb

This procedure tests whether a given UCDB file is a result of testplan import.

## Input Arguments

**Table 15-50. IsTplanUcldb Input Arguments**

Argument	Description
ucdbfile	Path to the UCDB file for the Action

## Returns

This procedure should return a “true” value (such as 1) if the UCDB file under examination exists, is readable, and is the result of a testplan import. Otherwise, a “false” value (such as 0) is returned.

## IsSweepUcdb

This procedure tests whether a given UCDB file is a sweep UCDB from a Questa ADMS flow.

### Input Arguments

**Table 15-51. IsSweepUcdb Input Arguments**

Argument	Description
ucdbfile	Path to the UCDB file for the Action

### Returns

This procedure should return a “true” value (such as 1) if the UCDB file under examination exists, is readable, and is a sweep UCDB file. Otherwise, a “false” value (such as 0) is returned.

## FindMaxTeststatus

This procedure determines the maximum TESTSTATUS of all the tests in a mergefile.

### Input Arguments

**Table 15-52. FindMaxTeststatus Input Arguments**

Argument	Description
ucdbfile	Path to the UCDB file for the Action

### Returns

This procedure will return the maximum integer TESTSTATUS value.

## TranslateUcdbContextStatus

Translates the status string returned by the *GetUcdbContextStatus* procedure into a more aesthetically pleasing form for use in status reports.

### Input Arguments

**Table 15-53. TranslateUcdbContextStatus Input Arguments**

Argument	Description
ucdbstat	Status string (one of the strings listed in the Returns section of the <a href="#">GetUcdbContextStatus</a> procedure).

## Returns

Aesthetically pleasing form of the UCDB status (that is, *warning* returns “Warning” while *merge* returns “Merge error”).

## CheckUcdbLock

This procedure checks for the presence of a lock on a specified UCDB file.

### Input Argument

**Table 15-54. CheckUcdbLock Input Arguments**

Argument	Description
ucdbfile	Path and name of the UCDB file to be checked

## Returns

If the UCDB file is not locked, the procedure returns zero (0) immediately.

If the UCDB file is locked, the procedure will wait up to 120 seconds or until the lock clears, whichever comes first, returning a zero (0) if the lock is clear or a one (1) if the UCDB file is still locked after the delay.

## ValidateUcdb

This procedure sorts UCDB files from coverstores and those with invalid paths.

### Input Argument

**Table 15-55. ValidateUcdb Input Arguments**

Argument	Description
ucdbfile	Path and name of the UCDB file to be checked
ucdbpathref	Name of variable to store real UCDB path (optional)
ucdbtestref	Name of variable to store UCDB testname (optional)

### Description

The variable named by ucdbpathref, if specified and non-blank, will be populated by the real path to the UCDB file or coverstore. This value will be the same as ucdbfile if ucdbfile points to a file or the path portion of a coverstore[:test] combination if the ucdbfile string points to a test in a coverstore.

The variable named by ucdbtestref, if specified and non-blank, will be populated by the testname if the ucdbfile string points to a test in a coverstore or blank otherwise.

The variables pointed to by ucdbpathref and ucdbtestref will not be changed if the ucdbpath is invalid.

## Returns

Returns true if the path points to a valid UCDB or coverstore.

## Event Log Access Procedures

The following procedures simplify access to the event log file from a user-definable procedure:

<b>FetchEventRecords</b> .....	<b>567</b>
<b>FetchEventActions</b> .....	<b>568</b>
<b>FetchEventUcdbs</b> .....	<b>568</b>

### FetchEventRecords

Scan for and fetch “done” status event messages from the event log for the current regression run. This procedure can be passed one or more options to select how the records are scanned and what information is to be returned. The return value is a list containing one element per matching status event message, the content of which is the returned data string.

#### Input Arguments

**Table 15-56. FetchEventRecords Options**

Option	Description
-action <regexp>	Regular expression to filter by Action context chain.
-status <regexp>	Regular expression to filter by completion status.
-script <regexp>	Regular expression to filter by script name (preScript, execScript, postScript, and so on).
-return <select>	Specify which data field to extract and return.

If the *-return* option is specified, the value of said option determines what information is returned as shown in [Table 15-57](#):

**Table 15-57. FetchEventRecords -return Selections**

Select	Returns
ACTION	Context chain for the Action (that is, action field).
RUNNABLE	Name of the leaf-level Group or Task responsible for the Action.
status	Completion status.
ucdbfile	Absolute path to the UCDB file for the Action.

If the *-return* option is not specified, then the entire status event message for each matching event is returned.

## Returns

A list of context chains if *-return action* is specified, a list of completion status strings if *-return status* is specified, a list of UCDB files if *-return ucdbfile* is specified, or a list of status event messages if no *-return* option is specified.

## FetchEventActions

Scans for “done” status event messages from the event log for the current regression run and returns a list of Action context chains for all matching events. This is implemented as *FetchEventRecords -return action* so all the selection options listed under that procedure are supported.

### Input Arguments

See [Table 15-56](#) and [Table 15-57](#).

## Returns

A list of context chains found in matching events in the event log.

## FetchEventUcdbs

Scans for “done” status event messages from the event log for the current regression run and returns a list of UCDB files for all matching events. This is implemented as *FetchEventRecords -return ucdbfile* so all the selection options listed under that procedure are supported.

### Input Arguments

See [Table 15-56](#) and [Table 15-57](#).

## Returns

A list of absolute UCDB file paths derived from matching events in the event log.

## Email-related Procedures

List of TCL utility procedures related to VRM Email functionality

<b>AutoEmail</b> .....	<b>569</b>
<b>AutoEmailSignature</b> .....	<b>569</b>
<b>GenerateStatusSummary</b> .....	<b>569</b>
<b>GetDefaultEmailParameters</b> .....	<b>570</b>
<b>GetEmailParameters</b> .....	<b>570</b>
<b>SendEmailMessage</b> .....	<b>571</b>

### AutoEmail

This procedure examines the RMDB that looks for relevant email settings and, if the required settings are found, composes and sends an email message. The built-in default values are used for any settings which are not provided in the input argument and are not found in the relevant RMDB parameters.

#### Input Arguments

settings — An optional list of key/value pairs, which will override the default settings and/or the settings discovered in RMDB parameters

#### Returns

Nothing. Will issue an error in case of failure.

### AutoEmailSignature

This procedure returns the pre-canned signature which is appended to the bottom of every email generated by the AutoEmail procedure.

#### Input Arguments

None

#### Returns

A string containing the pre-canned signature to be appended to the bottom of an email message.

### GenerateStatusSummary

This procedure returns a string containing one or more of the status summary sections emitted to the vrun standard output at the end of a regression run. It may be used to compose the body of

an email message to be sent either during the regression run (with partial results) or at the end of the regression run.

## **Input Arguments**

flags — A TCL list of one or more of the following flags

- all — Equivalent to “exe cpl sim apf”
- apf — Action script pass/fail status
- cpl — Action script completion status
- exe — Action script execution status
- gen — Action script generation status
- sim — Test (valid UCDB) status

## **Returns**

A string containing the same text that would have been emitted for the selected status summary sections at the end of the regression.

## **GetDefaultEmailParameters**

This procedure is used for initializing an email settings list with default values.

## **Input Arguments**

None

## **Returns**

A list of key/value pairs containing the default values for each of the email settings.

## **GetEmailParameters**

This procedure is used to fetch the relevant email settings from one or more RMDB parameters.

## **Input Arguments**

settings — An optional list of key/value pairs to be used as default values in the event one or more RMDB parameters are not found

## **Returns**

A list of key/value pairs containing the combined values for each of the email settings.

## SendEmailMessage

Composes and sends an email message. This procedure uses the built-in default values for any settings which are not provided in the input argument.

### Input Arguments

settings — An optional list of key/value pairs which will override the default settings and/or the settings discovered in RMDB parameters

### Returns

Nothing. Will issue an error in case of failure.



# Appendix A

## VRM Database (RMDB) API

---

VRM RMDB database is implemented as an XML file. The VRM application uses the API defined in this document to access the data in the database. The API is designed to support the semantics of the database, and not necessarily its implementation. It should be possible to implement an equivalent database using, for example, an SQL-compatible library and use the same API to access the data.

<b>Types of Elements</b> .....	<b>573</b>
<b>Data and Meta-data</b> .....	<b>575</b>
<b>Definitions of Variables Used in Commands</b> .....	<b>576</b>
<b>Command Shortcuts</b> .....	<b>577</b>
<b>Attributes Per Element Type</b> .....	<b>578</b>
<b>Element Attribute Default Values</b> .....	<b>582</b>
<b>Parameter and Script Inheritance</b> .....	<b>583</b>
<b>Document Type Definition and Validation</b> .....	<b>583</b>
<b>Including Nested XML Files</b> .....	<b>583</b>
<b>Global API</b> .....	<b>584</b>
<b>Read access API</b> .....	<b>587</b>
<b>Write Access API</b> .....	<b>590</b>
<b>Cached Access API</b> .....	<b>597</b>

## Types of Elements

There are seven main types of data elements in the VRM database. These types are as follows:

- action
- command
- member
- method
- parameter
- runnable
- usertcl

Note that *action* is a pseudo-element: the database uses the elements *preScript*, *execScript*, and *postScript* to encode action data (however, these three distinct element types are referred to in this document as the single *action* element type).

A *command* element contains a single parameterized command. The *parameter* element defines a single name-value pair that is used in the expansion of commands, other parameters, and some element attributes. Both the *command* element and the *parameter* element contain, as text content, a single free-form string that itself can contain other parameter references to be expanded by the VRM application.

The *Runnable* element is the major building block for regression suites. A *Runnable* can be one of the following four types:

- A *Runnable* element of type “task” defines a leaf-level regression Task. These elements are usually associated with a single simulation.
- A *Runnable* element of type “group” with a *sequential* attribute set to “yes” defines a Group of Tasks (or other Groups) that must be run in the order specified in the list of members.
- A *Runnable* element of type “group” with a *sequential* attribute set to anything other than “yes” (or with no *sequential* attribute at all) defines a Group of Tasks (or other Groups) that can be run in any sequence.
- A *Runnable* element of type “base” acts as a placeholder of information. It can define the same kinds of information as a Task or a Group but it cannot itself be executed. A “base” *Runnable* is often used to prevent duplication where there is a need for some group of Runnables to all define the same information in the same way.

A *Runnable* element can define zero or more *parameter* elements. The *parameter* elements are themselves contained in a *parameters* container element that is used only for grouping purposes.

A *Runnable* element of type “group” can define zero or more *member* elements. These *member* elements are themselves contained in a *members* container element that is used only for grouping purposes. Each *member* element contains, as its text content, the name of another *Runnable* element that is considered to be a member of the Group in which the *member* element is found.

A *Runnable* element can define one, two, or three *action* elements known as *preScript*, *execScript*, and *postScript*. The *execScript* element is used in the execution of *Runnable* elements of type “task.” The *preScript* and *postScript* elements are used in the execution of *Runnable* elements of type “group.” Because of Action script inheritance, it is often useful for a *Runnable* of type “group” to define an *execScript* element as well (which is only used by the Task descendants of that Group).

The *action* element defines zero or more *command* elements that make up the executable commands used to accomplish the Action in question. The *preScript* Action is executed prior to

any of the tests in the Group in which it is defined. The *execScript* Action is executed for each leaf-level Task (that is, runnable of type “task”). The *postScript* Action is executed after all the members of the Group have completed execution.

The *method* element defines a single command template that is used as a wrapper around each Action script to launch the script onto a compute grid or to layer on some other execution model (such as execution in the background of the local machine, where such execution is supported). The *method* element contains only a single *command* element. A *runnable* element can define one or more *method* elements. These *method* elements can be conditional or unconditional and are used to provide the execution method command for Action scripts executed under the Runnable in which the *method* element is defined.

The *usertcl* element defines a TCL code fragment used to override certain user-definable procedures within VRM.

A single *rmdb* element serves as the top-level element in the XML implementation (this is often called the “document element”). For the XML file to be valid, there must be only one *rmdb* element and it must occur at the top-level of the element hierarchy. All *runnable* and *usertcl* elements are contained within this *rmdb* element. One or more *method* elements can also be defined under the *rmdb* element. These can be used both as base elements referred to by other *method* elements or as global conditional execution method elements.

## Data and Meta-data

In hardware design, there is a “data path” and the “control path.” The data path generally takes the form of a multi-bit bus and any value that can be represented in the given number of bits can be conveyed via that data bus. The control path is discrete and can only take on the states for which it was designed.

Likewise, in software design there is “data” and “meta-data.” Data generally exists as free-form information, often supplied directly by the user. Meta-data is more constrained, taking on only certain predefined values to control the behavior of the system in question. A number would be data. Its type (*char*, *int*, *float*, *double*) is usually considered meta-data.

This is relevant to the VRM database in that there are two mechanisms for encoding data in the database. Command strings, parameters, and the names of members of a group are considered “data.” These values are generally encoded as the content of elements in the XML file. Method and parameter names, runnable names and types, timeouts, template references and launch mode flags are considered “meta-data.” For the most part, these fragments of information are limited to the specific values supported by VRM (for example, the launch mode flag must be either *exec* or *tcl* and nothing else). The meta-data is generally encoded as XML attributes attached to the elements in the XML file.

Another way to think about this is as follows:

- Parameters are generally free-form, user-defined, and freely available for use in macros.

- Attributes are system-defined, limited in scope, constrained to only specific values, and not directly visible to the user (except possibly via GUI widgets in the editor).

## Definitions of Variables Used in Commands

The TCL tokens representing names and/or strings associated with objects in the database are described in this section.

**Table A-1. Variables Used in Commands**

Token	Description
actionName	Name of an action element in the database (must be preScript, execScript, or postScript).
attrName	Name of an element attribute (legal attribute names are defined for each element type).
attrValue	Value assigned to an element attribute.
booleanValue	True or false value in any of the forms acceptable to the TCL interpreter.
commandString	String (possibly containing macros) to be used as an executable command in a method or an action.
commentText	String appearing as a comment in the XML database file.
dbFile	Name/path of the XML file containing VRM data.
dbHandle	Unique name assigned (by rmdb open) to an open VRM database.
dtdName	Name/path of the Document Type Definition (DTD) for the RMDB database format.
elementTag	Tagname of a specific XML element in the database.
localfileKey (*)	Name or index (see note below) of a localfile element in the database.
methodKey (*)	Name or index (see note below) of a method element in the database.
methodName	Name of a method element in the database.
paramName	Name of a parameter element in the database.
paramString	String (possibly containing macros) assigned as the value/content of a named parameter element.
runnableName	Name of a runnable element in the database.
runnableType	Value of the type attribute of a runnable element in the database (must be group, task, or base).
tclCode	A string containing valid TCL code.

**Table A-1. Variables Used in Commands (cont.)**

Token	Description
traceFile	Name/path of the file into which API commands are logged.
usertclName	Name of a usertcl element.
xslFile	Name/path of an XSLT stylesheet.

Note that since *localfile* and *method* elements can be anonymous, access to a specific element can be by name or by index. An index is an ordinal (one-based) integer preceded by an at sign (@). The integer index *N* refers to the *Nth* method element within the specified parent element. The special-case index @*end* refers to the last such element defined within the specified parent element (this is useful for referring to an element that was just added to the database, since elements are always added in document order). As of this writing, only *localfile* and *method* elements can be accessed by index but this functionality can be extended to other elements in the future.

## Command Shortcuts

Like most TCL commands, the RMDB API command keywords can be abbreviated. For example, instead of:

```
$dbHandle add runnable $runnable $type
```

you could use the command:

```
$dbHandle ad r $runnable $type
```

Abbreviation only applies to subcommand keywords. Attribute names (such as *type*), Action script names (such as *execScript*), enumerated values (such as yes or no), and so on, must still be spelled out in full.

### Special Case for method Access

For method access, the *rmdb method* command accepts a combined method/runnable key in addition to a *simple* method element name (or index). If a slash character is present in the *methodKey* string, it is used as the delimiter between the *runnable* element name and the *method* element name. **Table A-2** shows the combinations that can be used to access methods both at the top-level of the database and from within *runnable* elements:

**Table A-2. Keys for rmdb method Command**

Key	Method	Location
m1	Method m1	Top-level
@N	Nth method	Top-level
/m1	Method m1	Top-level

**Table A-2. Keys for rmdb method Command (cont.)**

Key	Method	Location
/@N	Nth method	Top-level
r1/m1	Method m1	Runnable r1
r1/@N	Nth method	Runnable r1
r1/	illegal	
In keeping with XML/XPATH tradition, the N in @N is one-based. That is, if there are three method elements in a given Runnable, they are accessed as @1, @2, and @3. There is no @0.		

## Attributes Per Element Type

The following tables describe the attributes defined for each element in the RMDB database. Attributes other than the ones listed can be added to the XML file but are recognized by neither VRM or the RMDB Edit GUI. In addition, if non-supported attributes are added to the database XML file, there is a non-zero chance their names can collide with supported attributes in a future release. User-defined attribute names should be prefixed by a company specific string to reduce the risk of such name collisions.

The RMDB DTD file (*vm\_src/rmdb.dtd*) recognizes only the attributes listed below. If non-standard attributes are added to the RMDB file, then VRM, RMDB API, and the RMDB Edit GUI will preserve the user-defined attributes but the XML file will not validate unless the DTD is updated to include the new attributes. Validation is an optional step provided by the RMDB API only.

### Document (root) Elements

Table A-3 contains the attribute information for the document root elements.

The *created* and *modified* attributes are inserted by RMDB Edit GUI and are defined in the DTD. However, VRM makes no use of either value. They are only for the user's convenience.

**Table A-3. Document Attributes**

Name	Mandatory	Default	Expanded?	Description
created	No		No	Date database file is created.
loadtcl	No		No	List of usertcl elements to be auto-loaded.
modified	No		No	Date database file was last modified.

**Table A-3. Document Attributes (cont.)**

Name	Mandatory	Default	Expanded?	Description
version	No	1.0	No	Version of VRM database. Alternate value is 1.1. Refer to the Appendix <a href="#">RMDB Versions</a> for more information.

**runnable Elements**

[Table A-4](#) contains the *runnable* attributes information.

The *sequential* attribute only applies to *runnable* elements whose *type* attribute is set to “group.”

**Table A-4. runnable Attributes**

Name	Mandatory	Default	Expanded?	Description
base	No		No	Base Runnable (list of runnable elements on the lateral inheritance chain).
foreach	No		Param/TCL	List of tokens for foreach repeat expansion.
if	No		Param/TCL	Positive conditional expression for conditional execution.
name	Yes		No	Name of the runnable.
repeat	No		Param/TCL	Integer number of iterations for repeat expansion.
sequential	No	no	No	Set to yes if the runnable element represents a sequential Group (must be yes or no).
type	Yes		No	Type of the runnable (must be group, task, or base).
unless	No		Param/TCL	Negative conditional expression for conditional execution.

**parameter Elements**

[Table A-5](#) contains the *parameter* attributes information.

**Table A-5. Parameter Attributes**

Name	Mandatory	Default	Expanded?	Description
name	Yes		No	Name of the parameter.

**Table A-5. Parameter Attributes (cont.)**

Name	Mandatory	Default	Expanded?	Description
type	No	text	No	Set to tcl if parameter value is to be expanded by the TCL interpreter (must be tcl or text).

#### action Elements (preScript, execScript, and postScript)

Table A-6 contains the *action* attributes information.

The name of an Action element is derived from its XML tag and is mandatory.

**Table A-6. action Attributes**

Name	Mandatory	Default	Expanded?	Description
file	No		Param	Path to file containing parameterized Action script commands.
launch	No	vsim	Param	Launch mode of the action (can be vsim or exec, or the path to an interpreter),
mintimeout	No		No	Action-specific minimum timeout.
usestderr	No	yes		When set to “no”, the run manager ignores non-empty <i>stderr</i> files when analyzing the pass/fail status of the Action script. Useful for instances where scripts emit messages to <i>stderr</i> even though they pass. For global control of this behavior use the <b>-nostderr</b> option to the <b>vrun</b> command

#### localfile Elements

Table A-7 contains the *localfile* attributes information.

**Table A-7. localfile Attributes**

Name	Mandatory	Default	Expanded?	Description
name	No		No	Destination of copy/link (that is, local file name).
src	No		No	Source of file/directory to be copied/linked.
type	No	copy	No	Either copy to copy the file or link to create a symlink.

**method Elements**

Table A-8 contains the *method* attributes information.

**Table A-8. method Attributes**

Name	Mandatory	Default	Expanded?	Description
action	No		No	Conditional string that matches Action script type (preScript, execScript, postScript).
base	No		No	Base method (list of method elements on the lateral inheritance chain).
context	No		No	Conditional string that matches calling context (partial match, like -g/-G).
if	No		Param/TCL	Positive conditional expression for conditional selection.
mintimeout	No		No	Method-specific minimum timeout.
name	No		No	Name of the method (if anonymous, method element must be accessed by index in document order).
Runnable	No		No	Conditional string that matches Runnable on whose behalf the Action is being executed.
unless	No		Param/TCL	Negative conditional expression for conditional selection.

**usertcl Element**

Table A-9 contains the *usertcl* attributes informations.

**Table A-9. usertcl Attributes**

Name	Mandatory	Default	Expanded?	Description
base	No		No	Base usertcl element (list of usertcl elements on the lateral inheritance chain).
file	No		Param	Path to file containing TCL override code (used in lieu of element contents).
name	Yes		No	Name of the usertcl element.

## Element Attribute Default Values

The XML format does not require defined attributes to actually be specified in any given XML file. For the most part, an attribute that is not specified for a given element assumes the empty string as its default value. This property makes creation of an RMDB file less tedious, as attributes whose value would consist of an empty string can be omitted from the file. There are a few attributes, however, whose value must take on one of a list of possible strings. An empty string on one of these attributes would be meaningless. For most of these attributes, the API returns a default value if the attribute is not specified in the XML file.

The *launch* attribute of the Action scripts can also take on other values not defined here. A non-empty value that does not match the strings *vsim* or *exec* is assumed to be the path of an interpreter program (such as */bin/csh*), which can execute the commands in that Action script.

If an enumerated-type attribute is missing from an element, then the API returns the default value listed for that attribute. If the default value is written via the Write-mode API, then the actual value is written into the attribute (that is, the attribute is not deleted if set to its default value). In order to delete an attribute, the attribute should be written using an empty string as the value. In the case of the enumerated-type attributes, this will return the attribute to its default value (for these attributes, the empty string is not allowed as a value). For the *type* attribute of the *Runnable* element, which has no logical default value, writing the attribute with a blank will render the database invalid (which a validity check would detect).

Table A-10 lists these enumerated-type attributes, along with the list of possible values for each and the default assumed if no value is given for the attribute.

**Table A-10. Enumerated Type Attributes**

Element	Attribute	Values	Default
rmdb	version		1.0  Alternate value is 1.1. Refer to the Appendix <a href="#">RMDB Versions</a> for more information.
Runnable	type	group/task/base	(value required, no default)
Runnable	sequential	yes/no	no
parameter	type	text/tcl	text
preScript/execScript/postScript	launch	vsim/exec, etc.	vsim
localfile	type	copy/link	copy

## Parameter and Script Inheritance

In VRM, parameters and action scripts defined in a group are “inherited” by the members of that group (recursively, if necessary). In other words, assume a group *alltests* has, as a member, a test *test1*. Assume, also, that *test1* does not define a parameter by the name of *fred* but a parameter of that name is defined by the group *alltests*. An action script command within *test1* that includes the macro (%*fred*%), when expanded, will satisfy that macro with the value of the *fred* parameter in the group *alltests* because *test1* does not define a parameter by that name. Action scripts also inherit in the same way.

This inheritance is a function of the VRM application. The VRM Database API does not follow group membership (it cannot, since any given test can be a member of multiple groups and the calling hierarchy cannot be known until one or more runnables are executed). The API also does not follow template references. The API only reports what is actually found in the database. It is up to the application using the API to implement any inheritance semantics layered over the raw configuration data.

## Document Type Definition and Validation

A Document Type Definition (DTD) file is included in the Questa release (*vm\_src/rmdb.dtd*). The *validate* command can be used to validate the contents of a given RMDB file but this validation is not performed by default in either VRM or the RMDB Edit GUI. The validation process checks to be sure that the elements in the file occur in legal places and that the attributes on each have legal values (including checking whether the enumerated-type attributes have values from their respective enumerated lists).

## Including Nested XML Files

The LibXML2 parser used in the RMDB API includes support for XInclude, a standard for defining and importing nested XML files. In the RMDB API, the XInclude expansion is only performed if the current XInclude namespace is defined in the document (*rmdb*) element. For example,

```
<rmdb version="1.1" ... xmlns:xi="http://www.w3.org/2003/XInclude">
  ...
</rmdb>
```

Note the *xi* part of the namespace declaration is user-defined. The convention is to use the string *xi* but any legal identifier can be used. The URI to which the namespace is assigned is the key to whether XInclude is enabled. However, if a string other than *xi* is used, the namespace for the *include* element described below must also be changed to match the identifier used to define the XInclude namespace.

If the proper XInclude namespace is defined, the entire document can be expanded via the *expand* command. Wherever the contents of a nested RMDB file fragment are to be included,

the *include* element under the XInclude namespace should be inserted into the parent RMDB file as follows:

```
<rmdb version="1.1" ... xmlns:xi="http://www.w3.org/2003/XInclude">
  <xi:include xi:href="child.rmdb"/>
  <runnable name="nightly" type="group" base="rootbase">
    ...
  </runnable>
  ...
</rmdb>
```

The child XML file would then contain only those elements that were to be inserted into the parent file as follows:

```
<runnable name="rootbase" type="base">
  ...
</runnable>
```

Note that the child XML file is not a complete RMDB file, as it does not contain an *rmdb* document element. However, the child XML file must be otherwise match the DTD.

Both the *include* element and the *href* attribute must be under the defined XInclude namespace (see above). The *href* attribute can be a relative file name (relative to the current working directory), an absolute file name, or a URI that resolves to a file (*file:/home/fred/myrunnables.rmdb*).

The *expand* command must be executed on an open RMDB database. If the XInclude namespace is not defined in the parent RMDB file, then the *expand* command is ignored.

In the API implementation, the XInclude namespace definition is recognized if defined on any element. Per XML rules, it is only necessary to define the namespace on or above any element on which it is used. However, the XInclude namespaces should always be defined on the document (*rmdb*) element only (to guarantee future RMDB API compatibility).

Refer to “[Split RMDB Database Among Several Files](#)” on page 64 for additional information.

## Global API

The following API commands are used to open/close and manage VRM database files. When the *rmdb* TCL package is loaded, the *rmdb* command is defined. This command is used to open a database file. Each open database is assigned a unique handle. This handle is used as the TCL command for all subsequent accesses to that particular in-memory database. The XML file is parsed completely when the database is open and there is no further access to the file. If modifications are made to the in-memory database, it must be written out in order to affect the XML file on the disk.

The keywords listed in these commands can be abbreviated, following the usual TCL rules. If the abbreviated keyword is sufficient to uniquely identify the operation to be performed or the

information requested, the abbreviation is recognized as valid. Element and/or attribute names cannot be abbreviated. Values containing spaces and/or other punctuation must be delimited in the usual TCL manner.

## Create, Open, Save, and Close Database Files

`rmdb new`

Creates a new (blank) database in memory.

`rmdb open dbFile [_mode_]`

Opens database by reading file at *dbFile*. Returns a string-based handle to the database (*dbHandle*).

The optional *mode* argument on the *rmdb* open command can be set to *ro* for read-only access or *rw* for read-write access. Read-only mode can be used to prevent wayward applications from inadvertently modifying the VRM database file when they were only designed to read and make use of the data therein.

If the *mode* argument is not specified, then the RMDB file is opened in read-write mode. An RMDB file created with the *new* command is always opened in read-write mode.

`dbHandle expand`

Expands any XInclude elements in the database. (This is an experimental feature and is only available if compiled into the API under a compile-time switch.)

The *expand* command is ignored unless the XInclude namespace attribute is defined in the *rmdb* document element.

`dbHandle save dbFile [_xslFile_]`

Saves the (in-memory) database represented by *dbHandle* back to an XML file. If the *xslFile* option is specified, the XSLT style sheet at that location is applied to the database prior to saving.

If the in-memory database is already associated with an XML file (that is, if it was opened from an XML file or saved to an XML file at any time), then the *dbFile* argument is optional. The database is saved to the XML file associated with the database.

If the in-memory database was created with the *rmdb new* command and has never been saved, then a *dbFile* must be specified or the command will return an error.

If the *dbFile* argument is specified as a literal hyphen (-), then the XML file data is dumped to the application's standard output.

`rmdb text [_xslFile_]`

Returns a formatted string containing the XML contents of the current database. If the *xslFile* option is specified, then the XSLT style sheet found at that location is applied to the database prior to generating the string.

```
dbHandle close
```

Closes the database represented by *dbHandle*.

If any data has been written to the database since it was last saved (or since it was opened, if it has never been saved), then the *close* command returns an error. The right way to get around the error is to save the database. If you really want to close without saving, use the *dirty* command (see below) to clear the dirty flag before calling *close*.

## Information Regarding Database Status

```
dbHandle file
```

Returns the name of the XML file associated with the database.

```
dbHandle info
```

Returns information (filename) related to the database.

```
dbHandle dirty
```

Returns “yes” or “no,” depending on whether the in-memory database has unsaved changes. A “yes” response means unsaved changes exist (that is, the in-memory copy is *dirty*).

```
dbHandle dirty booleanValue
```

Force the *dirty* state of the database. This can be used either to bypass the normal no-save-if-dirty check or to signal to the application (when the *close* command is used) that the database cannot be safely closed without saving it first.

```
dbHandle validate dtdName
```

Preforms DTD validation of in-memory database (created by *rmdb new* or loaded by *rmdb open*) against the DTD file at *dtdName*.

For convenience, there is a DTD file for the VRM database in the Questa release directory (*vm\_src/rmdb.dtd*).

## Control of Per-Command Debug Tracing

```
rmdb trace on traceFile
```

Enable global tracing of API calls. Each command is written to the *traceFile* that is opened in write mode as a result of this command. The file remains open until tracing is disabled.

```
rmdb trace off
```

Close the *traceFile* and disable API tracing.

## Information Regarding Enumerated Attribute Choices and Defaults

```
dbHandle default elementName attributeName
```

Returns the default value for the *attributeName* attribute of the *elementName* element. This command is only valid for the enumerated attributes listed in [Table A-10](#) on page 582.

```
dbHandle values elementName attributeName
```

Returns a list of the possible values for the *attributeName* attribute of the *elementName* element. This command is only valid for the enumerated attributes listed in [Table A-10](#) on page 582.

The list for the *launch* attribute of the *action* elements includes only the predefined values for the attribute. It does not include any user-defined values, even if these are used elsewhere in a given database. If a comprehensive list of values appearing in this attribute is needed, then the application should scan the database and compile the list.

## Read access API

The following commands are used to read information from the VRM database. The “command” used in each case is the handle (*dbHandle*) returned from the *rmdb open* command.

### Database and/or document Element

Note that the “plural” commands in this section also have a “singular” command to access details about a particular element. Therefore, these commands cannot be abbreviated.

```
dbHandle attr attrName
```

Returns the value of the *attrName* attribute of the document record.

```
dbHandle groups
```

Returns a list of the names of Groups (that is, Runnables of type *group*) defined in database. If there are no Groups in the database, an empty list is returned.

```
dbHandle methods
```

Returns a list of the names of all *methods* elements defined at the top-level of the database. Anonymous methods are returned as an empty string. If there are no top-level *method* elements in the database, an empty list is returned.

```
dbHandle params
```

Returns a list of the *name* attributes of all *parameter* elements defined in the database. If there are no parameters in the database, then an empty list is returned.

```
dbHandle runnables
```

Returns a list of the names of all Runnables (of all types) defined in the database. If there are no Runnables in the database, an empty list is returned.

```
dbHandle tasks
```

Returns a list of the names of all Tasks (that is, Runnables of type *task*) defined in database. If there are no Tasks in the database, an empty list is returned.

```
dbHandle usertcls
```

Returns a list of the names of all *usertcl* elements defined in the database. If no *usertcl* elements are defined, an empty list is returned.

## Runnable Elements

```
dbHandle runnable runnableName type
```

Returns the type of the *runnableName* runnable (that is, group, task, or base).

```
dbHandle runnable runnableName attr attrName
```

Returns the value of the *attrName* attribute of the *runnableName* runnable.

```
dbHandle runnable runnableName hasaction actionPerformed
```

Returns 1 if the *runnableName* runnable defines the *actionName* action, or 0 otherwise.

```
dbHandle runnable runnableName members
```

Returns the list of names of members of the *runnableName* Group (if the Runnable is a *group*). Returns an empty list of no members are defined.

```
dbHandle runnable runnableName params
```

Returns a list of the *name* attributes of all the *parameter* elements defined within the *runnableName* runnable.

```
dbHandle runnable runnableName localfiles
```

Returns a list of the attributes of all *localfile* elements defined within the *runnableName* runnable. Each member of the list is itself a list of key/value pairs that can be fed to the TCL *array set* command. All known attributes are included.

```
dbHandle runnable runnableName methods
```

Returns a list of the *name* attributes of all the *method* elements defined within the *runnableName* runnable. If a given *method* element has no *name* attribute, then an empty string exists in place of the missing name.

## parameter Elements

```
dbHandle runnable runnableName param paramName
```

Returns the value of the *paramName* parameter under the *runnableName* runnable.

```
dbHandle runnable runnableName param paramName attr attrName
```

Returns the value of the *attrName* attribute of the *paramName* parameter under the *runnableName* runnable.

## action Script Elements

```
dbHandle runnable runnableName action actionPerformed attr attrName
```

Returns the value of the *attrName* attribute of the *actionName* action under the *runnableName* runnable.

```
dbHandle runnable runnableName action actionPerformed commands
```

Returns the list of commands defined in the *actionName* action under the *runnableName* runnable.

## localfile Elements Defined Under runnable Elements

```
dbHandle runnable runnableName localfile localfileKey attr attrName
```

Returns the value of the *attrName* attribute of the *localfileKey* method under the *runnableName* runnable.

```
dbHandle runnable runnableName localfile localfileKey commands
```

Returns the list of commands defined in the *localfileKey* method under the *runnableName* runnable.

## method Elements Defined under runnable Elements

```
dbHandle runnable runnableName method methodKey attr attrName
```

Returns the value of the *attrName* attribute of the *methodKey* method under the *runnableName* runnable.

```
dbHandle runnable runnableName method methodKey commands
```

Returns the list of commands defined in the *methodKey* method under the *runnableName* runnable.

## method Elements Defined under document (rmdb) Element

```
dbHandle method methodKey attr attrName
```

Returns the value of the *attrName* attribute of the *methodKey* method under the document (*rmdb*) element.

```
dbHandle method methodKey commands
```

Returns the list of commands defined in the *methodKey* method of the *methodName* method under the document (*rmdb*) element.

A *method* is defined as a single command used to wrap an action script, primarily for submission to a compute grid. While the internal format of the database supports multiple commands, all commands but the first are ignored.

The *rmdbmethod* command supports a special case “runnable/method” syntax for *methodKey*. See [Special Case for method Access](#) for details.

## usertcl Elements

```
dbHandle usertcl usertclName attr attrName
```

Returns the value of the *attrName* attribute of the *usertclName usertcl* element.

```
dbHandle usertcl usertclName content
```

Returns the TCL content of the *usertclName usertcl* element.

# Write Access API

The following API commands are used to modify an open VRM database. The write-access API consists of the *add*, *clear*, and *delete* commands and (in the case of *runnable* elements), the *copy* command. The *add* command adds a data item to an existing element (possibly overwriting an existing data item). The *clear* command removes a specified type of content from an existing element. The *delete* command deletes the selected element from the database. The *copy* command clones a *runnable* element under a new name.

The write access API calls are guarded by a read-only flag set at the time the RMDB file is opened. If the RMDB file was opened in read-only mode, then an error is thrown if any of the write-mode API calls are invoked.

## Modify the document Element

Note that other *add* commands are listed under the specific element the command is intended to add. For example, the *\_dbHandle\_ add runnable* command is listed under the [Add and/or Modify Runnables](#).

```
dbHandle add attr attrName attrValue
```

Add an attribute to the document (*rmdb*) element.

```
dbHandle add comment commentText
```

Add a comment to the document (*rmdb*) element.

```
dbHandle clear all
```

Delete all *method*, *runnable*, and *ucdbtcl* elements (that is, all top-level elements) in the database.

```
dbHandle clear methods
```

Delete all top-level *method* elements in the database (does not affect *method* elements defined under *runnable* elements).

```
dbHandle clear runnables
```

Delete all *runnable* elements in the database.

```
dbHandle clear usertcl
```

Delete all *usertcl* elements in the database.

## Add and/or Modify Runnables

```
dbHandle add runnable runnableName runnableType
```

Add a Runnable named *runnableName* of type *runnableType* to the document (*rmdb*) element. If a *runnable* element with the name *runnableName* already exists, it is deleted.

```
dbHandle runnable runnableName add attr attrName attrValue
```

Add/set an attribute on the *runnableName runnable* element.

```
dbHandle runnable runnableName add member memberName
```

Add the *memberName* Runnable to the member list of the *runnableName* Runnable.

```
dbHandle runnable runnableName add comment commentText
```

Add a comment to the *runnableName* Runnable.

```
dbHandle runnable runnableName clear all
```

Delete the member list, parameter list, and any scripts from the *runnableName* Runnable.

```
dbHandle runnable runnableName clear members
```

Delete the member list and its contents from the *runnableName* Runnable.

```
dbHandle runnable runnableName clear localfiles
```

Delete all *localfile* elements from the *runnableName* Runnable.

```
dbHandle runnable runnableName clear methods
```

Delete all *method* elements from the *runnableName* Runnable.

```
dbHandle runnable runnableName clear params
```

Delete the parameter list and its contents from the *runnableName* Runnable.

```
dbHandle runnable runnableName copy newRunnableName
```

Copy the *runnableName* Runnable to *newRunnableName* (as a new Runnable).

```
dbHandle runnable runnableName delete
```

Delete the *runnableName* Runnable.

## Add and/or Modify Parameters

```
dbHandle runnable runnableName add param paramName paramValue
```

Add a parameter named *paramName* with value *paramValue* to the *runnableName* Runnable. If a parameter element with the name *paramName* already exists, its value is overwritten.

```
dbHandle runnable runnableName param paramName add attr attrName attrValue
```

Add/set an attribute on the *paramName parameter* element in the *runnableName* Runnable.

```
dbHandle runnable runnableName param paramName add comment commentText
```

Add a comment to the *paramName parameter* element in the *runnableName* runnable.

```
dbHandle runnable runnableName param paramName clear
```

Delete the content (and any child elements) from the *paramName parameter* element in the *runnableName* Runnable.

```
dbHandle runnable runnableName param paramName delete
```

Delete the *paramName parameter* from the *runnableName* Runnable.

## Add and/or Modify Actions

```
dbHandle runnable runnableName add action actionPerformed
```

Add an empty *actionName* Action to the *runnableName* Runnable. If an Action script element with the name *actionName* already exists, it is deleted.

```
dbHandle runnable runnableName action actionPerformed add attr attrName
attrValue
```

Add/set an attribute on the *actionName* Action in the *runnableName* Runnable.

```
dbHandle runnable runnableName action actionPerformed add command commandString
```

Add a command to the *actionName* Action in the *runnableName* Runnable.

```
dbHandle runnable runnableName action actionPerformed add comment commentText
```

Add a comment to the *actionName* Action in the *runnableName* Runnable.

```
dbHandle runnable runnableName action actionPerformed clear all
```

Delete all child elements and/or content from the *actionName* Action in the *runnableName* Runnable.

```
dbHandle runnable runnableName action actionPerformed clear commands
```

Delete the commands from the *actionName* Action in the *runnableName* Runnable.

```
dbHandle runnable runnableName action actionPerformed delete
```

Delete the *actionName* Action from the *runnableName* Runnable.

## Add and/or Modify Top-level method Elements

Note that the *rmdbmethod* command supports a special case “runnable/method” syntax for *methodKey*. See [Special Case for method Access](#) for details.

```
dbHandle add method [_methodName_]
```

Add a *method* element (with the optional name *methodName*) to the document (*rmdb*) element. If *methodName* is specified and a *method* element with that name already exists, it is deleted.

```
dbHandle method methodKey add attr attrName attrValue
```

Add/set the *attrName* attribute on the *methodKey method* element.

```
dbHandle method methodKey add command commandString
```

Add a command to the *methodKey method* element.

```
dbHandle method methodKey add comment commentText
```

Add a comment to the *methodKey method* element.

```
dbHandle method methodKey clear all
```

Delete all child elements and/or content from the *methodKey method* element.

```
dbHandle method methodKey clear commands
```

Delete any command elements from the *methodKey method* element.

```
dbHandle method methodKey delete
```

Delete the *methodKey method* element.

## Add and/or Modify localfile Elements within Runnables

```
dbHandle runnable runnableName add localfile [_localfileName_]
```

Add a *localfile* element (with the optional name *localfileName*) to the *runnableName* Runnable. If *localfileName* is specified and a *localfile* element with that name already exists, it is deleted.

```
dbHandle runnable runnableName localfile localfileKey add attr attrName  
attrValue
```

Add/set an attribute on the *localfileKey localfile* element under the *runnableName* Runnable.

```
dbHandle runnable runnableName localfile localfileKey add command  
commandString
```

Add a command to the *localfileKey localfile* element under the *runnableName* Runnable.

```
dbHandle runnable runnableName localfile localfileKey add comment  
commentText
```

Add a comment to the *localfileKey localfile* element under the *runnableName* Runnable.

```
dbHandle runnable runnableName localfile localfileKey clear all
```

Delete all child elements and/or content from the *localfileKey localfile* element under the *runnableName* Runnable.

```
dbHandle runnable runnableName localfile localfileKey clear commands
```

Delete any command elements from the *localfileKey localfile* element under the *runnableName* Runnable.

```
dbHandle runnable runnableName localfile localfileKey delete
```

Delete the *localfileKey method* element under the *runnableName* Runnable.

## Add and/or Modify method Elements within Runnables

```
dbHandle runnable runnableName add method [_methodName_]
```

Add a *method* element (with the optional name *methodName*) to the *runnableName* Runnable. If *methodName* is specified and a *method* element with that name already exists, it is deleted.

```
dbHandle runnable runnableName method methodKey add attr attrName
attrValue
```

Add/set an attribute on the *methodKey method* element under the *runnableName* Runnable.

```
dbHandle runnable runnableName method methodKey add command commandString
```

Add a command to the *methodKey method* element under the *runnableName* Runnable.

```
dbHandle runnable runnableName method methodKey add comment commentText
```

Add a comment to the *methodKey method* element under the *runnableName* Runnable.

```
dbHandle runnable runnableName method methodKey clear all
```

Delete all child elements and/or content from the *methodKey method* element under the *runnableName* Runnable.

```
dbHandle runnable runnableName method methodKey clear commands
```

Delete any command elements from the *methodKey method* element under the *runnableName* Runnable.

```
dbHandle runnable runnableName method methodKey delete
```

Delete the *methodKey method* element under the *runnableName* Runnable.

## Add and/or Modify usertcl Elements

```
dbHandle add usertcl usertclName
```

Add a *usertcl* element named *usertclName* to the document (*rmdb*) element. If a *usertcl* element with the name *usertclName* already exists, it is deleted.

```
dbHandle usertcl usertclName add attr attrName attrValue
```

Add/set an attribute on the *usertclName usertcl* element.

```
dbHandle usertcl usertclName add comment commentText
```

Add a comment to the *usertclName method* element.

```
dbHandle usertcl usertclName add content tclCode
```

Add a TCL content to the *usertclName method* element.

```
dbHandle usertcl usertclName clear all
```

Delete all child elements and/or content from the *usertclName usertcl* element.

```
dbHandle usertcl usertclName delete
```

Delete the *usertclName* *usertcl* element.

# Cached Access API

Some applications (such as the Edit GUI) require quick access to a list of parents of a given Runnable. Since building such a list requires complete traversal of the entire database, a layer exists to cache the membership topology information (parent-to-child and child-to-parent) as it changes. The cache is refreshed when the RMDB file is first opened or upon command. Write-mode API commands are monitored and the cache is updated as the underlying RMDB is updated. As with the raw API, multiple cached RMDB files can be open at any one time and each has its own cache.

<b>Opening and Closing Cached RMDB Files .....</b>	<b>597</b>
<b>Additional Commands.....</b>	<b>598</b>
<b>Enhanced Commands .....</b>	<b>598</b>

## Opening and Closing Cached RMDB Files

Creating an instance of the *CachedRmdb* object opens or creates an RMDB database via the API. In general, the commands that the *CachedRmdb* object accepts are the same as those listed for the raw API above. The *new*, *open*, *close*, and *expand* API commands are not supported, as these are handled internal to the class implementation.

In order to open an RMDB database from an existing file, create an instance of the *CachedRmdb* class passing the name of the file to the constructor as follows:

```
RunManager::CachedRmdb _dbName_ [_options_] _rmdbFileName_
```

In order to create a new RMDB database, simply omit the name of the file as follows:

```
RunManager::CachedRmdb _dbName_ [_options_]
```

The *dbName* argument is the name given to the *CachedRmdb* object. If the string *#auto* is used instead, the *CachedRmdb* class will assign a name like *cachedRmdb<sub>n</sub>*, where *n* is a numeric suffix added to guarantee uniqueness. The options can include any of the following:

**Table A-11. Options for dbName Argument**

Option	Description
-expand	Expand all XInclude references in the database.
-debug	Enable debug mode in the cache.

The *-expand* option invokes the *expand* API command when an RMDB file is first opened (it has no effect if a new database was created without specifying a file). This option name can be abbreviated.

The *-debug* option adds some additional debugging information. This option name can be abbreviated.

The class constructor also returns the name of the *CachedRmdb* object that can be used as a handle. The following code opens an RMDB file called *my.rmdb*, auto-generates a handle (that is, name) for the *CachedRmdb* object, and stores that name in the variable *dbHandle* for use in passing subsequent commands to the object as follows:

```
set dbHandle [RunManager::CachedRmdb {#auto} my.rmdb]
```

When the *CachedRmdb* object is destroyed (manually or by TCL's garbage collection process), the underlying RMDB file is closed. This can result in an error if unsaved changes have been made to the in-memory database. In order to manually delete the *CachedRmdb* object, use the following command:

```
itcl::delete object _dbHandle_
```

## Additional Commands

In addition to the commands documented in the sections above (the basic Read-mode and Write-mode API commands), the *CachedRmdb* class provides its own unique commands. As with the raw API commands, these subcommand keywords can be abbreviated.

```
dbHandle refresh
```

Refresh the membership cache by re-scanning the RMDB database.

```
dbHandle toprunnables
```

Return the list of non-base Runnables that are not in any other Runnable's membership list.

```
dbHandle runnable runnableName icon
```

Returns a string containing the icon identifier for the specified Runnable.

```
dbHandle runnable runnableName parents
```

Returns a list of Runnables that refer to the specified Runnable as a member.

## Enhanced Commands

The following command behaves somewhat differently when using a *CachedRmdb* object as opposed to the raw RMDB API:

```
dbHandle runnable runnableName type
```

Return type of Runnable or “unknown” if Runnable is not found.

The API *Runnable runnableName type* command throws an error if the Runnable cannot be found. The *CachedRmdb* object returns the string “unknown” as an aid to GUI applications that must put something on the screen when the membership list for a given Runnable points to one or more non-existent Runnables. Likewise, the *Runnable runnableName icon* command documented in the section above returns the name of a special “unknown or deleted” icon when the Runnable specified in the command does not exist.



# Appendix B

## Replacing SCRATCH with VRMDATA

---

The changes described in this Appendix applies only to release 10.1 and beyond.

New users starting with release 10.1 or beyond do not have to deal with the fact that *VRMDATA* used to be *SCRATCH* in a previous releases. However, existing users might benefit from knowing what has changed so they can alter their RMDB files accordingly. Therefore, this Appendix is added to the User's Guide with the caveat that the contents therein are only applicable to pre-10.1 regression suites being migrated to the new terminology.

Verification Run Manager (VRM) maintains a directory within which the *Actions* that make up a regression suite are launched and various log files are maintained. In previous versions of VRM, this file was called *SCRATCH* by default and the command-line option to specify some other name for this directory was *-scratch*. Some users argued that the word *SCRATCH* denoted that the directory contained temporary data that could be deleted and regenerated with every regression run. In the very first versions of VRM, this was indeed true; the intention was that the user could easily delete this directory with every run and there is a command-line option (*-clean*) that supports this use model.

However, with the introduction of the GUI and stored history, the *SCRATCH* directory has become more of a repository of past results and history that the user would not want to delete with each run. For this reason, the name of the *SCRATCH* directory is changed to the *VRMDATA* directory to better reflect its current role in the Questa VRM environment.

<b>Terminology and Typography .....</b>	<b>601</b>
<b>Command-line Option and Default Value.....</b>	<b>602</b>
<b>Pre-defined Parameters.....</b>	<b>603</b>
<b>User-definable Procedures and TCL Utility Procedures.....</b>	<b>603</b>

## Terminology and Typography

The default name for the directory within which *vrun* launches *Actions* as part of a regression run is *VRMDATA*. When this directory is referred to in the VRM GUI or in *vrun* error messages, it is referred to as the *VRM Data* directory. This documentation refers to this directory using either *VRMDATA* directory or *VRM Data* directory. The second form (*VRM Data*) should always capitalize *Data* to distinguish between this specific directory and any random directory that may include data related to VRM. The term *VRMDATA* is used as a generic reference to the directory used by *vrun*, even if the user specifies some other name for the directory using the *-vrmdata* command-line option.

Internal to RMDB files and user-supplied TCL procedures, this directory is known as simply the “data directory” and internal references will refer to the directory by the name *DATADIR*. This is in keeping with the general pattern such as *RMDBDIR* for the value of the *-rmdb* command-line option and *VRUNDIR* for the value of the directory in which *vrun* was initially launched.

## Command-line Option and Default Value

The command-line option to set the name of the directory used by *vrun* to launch *Action* scripts changes from *-scratch* to *-vrldata*. The *-scratch* command-line option will continue to be supported for the foreseeable future but it will no longer be documented and should be considered deprecated.

If neither option is specified (*-scratch* or *-vrldata*), then *vrun* will use the name *VRMDATA* as the default directory name. The layout and content of this directory remain unaffected by this change.

A batch-mode *vrun* process will attempt to recognize the case where a legacy *SCRATCH* directory exists that would have been picked up by default by a pre-10.1 *vrun* process. The rules are as follows:

- If the *vrun* command line specifies a directory name/path using either the *-vrldata* command-line option or the (undocumented) *-scratch* command-line option, then that directory name is used.
- Otherwise, if *vrun* detects an existing *VRMDATA* directory under the current directory, then that directory is used (without checking for a *SCRATCH* directory).
- Otherwise, if *vrun* detects a *SCRATCH* directory, then it will issue an error message and terminate the regression run.
- Otherwise, if neither directory is detected, then *vrun* will create a new *VRMDATA* directory in the current directory and use that.

Note that the reason *vrun* issues an error when a *SCRATCH* directory exists but no *VRMDATA* directory exists is because, in certain circumstances (such as when the *-clean* command-line option is specified), it is possible that after the first regression run using release 10.1 or beyond, the *SCRATCH* directory will disappear and a *VRMDATA* directory will appear in its place. While this is not fatal, it could be enough of a surprise to an unsuspecting user that it was decided an error would be the safer choice. To get past the error, the user can either add a *-vrldata* command-line option to the *vrun* command, pointing to the legacy *SCRATCH* directory, or the user can simply rename the *SCRATCH* directory to *VRMDATA*. The name *SCRATCH* is no longer the default name when the *-vrldata* command-line option is not specified so the error is only there to remind the users that they may be relying on previous default behavior which since has been changed.

## Pre-defined Parameters

The pre-defined parameter used to access the path specified with the `-vrmdata` command-line option will be `DATADIR` (to match the other `DIR` parameters like `RMDBDIR` and `VRUNDIR`). The `SCRATCH` pre-defined parameter will continue to be supported for the foreseeable future, but it will no longer be documented and should be considered deprecated.

## User-definable Procedures and TCL Utility Procedures

The user-definable procedures such as `AnalyzePassFail`, `OkToMerge`, and others are passed the name of a `userdata` array containing various values relevant to the `Action` on whose behalf the procedure was called. One of the elements of this array in pre-10.1 releases was called `SCRATCH`. A new element by the name of `DATADIR` has been introduced and will contain the current value of the `-vrmdata` command-line option (or the default directory name, if the command-line option was not specified). The `SCRATCH` data element in the `userdata` array will continue to be supported for the foreseeable future, but it will no longer be documented and should be considered deprecated.

A new utility procedure is available to user-definable procedures and other user-supplied TCL procedures. This procedure is called `PathRelativeToDatadir` and will take a relative path (relative to the `VRMDATA` directory) and will return an absolute path referring to the same location. The existing `PathRelativeToScratch` will continue to be supported for the foreseeable future, but it will no longer be documented and should be considered deprecated.



# Appendix C

## RMDB Reference

---

The Run Manager Database (RMDB) file is an XML file that describes the topology of one or more regression test suites. You should model each regression suite as a tree, where non-leaf nodes represent groups of leaf nodes or other non-leaf nodes.

A well-formed XML file must include exactly one top-level element, which, in the case of an RMDB file, is the r mdb element. The r mdb element usually contains one or more runnable elements that describe the various nodes in this tree. The tree itself is defined by membership lists in each of the non-leaf nodes.

The Document Type Definition (DTD) file in the Questa release tree (*<install\_dir>vm\_src/rmdb.dtd*) is the definitive source of the child element and attribute tables and supersedes this document in the event of a conflict.

Some notes about the syntax include:

- Any given runnable may be a member of more than one group.
- Whitespace within XML elements is ignored, except in the parameter, command, and usertcl elements.
- Executable scripts are represented by Action elements (such as preScript, execScript, and postScript), which are embedded in the runnable element with which they are associated.

---

### Note

 This appendix assumes that you have familiarity with both XML syntax and terminology as well as with VRM terminology.

---

Element Definitions .....	606
---------------------------	-----

## Element Definitions

These are the elements, shown roughly in top-down order, that may appear in an RMDB file.

The element descriptions contain, where appropriate, the following tables:

- Allowed Child Elements table — Lists the child elements that may appear within that element.
- Supported Attributes table — Lists the attributes supported by that element.
  - The “Expanded” column indicates whether or not the attribute value is parameter-expanded.
  - The “Evaluated” column indicates whether or not the parameter-expanded value is evaluated as a TCL expression.
  - The “Default” column lists the value assumed if the attribute is missing or empty. If “(required)” is listed in the “Default” column, that means the attribute must be present and must have a non-empty value.

Some attributes must be XML “name tokens”, which means they cannot contain embedded spaces and cannot start with a digit or a hyphen. There are other restrictions defined in the XML standard but, in general, VRM can only handle names containing alphanumeric characters and/or underscores. Other characters, even when allowed by the XML syntax, may collide with the internal use of the values of the attributes.

<b>rmdb</b> .....	<b>607</b>
<b>runnable</b> .....	<b>608</b>
<b>parameters and parameter</b> .....	<b>614</b>
<b>members and member</b> .....	<b>616</b>
<b>localfile</b> .....	<b>616</b>
<b>preScript</b> .....	<b>618</b>
<b>execScript</b> .....	<b>619</b>
<b>postScript</b> .....	<b>621</b>
<b>mergeScript</b> .....	<b>623</b>
<b>tplanScript</b> .....	<b>624</b>
<b>trendScript</b> .....	<b>626</b>
<b>triageScript</b> .....	<b>627</b>
<b>method</b> .....	<b>629</b>
<b>command</b> .....	<b>632</b>
<b>userTcl</b> .....	<b>633</b>

## rmdb

The rmdb element is the top-level element of every RMDB file.

You may specify only one top-level element in an RMDB file. The order of elements within the rmdb element is not significant. Text content within the rmdb element is ignored.

**Table C-1. rmdb Element — Allowed Child Elements**

Elements	Description
method	Execution method definition
rmdb	Nested RMDB data (the extra rmdb element is transparent to VRM)
Runnable	Definition of a node in the regression tree
usertcl	Block of user-supplied TCL code
xi:include	Used to import an external RMDB file <sup>1</sup>

1. The RMDB parser supports nested files via the published XInclude standard. The xi:include element and the XML-standard namespace attributes are listed under the rmdb element but are not part of this document. Refer to the XInclude standard from W3C for details on XML file inclusion.

**Table C-2. rmdb Element — Supported Attributes**

Attributes	Expanded	Evaluated	Default	Description
created			(empty)	{<date>} Optional creation date in any format.
loadtcl			(empty)	{<usertcl_name> ...} Space- or comma-separated list of usertcl blocks to load on startup. Refer to the section “ <a href="#">Loading User-supplied TCL Code Using usertcl into VRM</a> ” for more information.
modified			(empty)	{<date>} Optional “last-modified” date in any format.
options			(empty)	{<option_list>} Command-line options to be added to the vrun command Refer to the section “ <a href="#">Embedding Options in the RMDB File</a> ” for more information.

**Table C-2. rmdb Element — Supported Attributes (cont.)**

Attributes	Expanded	Evaluated	Default	Description
toprunnables			(empty)	{<runnable_name> ...} Space- or comma-separated list of top-most runnable elements in regression tree Refer to the section “ <a href="#">Database Structure</a> ” for more information.
version			1.0	{<n>.<m>} Optional version string, for example 1.0 or 1.1. Refer to the section “ <a href="#">RMDB Versions</a> ” for more information.
xmlns:xi			(n/a)	XInclude namespace URI (should be http://www.w3.org/2001/XInclude)
xml:base			(n/a)	Allowed but not used by VRM (sometimes emitted by edit tools)

## Examples

```
<rmdb version="1.0" toprunnables="nightly">
  ...
</rmdb>
<rmdb version="1.0" loadtcl="default" options="-j 2">
  ...
</rmdb>
```

## runnable

The runnable element describes one node in the regression tree.

A runnable may be a group which includes other runnable nodes as members. A group runnable can define the following:

- a membership list.
- a [preScript](#), which runs prior to any Action scripts defined by member runnable nodes.
- a [postScript](#), which runs after all the Action scripts defined by member runnable nodes have completed.

A task runnable can define the following:

- an [execScript](#), which defines the main Action script to be run but cannot define a membership list; leaf node runnable element.

A group runnable may also define an execScript which may be inherited by any task runnables under that group, but the group runnable itself does not make use of the execScript.

A runnable can also have a base type, which can define parameters, Action scripts, a membership list, or methods that are inherited by other runnables through base inheritance. Refer to the section “[Inheritance, Overrides, and Search Order](#)” for more information.

Group and task runnables participate in the execution graph while base runnables do not. The order of elements within the runnable element is not significant. Text content within the runnable element is ignored.

**Table C-3. runnable Element — Allowed Child Elements**

Elements	Description
<a href="#">execScript</a>	Defines an Action script to be run
<a href="#">localfile</a>	Defines a file to be generated, copied, or linked prior to script execution
<a href="#">members and member</a>	Defines a list of member runnable elements (group runnable only)
<a href="#">mergeScript</a>	Defines an Action script to merge one or more leaf-level UCDB files into a merged UCDB file
<a href="#">method</a>	Defines an execution method to apply to all or a subset of Action scripts
<a href="#">parameters and parameter</a>	Defines a list of VRM parameters
<a href="#">postScript</a>	Defines an Action script to be run after all member runnable scripts have completed (group runnable only)
<a href="#">preScript</a>	Defines an Action script to be run prior to any member runnable scripts being launched (group runnable only)
<a href="#">trendScript</a>	Defines an Action script to perform trend analysis on a merged UCDB file
<a href="#">triageScript</a>	Defines an Action script to import messages to a Triage Database (TDB) file

**Table C-4. runnable Element — Supported Attributes**

Attributes	Expanded	Evaluated	Default	Description
base			(empty)	{<base_runnable_name> ... } Space- or comma-separated list of base runnable element names. Refer to the section “ <a href="#">Inheritance, Overrides, and Search Order</a> ” for more information.
foreach	Yes		(empty)	{<string> ... } Space- or comma-separated list of iteration tokens Refer to the section “ <a href="#">Controlling the Repeat Index (foreach Functionality)</a> ” for more information.
if	Yes	Yes	true	{<expression>} Where <expression> is a TCL expression. If the expression is false, runnable membership in parent group is ignored Refer to the section “ <a href="#">Selective Membership in a Group</a> ” for more information.
index			(empty)	The name of parameter created from iteration index
name			(required)	{<string>} Runnable name (must be a name token)
onerror	Yes		See note <sup>1</sup>	{abandon   continue} Policy for continuation after an error <sup>2</sup> . Refer to the section “ <a href="#">Failure Propagation</a> ” for more information.
repeat	Yes		1	{<integer>} Repeat count. Refer to the section “ <a href="#">Repeating a Task or a Group</a> ” for more information.

**Table C-4. runnable Element — Supported Attributes (cont.)**

<b>Attributes</b>	<b>Expanded</b>	<b>Evaluated</b>	<b>Default</b>	<b>Description</b>
sequential			no	{yes   no} If yes, members run sequentially, if no, members run concurrently Refer to the section “ <a href="#">Execution Model</a> ” for more information.
testlist				<file-path> Specifies a path, relative to the directory from which the RMDB is read, to a <a href="#">Testlist File</a> , which enables the generation of random tests.  If you use a glob character in the testlist argument and match more than one file, the one with the most recent modification time will be used. If no glob characters are included, the file matching the specified path will be used.  The testlist attribute overrides both the repeat and the foreach attributes on the same runnable element.  If the file cannot be found, an error will be reported and the RMDB expansion will fail.
type			(required)	{base   group   task} Type of runnable. Refer to the sections “ <a href="#">Runnables and the Regression Suite Hierarchy</a> ” and “ <a href="#">Runnable Types and Inheritance</a> ” for more information.
unless	Yes	Yes	false	{<expression>} Where <expression> is a TCL expression. If the expression is true, runnable membership in parent group is ignored Refer to the section “ <a href="#">Selective Membership in a Group</a> ” for more information.

1. The default value is abandon for sequential groups and continue for non-sequential groups.

2. The onerror attribute controls whether the failure of an Action affects the running of subsequent Actions. If the attribute is set to abandon, subsequent Actions that rely on the failed Action will be skipped. If the attribute is set to continue, the error will not affect subsequent Actions.

## Examples

```
<runnable name="compile" type="group" sequential="yes">
  ...
</runnable>
<runnable name="compile_ovm_pkg" type="task" if="(%DO_COMPILE_OVM%) == 1">
  ...
</runnable>
<runnable name="simulation" type="task" foreach="(%TESTCASE%)">
  ...
</runnable>
```

## Testlist File

The testlist consists of one or more test specifications lines, where each line expands into one or more Runnable instances according to the *repeat-count* specified. The test specification syntax is:

```
<testname> <repeat-count> [{ <seed> | <key>=<value> } ...]
```

Each test specification must appear on a separate line. Blank lines and comment lines (those whose first non-whitespace character is a pound sign (#)) are ignored. Newline characters preceded by a backslash (\) character are ignored. Whitespace within each test specification is also ignored except to delimit the tokens on the line.

- *testname* — (required) A string that specifies the name of the test. These tests support the %n and %s tokens in parameter overrides. This value is also used to populate the *testname* parameter.

---

### Note



Tests expanded with the VRM testlist format automatically populate the *testname* parameter to make it compatible with VRM test-centric mode. The automatically populated *testname* parameter is the default value, used only when the parameter is not defined elsewhere in the RMDB.

- *repeat-count* — (optional) An integer that specifies the number of times *testname* should be repeated. No entry indicates that there is only one entry for that test.
- *seed* — (optional) An integer that initializes the *seed* parameter of the generated Runnable instances. If there are more *seed* values than needed to satisfy the *repeat-count*, the excess values are ignored. If there are fewer *seed* values than needed to satisfy the *repeat-count*, each of the remaining Runnable instances will be provided a random *seed* value generated by the (potentially pre-seeded) TCL random number generator.
- <*key*>=<*value*> — (optional) A token pair used to override other parameters in all generated Runnable instances of the test specification line. Any pairs must not contain

any unquoted whitespace. Use double quotes for specifying values that contain embedded whitespace.

To further customize the names of the generated Runnable instances, you can use one or more of the following substitution tags in the *testname*.

- %n — this tag is replaced by the numeric index of the Runnable instance (which starts at 1 and increments by 1 for each instance)
- %s — this tag is replaced by the integer seed (whether read from the testlist file or generated internally by vrun). It is not recommended to use only the seed in the iteration string, since there is a small but non-zero chance that a random seed could be duplicated.

If either substitution tag is present in the *testlist* token, the specified format overrides the default format for the iteration string.

### Testlist File Example 1

If the runnable element looks like this:

```
<Runnable name="Simulate" type="task" testlist="mytests">
...
</Runnable>
```

and the "mytests" file contained the following line:

```
randtest 3 123 456 789
```

the resulting Runnable instance names would be:

```
Simulate~randtest_1
Simulate~randtest_2
Simulate~randtest_3
```

### Testlist File Example 2

If the runnable element looks like this:

```
<Runnable name="Simulate" type="task" testlist="mytests">
...
</Runnable>
```

and the "mytests" file contained the following line:

```
randtest_%s_%n 3 123 456 789
```

the resulting Runnable instance names would be:

```
Simulate~randtest_123_1
Simulate~randtest_456_2
Simulate~randtest_789_3
```

## parameters and parameter

The parameters element is a container for zero or more parameter elements.

The order of parameter elements within the parameters element is not significant. The parameters element has no attributes. Text content within the parameters element is ignored.

**Table C-5. parameters Element — Allowed Child Elements**

Elements	Description
parameter	Defines a single name-value pair

The parameter element defines a name-value pair which can be referenced from other parameter values, from Action scripts, or from certain element attributes. Script commands and certain attribute values may contain references to parameters. These references are resolved using the values supplied by parameter elements.

The parameter elements within the parameters element are processed in the order they appear.

The parameter element defines no child elements. The value of a parameter element is its text content. The text content is parsed in a way, such that you must escape characters that are reserved in the XML syntax.

**Table C-6. parameter Elements — Supported Attributes**

Attributes	Expanded	Evaluated	Default	Description
accept			(empty)	{<pattern>} Specifies the pattern of acceptable values for the parameter Refer to the section “ <a href="#">Selecting Values for Embedded Options</a> ” for more information.
ask			(empty)	{<prompt>} Enables interactive parameter-value prompting and supplies the text of the prompt Refer to the section “ <a href="#">Interaction with Parameter Prompts</a> ” for more information.
export			no	{yes   no} If yes, the parameter is exported to scripts as an environment variable Refer to the section “ <a href="#">Exporting Parameters to Action Scripts</a> ” for more information.

**Table C-6. parameter Elements — Supported Attributes (cont.)**

Attributes	Expanded	Evaluated	Default	Description
name			(required)	{<parameter_name_token>} Parameter name (must be a name token)
type			text	{file   tcl   text} Parameter type <sup>1</sup> . Refer to the section “ <a href="#">Including TCL Code in a Parameter Definition</a> ” and “ <a href="#">Referring to Source Files Via a Parameter</a> ” for more information.

1. If the parameter type is file, the value of the parameter is treated as being as a path relative to the directory from which the RMDB was read. If the parameter type is tcl, the value of the parameter is evaluated as a TCL statement before being used to expand a parameter reference.

## Predefined Parameters

Predefined parameters are those initialized to global paths or names relevant to the VRM algorithm. Refer to the section “[Predefined Parameters](#)” for a full listing.

The VRM convention is to capitalize those particular parameters to avoid namespace collisions with other user-defined parameters. Those parameters can be expanded in parameter-expanded attributes, however some are only meaningful in the context in which the parameter is defined.

## Special Named Parameters

You can use a special group of parameter names to enable additional VRM functionality. Refer to the section [Parameters Recognized/Defined by VRM](#) for a complete description of these parameters. It is recommended to avoid using the names of the parameters defined in [Table 14-3](#) if the additional functionality is not intended.

## Examples

```
<parameters>
    <parameter name="NUM_SIM" ask="Enter number of simulation repeats : "
        accept="num(1,500)">2</parameter>
    <parameter name="TESTCASE">fpu_test_patternset fpu_test_sequence_fair
        fpu_test_neg_sqr_sequence fpu_test_random_sequence
        fpu_test_simple_sanity</parameter>
    <parameter name="OVM_VERBOSITY_LEVEL">OVM_FULL</parameter>
    <parameter name="DPI_HEADER_FILE" type="file">(%DATADIR%)
        dpiheader.h</parameter>
    <parameter name="REF_MODEL_CPP">fpu_tlm_dpi_c.cpp</parameter>
</parameters>
```

## members and member

The members element is a container for zero or more member elements.

The order of member elements within the members element is only significant for runnable elements whose sequential attribute is set to yes (in which case, the order of the member elements is the order in which the runnables will be executed). The members element has no attributes. Text content within the members element is ignored.

**Table C-7. members Element — Allowed Child Elements**

Elements	Description
member	References a single member runnable

The member element references another runnable which becomes a member of the group runnable in which it is found. The member element has no attributes. The text content of each member attribute must match the name of a runnable element defined in the RMDB (after importing any nested RMDB files).

---

### Note

 Duplicate member elements within a members element are not currently supported. In the case of a sequential group, an error is reported at runtime. In the case of a non-sequential group, all but the first occurrence of a given member are ignored and a warning is reported. Refer to the section “[Restrictions on Group Membership](#)” for more information.

---

## Examples

```
<members>
  <member>compile</member>
  <member>sim_group</member>
  <member>vm</member>
</members>
```

## localfile

The localfile element specifies a file that is to be generated, copied, or symbolically linked into the working directory associated with the runnable element in which it is found.

Refer to the section “[Local File Generation](#)” for more information.

The order of command elements within the localfile element is significant. Text content within the localfile element is ignored.

**Table C-8. localfile Element — Allowed Child Elements**

Elements	Description
command	Defines a single line to be written to the local file.

**Table C-9. localfile Element — Supported Attributes**

<b>Attributes</b>	<b>Expanded</b>	<b>Evaluated</b>	<b>Default</b>	<b>Description</b>
append			no	{yes   no} If yes, the contents of the element will be appended to the file having the same target path, if it exists.
executable			no	{yes   no} If yes, the file is marked as executable after generation/copying.
expand			no	{yes   no} If yes, the command elements are parameter-expanded as the file is generated
name	Yes		1	{[<path>]<name>} Specifies the path/name of the file or link to be created, if different from the src attribute (relative to TASKDIR)
src	Yes			{[<path>]<name>} Specifies the path/name of the source file (relative to RMDBDIR)
type			copy	{copy   link} Specifies whether to copy or symbolically link the source file to the local directory

1. If the src attribute is missing or blank, the file is generated based on the command elements found in the localfile element and the type attribute is ignored. If a non-blank value is specified for the src attribute, any command elements in the localfile element are ignored. In the latter case, the type attribute determines whether the source file is copied to the local directory or just a symbolic link created.

## Examples

```
<localfile name="run.do" expand="yes">
  <command>run -all</command>
  <command>coverage save (%ucdbfile%)</command>
</localfile>
```

## preScript

The preScript element defines an Action script that is executed prior to executing the member runnable elements defined in a group runnable element.

The preScript element is valid only in runnable elements whose type attribute is set to group (or base runnables of such a group runnable). Text content within a preScript element is ignored.

**Table C-10. preScript Element — Allowed Child Elements**

Elements	Description
<a href="#">command</a>	Defines a single line to be written to the script file

**Table C-11. preScript Element — Supported Attributes**

Attributes	Expanded	Evaluated	Default	Description
after				{<list of arguments>} Specifies a list of arguments to append to the command launching the script (launch attribute).
file	Yes		(empty)	{<filename>} If specified, commands are read from a file rather than from command elements (relative to RMDBDIR)
launch			vsim	{exec   vsim   <interpreter_path>} Specifies how to launch the script (exec, vsim, or the path to another interpreter) Refer to the section “ <a href="#">Action and Wrapper Script Generation</a> ” for more information.
mintimeout	Yes		300	{<integer>} (in seconds) Minimum execution timeout for the script A value of 0 disables timeout monitoring for the Action. This value takes precedence over other global and local timeout values. Refer to the section “ <a href="#">Timeouts</a> ” for more information.

**Table C-11. preScript Element — Supported Attributes (cont.)**

Attributes	Expanded	Evaluated	Default	Description
role			The same as the element tag name.	{<action_element_type>} A string matching any of the Action script types: mergeScript, execScript, and so on.
timeoutmargin	Yes		20	{<integer>} Percentage of maximum execution time added for time-out computation.
ucdbfile	Yes		(empty)	{<pathname>} Path to the UCDB file generated by the script, if any. Refer to the section “ <a href="#">Pass/Fail Analysis</a> ” for more information.
usestderr			yes	{yes   no} If yes, content written to the standard error channel will cause the script to fail

## Examples

```
<preScript>
  <command>vlib work</command>
  <command>vlog -coverAll (%RMDBPATH%)/src/top.sv</command>
</preScript>
```

## execScript

The execScript element defines the Action script which is executed in association with a task or group runnable element.

The execScript element is valid only in runnable elements whose type attribute is set to task (or base runnables of such a task runnable). Text content within a execScript element is ignored.

**Table C-12. execScript Element — Allowed Child Elements**

Elements	Description
<a href="#">command</a>	Defines a single line to be written to the script file

**Table C-13. execScript Element — Supported Attributes**

Attributes	Expanded	Evaluated	Default	Description
after				{<list of arguments>} Specifies a list of arguments to append to the command launching the script (launch attribute).
file	Yes		(empty)	{<filename>} If specified, commands are read from a file rather than from command elements (relative to RMDBDIR)
launch			vsim	{exec   vsim   <interpreter_path>} Specifies how to launch the script (exec, vsim, or the path to another interpreter) Refer to the section “ <a href="#">Action and Wrapper Script Generation</a> ” for more information.
mintimeout	Yes		300	{<integer>} (in seconds) Minimum execution timeout for the script A value of 0 disables timeout monitoring for the Action. This value takes precedence over other global and local timeout values. Refer to the section “ <a href="#">Timeouts</a> ” for more information.
role			The same as the element tag name.	{<action_element_type>} A string matching any of the Action script types: mergeScript, execScript, and so on.
timeoutmargin	Yes		20	{<integer>} Percentage of maximum execution time added for time-out computation.
ucdbfile	Yes		(empty)	{<pathname>} Path to the UCDB file generated by the script, if any Refer to the section “ <a href="#">Pass/Fail Analysis</a> ” for more information.

**Table C-13. execScript Element — Supported Attributes (cont.)**

Attributes	Expanded	Evaluated	Default	Description
usestderr			yes	{yes   no} If yes, content written to the standard error channel will cause the script to fail

## Examples

```
<execScript launch="vsim">
  <command>vsim -c top (%vsimopts%)</command>
  <command>run -all</command>
  <command>coverage save (%ucdbfile%)</command>
</execScript>
```

## postScript

The postScript element defines an Action script which is executed after the member runnable elements defined in a group runnable element have completed.

The postScript element is valid only in runnable elements whose type attribute is set to group (or base runnables of such a group runnable). Text content within a postScript element is ignored.

**Table C-14. postScript Element — Allowed Child Elements**

Elements	Description
command	Defines a single line to be written to the script file

**Table C-15. postScript Element — Supported Attributes**

Attributes	Expanded	Evaluated	Default	Description
after				{<list of arguments>} Specifies a list of arguments to append to the command launching the script (launch attribute).
file	Yes		(empty)	{<pathname>} If specified, commands are read from a file rather than from command elements (relative to RMDBDIR)

**Table C-15. postScript Element — Supported Attributes (cont.)**

Attributes	Expanded	Evaluated	Default	Description
launch			vsim	{exec   vsim   <interpreter_path>} Specifies how to launch the script (exec, vsim, or the path to another interpreter) A value of 0 disables timeout monitoring for the Action. This value takes precedence over other global and local timeout values. Refer to the section “ <a href="#">Action and Wrapper Script Generation</a> ” for more information.
mintimeout	Yes		300	{<integer>} (in seconds) Minimum execution timeout for the script Refer to the section “ <a href="#">Timeouts</a> ” for more information.
role			The same as the element tag name.	{<action_element_type>} A string matching any of the Action script types: mergeScript, execScript, and so on.
timeoutmargin	Yes		20	{<integer>} Percentage of maximum execution time added for time-out computation.
ucdbfile	Yes		(empty)	{<pathname>} Path to the UCDB file generated by the script, if any Refer to the section “ <a href="#">Pass/Fail Analysis</a> ” for more information.
usestderr			yes	{yes   no} If yes, content written to the standard error channel will cause the script to fail

## Examples

```
<postScript>
    <command>vrun -vrmdata (%DATADIR%) -status -full -html
        -htmldir (%DATADIR%)/vrun</command>
</postScript>
```

## mergeScript

The mergeScript element overrides the default Action script which is executed by auto-merge in order to merge one or more leaf-level UCDBs into a common merged UCDB file.

Refer to the section “[Enabling and/or Disabling Coverage Merge Flows](#)” for more information.

Since VRM defines a global mergeScript by default, this element is only used when it becomes necessary to override the default auto-merge behavior. Text content within a mergeScript element is ignored.

**Table C-16. mergeScript Element — Allowed Child Elements**

Elements	Description
<a href="#">command</a>	Defines a single line to be written to the script file

**Table C-17. mergeScript Element — Supported Attributes**

Attributes	Expanded	Evaluated	Default	Description
after				{<list of arguments>} Specifies a list of arguments to append to the command launching the script (launch attribute).
inctimeout	Yes		120	{<integer>} (in seconds) Additional time added to the minimum execution timeout for each UCDB to be merged Refer to the section “ <a href="#">vrun Command</a> ” for more information.
file	Yes		(empty)	{<filename>} If specified, commands are read from a file rather than from command elements (relative to RMDBDIR) Refer to the section “ <a href="#">Actions and Commands</a> ” for more information.

**Table C-17. mergeScript Element — Supported Attributes (cont.)**

Attributes	Expanded	Evaluated	Default	Description
launch			vsim	{exec   vsim   <interpreter_path>} Specifies how to launch the script (exec, vsim, or the path to another interpreter) Refer to the section “ <a href="#">Action and Wrapper Script Generation</a> ” for more information.
mintimeout	Yes		300	{<integer>} (in seconds) Minimum execution timeout for the script A value of 0 disables timeout monitoring for the Action. This value takes precedence over other global and local timeout values. Refer to the section “ <a href="#">Timeouts</a> ” for more information.
role			The same as the element tag name.	{<action_element_type>} A string matching any of the Action script types: mergeScript, execScript, and so on.
timeoutmargin	Yes		20	{<integer>} Percentage of maximum execution time added for timeout computation.
usestderr			yes	{yes   no} If yes, content written to the standard error channel will cause the script to fail

## tplanScript

The tplanScript element overrides the default Action script which is executed by automatic testplan import.

Refer to the section “[Testplan Auto-import](#)” for more information.

**Table C-18. tplanScript Element — Allowed Child Elements**

Elements	Description
<a href="#">command</a>	Defines a single line to be written to the script file

**Table C-19. tplanScript Element — Supported Attributes**

Attributes	Expanded	Evaluated	Default	Description
after				{<list of arguments>} Specifies a list of arguments to append to the command launching the script (launch attribute).
file	Yes		(empty)	{<filename>} If specified, commands are read from a file rather than from command elements (relative to RMDBDIR)
launch			vsim	{exec   vsim   <interpreter_path>} Specifies how to launch the script (exec, vsim, or the path to another interpreter) Refer to the section “ <a href="#">Action and Wrapper Script Generation</a> ” for more information.
mintimeout	Yes		300	{<integer>} (in seconds) Minimum execution timeout for the script A value of 0 disables timeout monitoring for the Action. This value takes precedence over other global and local timeout values. Refer to the section “ <a href="#">Timeouts</a> ” for more information.
role			The same as the element tag name.	{<action_element_type>} A string matching any of the Action script types: mergeScript, execScript, and so on.

**Table C-19. tplanScript Element — Supported Attributes (cont.)**

Attributes	Expanded	Evaluated	Default	Description
timeoutmargin	Yes		20	{<integer>} Percentage of maximum execution time added for time-out computation.
usestderr			yes	{yes   no} If yes, content written to the standard error channel will cause the script to fail

## trendScript

The trendScript element overrides the default Action script which is executed by auto-trend in order to perform trending analysis of merged files from the auto-merge functionality.

Refer to the section “[Automated Trend Analysis](#)” for more information.

**Table C-20. trendScript Element — Allowed Child Elements**

Elements	Description
<a href="#">command</a>	Defines a single line to be written to the script file

**Table C-21. trendScript Element — Supported Attributes**

Attributes	Expanded	Evaluated	Default	Description
after				{<list of arguments>} Specifies a list of arguments to append to the command launching the script (launch attribute).
file	Yes		(empty)	{<filename>} If specified, commands are read from a file rather than from command elements (relative to RMDBDIR)
launch			vsim	{exec   vsim   <interpreter_path>} Specifies how to launch the script (exec, vsim, or the path to another interpreter) Refer to the section “ <a href="#">Action and Wrapper Script Generation</a> ” for more information.

**Table C-21.** trendScript Element — Supported Attributes (cont.)

Attributes	Expanded	Evaluated	Default	Description
mintimeout	Yes		300	{<integer>} (in seconds) Minimum execution timeout for the script  A value of 0 disables timeout monitoring for the Action. This value takes precedence over other global and local timeout values.  Refer to the section “ <a href="#">Timeouts</a> ” for more information.
role			The same as the element tag name.	{<action_element_type>} A string matching any of the Action script types: mergeScript, execScript, and so on.
timeoutmargin	Yes		20	{<integer>} Percentage of maximum execution time added for time-out computation.
usestderr			yes	{yes   no} If yes, content written to the standard error channel will cause the script to fail

## triageScript

The triageScript element overrides the default Action script which is executed by auto-triage in order to import the messages associated with the test stored in one or more UCDB files into a Triage Database (TDB) file.

Refer to the section “[Automated Results Analysis](#)” for more information.

Since VRM defines a global triageScript by default, this element is only used when it becomes necessary to override the default auto-triage behavior. Text content within a triageScript element is ignored.

**Table C-22.** triageScript Element — Allowed Child Elements

Elements	Description
<a href="#">command</a>	Defines a single line to be written to the script file

**Table C-23. triageScript Element — Supported Attributes**

Attributes	Expanded	Evaluated	Default	Description
after				{<list of arguments>} Specifies a list of arguments to append to the command launching the script (launch attribute).
inctimeout	Yes		120	{<integer>} (in seconds) Additional time added to the minimum execution timeout for each UCDB to be triaged Refer to the section “ <a href="#">vrun Command</a> ” for more information.
file	Yes		(empty)	{<filename>} If specified, commands are read from a file rather than from command elements (relative to RMDBDIR)
launch			vsim	{exec   vsim   <interpreter_path>} Specifies how to launch the script (exec, vsim, or the path to another interpreter) Refer to the section “ <a href="#">Action and Wrapper Script Generation</a> ” for more information.
mintimeout	Yes		300	{<integer>} (in seconds) Minimum execution timeout for the script A value of 0 disables timeout monitoring for the Action. This value takes precedence over other global and local timeout values. Refer to the section “ <a href="#">Timeouts</a> ” for more information.
role			The same as the element tag name.	{<action_element_type>} A string matching any of the Action script types: mergeScript, execScript, and so on.

**Table C-23. triageScript Element — Supported Attributes (cont.)**

Attributes	Expanded	Evaluated	Default	Description
timeoutmargin	Yes		20	{<integer>} Percentage of maximum execution time added for time-out computation.
usestderr			yes	{yes   no} If yes, content written to the standard error channel will cause the script to fail

## method

The method element provides a way to control the execution of one or more Action scripts.

There may be, at most, one command element within a method element. Text content within a method element is ignored. Each Action script is controlled by only one method element. Multiple method elements can be defined in a runnable element, the order of which, together with the method's conditional attributes, determines the method used to control an Action script. Methods also follow base and group inheritance.

**Table C-24. method Element — Allowed Child Elements**

Elements	Description
command	Defines a single line to be written to the script file

**Table C-25. method Element — Supported Attributes**

Attributes	Expanded	Evaluated	Default	Description
action	Yes		(empty)	{<action_element_name>} If specified, limits application of this method to a specific Action script type Refer to the section “ <a href="#">Conditional Execution Methods</a> ” for more information.

**Table C-25. method Element — Supported Attributes (cont.)**

Attributes	Expanded	Evaluated	Default	Description
base			(empty)	{<base_method_name> ... } Space- or comma-separated list of base method elements Refer to the section “ <a href="#">Methods and Base Inheritance</a> ” for more information.
context	Yes		(empty)	{<context_string>} If specified, limits application of this method to a specific context Refer to the section “ <a href="#">Conditional Execution Methods</a> ” for more information.
donedelay	Yes		0	{<integer>} Number of milliseconds delay to insert before processing the results of completed scripts.
gridtype			(empty)	{lsf   sge   uge   rtda   <user_defined>} Type of grid software being used. Refer to the section “ <a href="#">Controlling the Grid Submission Command</a> ” for more information.
if	Yes	Yes	true	{<expression>} Where <expression> is a TCL expression. If false, blocks application of this method
maxarray	Yes		(empty)	{<integer>} If specified, specifies maximum number of Action scripts to be clubbed into a single array job. When set to zero (0), the maximum is unlimited. Refer to the section “ <a href="#">Creating Grid Array Jobs</a> ” for more information.

**Table C-25. method Element — Supported Attributes (cont.)**

<b>Attributes</b>	<b>Expanded</b>	<b>Evaluated</b>	<b>Default</b>	<b>Description</b>
maxclubbing	Yes		(empty)	{<integer>} If specified, specifies maximum number of Action scripts to be clubbed into a single script, specifically a single job. When set to zero (0), the maximum is unlimited. Refer to the section “ <a href="#">Multi-Action Clubbing Fixed-list Mode</a> ” for more information.
maxrunning	Yes		(empty)	{<integer>} If specified, specifies maximum number of jobs that may be running concurrently. If you set maxrunning to a value “m” and maxclubbing to a value “n”, then the named queue can run a maximum of “m” jobs containing “n” actions. Refer to the section “ <a href="#">Named Execution Queues</a> ” for more information.
mintimeout			60	{<integer>} Minimum queuing timeout for scripts using this method A value of 0 disables timeout monitoring for the Action. This value takes precedence over other global and local timeout values. Refer to the section “ <a href="#">Methods and Timeouts</a> ” for more information.
name	Yes		(empty)	{<string>} Name of the method (if present, must be a name token)

**Table C-25. method Element — Supported Attributes (cont.)**

Attributes	Expanded	Evaluated	Default	Description
queue			(empty)	{<string>} Name of the internal queue to be used (if present, must be a name token) Refer to the section “ <a href="#">Named Execution Queues</a> ” for more information.
Runnable	Yes		(empty)	{<Runnable_name>} If specified, limits application of this method to Action scripts from a specific runnable element Refer to the section “ <a href="#">Conditional Execution Methods</a> ” for more information.
unless	Yes	Yes	false	{<expression>} Where <expression> is a TCL expression. If true, blocks application of this method
window	Yes		(empty)	{<string>} Settings for adjustable execution limit window Refer to the section “ <a href="#">Limiting the Execution Time of a Given Named Queue</a> ” for more information.

There are three method-specific conditional attributes: action, runnable and context; in addition to the if and unless also used for the conditional execution of the runnable elements. Refer to the section “[Conditional Execution Methods](#)” for more information.

## Examples

```
<method name="grid" gridtype="sge" mintimeout="600"
       if="{(%MODE:%) eq {grid}}">
    <command>qsub -V -cwd -b yes -e /dev/null -o /dev/null
              -N (%INSTANCE%) (%WRAPPER%)</command>
</method>
```

## command

The command element specifies one line of an Action script, localfile element, or execution method.

The command element has no attributes. The text content of the command element is parameter-expanded and copied to the Action script or, in the case of a method element, used to launch the Action script to which the method element applies.

## Examples

```
<command>vsim -c top (%vsimopts%)</command>
<command>run -all</command>
<command>coverage save (%ucdbfile%)</command>
```

## usertcl

The usertcl element specifies a block of TCL code which may be optionally loaded into VRM during initialization.

The text content of the usertcl element, when loaded, is sent to the TCL parser verbatim.

**Table C-26. usertcl Element — Supported Attributes**

Attributes	Expanded	Evaluated	Default	Description
base			(empty)	{<base_usertcl_name> ...} Space- or comma-separated list of base usertcl elements
file	Yes		(empty)	{<filename>} If specified, TCL code is read from the specified file instead of from the usertcl element (relative to RMDBDIR)
if	Yes	Yes	true	{<expression>} Where <expression> is a TCL expression. If false, blocks loading of this usertcl element
name			(require d)	{<string>} Name of the usertcl element (must be name token)
unless	Yes	Yes	false	{<expression>} Where <expression> is a TCL expression. If true, blocks loading of this usertcl element

## Examples

```
<usertcl name="mytcl">
    proc ActionStarted {userdata} {
        upvar $userdata data
        echo "ActionStarted (user): Action '$data(ACTION)' started"
    }
    proc ActionCompleted {userdata} {
        upvar $userdata data
        echo "ActionCompleted (user): Action '$data(ACTION)' completed"
    }
</usertcl>
```

# Appendix D

## RMDB Versions

---

This appendix provides a reference to the various versions of the Run Manager Database (RMDB).

You specify the RMDB version through the use of the **version** attribute of the **rmdb** document element of the top-most RMDB file.

```
<?xml version="1.0" ?>
<rmdb version="1.0"><runnable name="nightly" type="group">
  <parameters>
    <parameter name="mergefile">merge.ucdb</parameter>
  ...
</rmdb>
```

When you include an RMDB file in another RMDB file via the **xi:include** mechanism it is embedded at the point of the **xi:include** element, therefore the **rmdb** element of the included RMDB file is not the top-most **rmdb** element. Only the **version** attribute in the **document** element of the RMDB file that is actually passed to **vrun** is considered.

**Version Descriptions . . . . .** **636**

## Version Descriptions

---

This section provides a description of how the different versions vary from each other.

<b>RMDB 1.0</b> .....	<b>636</b>
<b>RMDB 1.1</b> .....	<b>636</b>

### RMDB 1.0

This version is the original RMDB format.

If you do not specify a version attribute in the **rmdb** element, it defaults to version 1.0.

### RMDB 1.1

This version introduces a change in the way execution method commands are parsed.

In version 1.0, if the **method** element had a **gridtype** attribute, its value was used to call a procedure by the name of **<gridtype>SubmitOptions**. The return value of that procedure was treated as a string and used to expand the **GRIDOPTS** parameter reference in the execution method command supplied as part of the **method** element. The expanded command was then parsed into tokens according to shell quoting rules and passed to the TCL **exec** command. The final tokenization stage made it difficult to properly quote command-line arguments originating from the **<gridtype>SubmitOptions** procedure.

In version 1.1, the execution **method** command supplied in the **method** element is first parsed into tokens in the RMDB API. The string form of the resulting TCL list then undergoes parameter expansion. The **<gridtype>SubmitOptions** procedure also supplies the **GRIDOPTS** value as a TCL list (rather than as a string). The TCL list resulting from parameter expansion is flattened and passed to the TCL **exec** command with no further parsing.

The specific change is that the **<gridtype>SubmitOptions** procedure must compose its return value as a list of pre-tokenized options rather than as a string.

The version change was made necessary because existing RMDB files in which the **<gridtype>SubmitOptions** procedure returns a string have no way to indicate where the command-line options should be tokenized and the new flow will treat that string as a single token option.

## Example

```
<usertcl name="...">
proc LsfSubmitOptions {userdata} {
    set retval [list]
    lappend retval -n 2
    lappend retval -do "embedded spaces"
    return $retval
}
</usertcl>
```

Note that the TCL **lappend** command accepts multiple strings for appending onto the list. Each TCL token after the list name, in this case “`retval`”, is appended as a separate item. If a single token in the final execution method command should include TCL-special characters, such as square brackets ( [ ] ), you should enclose that token in curly braces ( { } ) to escape the special characters, for example:

```
lappend retval -include {[square brackets]}
```

In this case, the square brackets are part of the token itself. The list format of the return value of the **<Grid>SubmitOptions** procedure will protect the bracketed token and it will be passed as an intact token to the TCL **exec** command.



# End-User License Agreement

The latest version of the End-User License Agreement is available on-line at:  
[www.mentor.com/eula](http://www.mentor.com/eula)

## IMPORTANT INFORMATION

**USE OF ALL SOFTWARE IS SUBJECT TO LICENSE RESTRICTIONS. CAREFULLY READ THIS LICENSE AGREEMENT BEFORE USING THE PRODUCTS. USE OF SOFTWARE INDICATES CUSTOMER'S COMPLETE AND UNCONDITIONAL ACCEPTANCE OF THE TERMS AND CONDITIONS SET FORTH IN THIS AGREEMENT. ANY ADDITIONAL OR DIFFERENT PURCHASE ORDER TERMS AND CONDITIONS SHALL NOT APPLY.**

## END-USER LICENSE AGREEMENT ("Agreement")

This is a legal agreement concerning the use of Software (as defined in Section 2) and hardware (collectively "Products") between the company acquiring the Products ("Customer"), and the Mentor Graphics entity that issued the corresponding quotation or, if no quotation was issued, the applicable local Mentor Graphics entity ("Mentor Graphics"). Except for license agreements related to the subject matter of this license agreement which are physically signed by Customer and an authorized representative of Mentor Graphics, this Agreement and the applicable quotation contain the parties' entire understanding relating to the subject matter and supersede all prior or contemporaneous agreements. If Customer does not agree to these terms and conditions, promptly return or, in the case of Software received electronically, certify destruction of Software and all accompanying items within five days after receipt of Software and receive a full refund of any license fee paid.

### 1. ORDERS, FEES AND PAYMENT.

- 1.1. To the extent Customer (or if agreed by Mentor Graphics, Customer's appointed third party buying agent) places and Mentor Graphics accepts purchase orders pursuant to this Agreement (each an "Order"), each Order will constitute a contract between Customer and Mentor Graphics, which shall be governed solely and exclusively by the terms and conditions of this Agreement, any applicable addenda and the applicable quotation, whether or not those documents are referenced on the Order. Any additional or conflicting terms and conditions appearing on an Order or presented in any electronic portal or automated order management system, whether or not required to be electronically accepted, will not be effective unless agreed in writing and physically signed by an authorized representative of Customer and Mentor Graphics.
- 1.2. Amounts invoiced will be paid, in the currency specified on the applicable invoice, within 30 days from the date of such invoice. Any past due invoices will be subject to the imposition of interest charges in the amount of one and one-half percent per month or the applicable legal rate currently in effect, whichever is lower. Prices do not include freight, insurance, customs duties, taxes or other similar charges, which Mentor Graphics will state separately in the applicable invoice. Unless timely provided with a valid certificate of exemption or other evidence that items are not taxable, Mentor Graphics will invoice Customer for all applicable taxes including, but not limited to, VAT, GST, sales tax, consumption tax and service tax. Customer will make all payments free and clear of, and without reduction for, any withholding or other taxes; any such taxes imposed on payments by Customer hereunder will be Customer's sole responsibility. If Customer appoints a third party to place purchase orders and/or make payments on Customer's behalf, Customer shall be liable for payment under Orders placed by such third party in the event of default.
- 1.3. All Products are delivered FCA factory (Incoterms 2010), freight prepaid and invoiced to Customer, except Software delivered electronically, which shall be deemed delivered when made available to Customer for download. Mentor Graphics retains a security interest in all Products delivered under this Agreement, to secure payment of the purchase price of such Products, and Customer agrees to sign any documents that Mentor Graphics determines to be necessary or convenient for use in filing or perfecting such security interest. Mentor Graphics' delivery of Software by electronic means is subject to Customer's provision of both a primary and an alternate e-mail address.

2. **GRANT OF LICENSE.** The software installed, downloaded, or otherwise acquired by Customer under this Agreement, including any updates, modifications, revisions, copies, documentation, setup files and design data ("Software") are copyrighted, trade secret and confidential information of Mentor Graphics or its licensors, who maintain exclusive title to all Software and retain all rights not expressly granted by this Agreement. Except for Software that is embeddable ("Embedded Software"), which is licensed pursuant to separate embedded software terms or an embedded software supplement, Mentor Graphics grants to Customer, subject to payment of applicable license fees, a nontransferable, nonexclusive license to use Software solely: (a) in machine-readable, object-code form (except as provided in Subsection 4.2); (b) for Customer's internal business purposes; (c) for the term of the license; and (d) on the computer hardware and at the site authorized by Mentor Graphics. A site is restricted to a one-half mile (800 meter) radius. Customer may have Software temporarily used by an employee for telecommuting purposes from locations other than a Customer office, such as the employee's residence, an airport or hotel, provided that such employee's primary place of employment is the site where the Software is authorized for use. Mentor Graphics' standard policies and programs, which vary depending on Software, license fees paid or services purchased, apply to the following: (a) relocation of Software; (b) use of Software, which may be limited, for example, to execution of a single session by a single user on the authorized hardware or for a restricted period of time (such limitations may be technically implemented through the use of authorization codes or similar devices); and (c) support services provided, including eligibility to receive telephone support, updates, modifications, and revisions. For the avoidance of doubt, if Customer provides any feedback or requests any change or enhancement to Products, whether in the course of receiving support or consulting services, evaluating Products, performing beta testing or otherwise, any inventions, product improvements, modifications or developments made by Mentor Graphics (at Mentor Graphics' sole discretion) will be the exclusive property of Mentor Graphics.

### **3. BETA CODE.**

- 3.1. Portions or all of certain Software may contain code for experimental testing and evaluation (which may be either alpha or beta, collectively “Beta Code”), which may not be used without Mentor Graphics’ explicit authorization. Upon Mentor Graphics’ authorization, Mentor Graphics grants to Customer a temporary, nontransferable, nonexclusive license for experimental use to test and evaluate the Beta Code without charge for a limited period of time specified by Mentor Graphics. Mentor Graphics may choose, at its sole discretion, not to release Beta Code commercially in any form.
- 3.2. If Mentor Graphics authorizes Customer to use the Beta Code, Customer agrees to evaluate and test the Beta Code under normal conditions as directed by Mentor Graphics. Customer will contact Mentor Graphics periodically during Customer’s use of the Beta Code to discuss any malfunctions or suggested improvements. Upon completion of Customer’s evaluation and testing, Customer will send to Mentor Graphics a written evaluation of the Beta Code, including its strengths, weaknesses and recommended improvements.
- 3.3. Customer agrees to maintain Beta Code in confidence and shall restrict access to the Beta Code, including the methods and concepts utilized therein, solely to those employees and Customer location(s) authorized by Mentor Graphics to perform beta testing. Customer agrees that any written evaluations and all inventions, product improvements, modifications or developments that Mentor Graphics conceived or made during or subsequent to this Agreement, including those based partly or wholly on Customer’s feedback, will be the exclusive property of Mentor Graphics. Mentor Graphics will have exclusive rights, title and interest in all such property. The provisions of this Subsection 3.3 shall survive termination of this Agreement.

### **4. RESTRICTIONS ON USE.**

- 4.1. Customer may copy Software only as reasonably necessary to support the authorized use. Each copy must include all notices and legends embedded in Software and affixed to its medium and container as received from Mentor Graphics. All copies shall remain the property of Mentor Graphics or its licensors. Except for Embedded Software that has been embedded in executable code form in Customer’s product(s), Customer shall maintain a record of the number and primary location of all copies of Software, including copies merged with other software, and shall make those records available to Mentor Graphics upon request. Customer shall not make Products available in any form to any person other than Customer’s employees and on-site contractors, excluding Mentor Graphics competitors, whose job performance requires access and who are under obligations of confidentiality. Customer shall take appropriate action to protect the confidentiality of Products and ensure that any person permitted access does not disclose or use Products except as permitted by this Agreement. Customer shall give Mentor Graphics written notice of any unauthorized disclosure or use of the Products as soon as Customer becomes aware of such unauthorized disclosure or use. Customer acknowledges that Software provided hereunder may contain source code which is proprietary and its confidentiality is of the highest importance and value to Mentor Graphics. Customer acknowledges that Mentor Graphics may be seriously harmed if such source code is disclosed in violation of this Agreement. Except as otherwise permitted for purposes of interoperability as specified by applicable and mandatory local law, Customer shall not reverse-assemble, disassemble, reverse-compile, or reverse-engineer any Product, or in any way derive any source code from Software that is not provided to Customer in source code form. Log files, data files, rule files and script files generated by or for the Software (collectively “Files”), including without limitation files containing Standard Verification Rule Format (“SVRF”) and Tcl Verification Format (“TVF”) which are Mentor Graphics’ trade secret and proprietary syntaxes for expressing process rules, constitute or include confidential information of Mentor Graphics. Customer may share Files with third parties, excluding Mentor Graphics competitors, provided that the confidentiality of such Files is protected by written agreement at least as well as Customer protects other information of a similar nature or importance, but in any case with at least reasonable care. Customer may use Files containing SVRF or TVF only with Mentor Graphics products. Under no circumstances shall Customer use Products or Files or allow their use for the purpose of developing, enhancing or marketing any product that is in any way competitive with Products, or disclose to any third party the results of, or information pertaining to, any benchmark.
  - 4.2. If any Software or portions thereof are provided in source code form, Customer will use the source code only to correct software errors and enhance or modify the Software for the authorized use, or as permitted for Embedded Software under separate embedded software terms or an embedded software supplement. Customer shall not disclose or permit disclosure of source code, in whole or in part, including any of its methods or concepts, to anyone except Customer’s employees or on-site contractors, excluding Mentor Graphics competitors, with a need to know. Customer shall not copy or compile source code in any manner except to support this authorized use.
  - 4.3. Customer agrees that it will not subject any Product to any open source software (“OSS”) license that conflicts with this Agreement or that does not otherwise apply to such Product.
  - 4.4. Customer may not assign this Agreement or the rights and duties under it, or relocate, sublicense, or otherwise transfer the Products, whether by operation of law or otherwise (“Attempted Transfer”), without Mentor Graphics’ prior written consent and payment of Mentor Graphics’ then-current applicable relocation and/or transfer fees. Any Attempted Transfer without Mentor Graphics’ prior written consent shall be a material breach of this Agreement and may, at Mentor Graphics’ option, result in the immediate termination of the Agreement and/or the licenses granted under this Agreement. The terms of this Agreement, including without limitation the licensing and assignment provisions, shall be binding upon Customer’s permitted successors in interest and assigns.
  - 4.5. The provisions of this Section 4 shall survive the termination of this Agreement.
5. **SUPPORT SERVICES.** To the extent Customer purchases support services, Mentor Graphics will provide Customer with updates and technical support for the Products, at the Customer site(s) for which support is purchased, in accordance with Mentor Graphics’ then current End-User Support Terms located at <http://supportnet.mentor.com/supportterms>.
  6. **OPEN SOURCE SOFTWARE.** Products may contain OSS or code distributed under a proprietary third party license agreement, to which additional rights or obligations (“Third Party Terms”) may apply. Please see the applicable Product documentation (including license files, header files, read-me files or source code) for details. In the event of conflict between the terms of this Agreement

(including any addenda) and the Third Party Terms, the Third Party Terms will control solely with respect to the OSS or third party code. The provisions of this Section 6 shall survive the termination of this Agreement.

## 7. LIMITED WARRANTY.

- 7.1. Mentor Graphics warrants that during the warranty period its standard, generally supported Products, when properly installed, will substantially conform to the functional specifications set forth in the applicable user manual. Mentor Graphics does not warrant that Products will meet Customer's requirements or that operation of Products will be uninterrupted or error free. The warranty period is 90 days starting on the 15th day after delivery or upon installation, whichever first occurs. Customer must notify Mentor Graphics in writing of any nonconformity within the warranty period. For the avoidance of doubt, this warranty applies only to the initial shipment of Software under an Order and does not renew or reset, for example, with the delivery of (a) Software updates or (b) authorization codes or alternate Software under a transaction involving Software re-mix. This warranty shall not be valid if Products have been subject to misuse, unauthorized modification, improper installation or Customer is not in compliance with this Agreement. MENTOR GRAPHICS' ENTIRE LIABILITY AND CUSTOMER'S EXCLUSIVE REMEDY SHALL BE, AT MENTOR GRAPHICS' OPTION, EITHER (A) REFUND OF THE PRICE PAID UPON RETURN OF THE PRODUCTS TO MENTOR GRAPHICS OR (B) MODIFICATION OR REPLACEMENT OF THE PRODUCTS THAT DO NOT MEET THIS LIMITED WARRANTY. MENTOR GRAPHICS MAKES NO WARRANTIES WITH RESPECT TO: (A) SERVICES; (B) PRODUCTS PROVIDED AT NO CHARGE; OR (C) BETA CODE; ALL OF WHICH ARE PROVIDED "AS IS."
  - 7.2. THE WARRANTIES SET FORTH IN THIS SECTION 7 ARE EXCLUSIVE. NEITHER MENTOR GRAPHICS NOR ITS LICENSORS MAKE ANY OTHER WARRANTIES EXPRESS, IMPLIED OR STATUTORY, WITH RESPECT TO PRODUCTS PROVIDED UNDER THIS AGREEMENT. MENTOR GRAPHICS AND ITS LICENSORS SPECIFICALLY DISCLAIM ALL IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NON-INFRINGEMENT OF INTELLECTUAL PROPERTY.
8. **LIMITATION OF LIABILITY.** TO THE EXTENT PERMITTED UNDER APPLICABLE LAW, IN NO EVENT SHALL MENTOR GRAPHICS OR ITS LICENSORS BE LIABLE FOR INDIRECT, SPECIAL, INCIDENTAL, OR CONSEQUENTIAL DAMAGES (INCLUDING LOST PROFITS OR SAVINGS) WHETHER BASED ON CONTRACT, TORT OR ANY OTHER LEGAL THEORY, EVEN IF MENTOR GRAPHICS OR ITS LICENSORS HAVE BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES. IN NO EVENT SHALL MENTOR GRAPHICS' OR ITS LICENSORS' LIABILITY UNDER THIS AGREEMENT EXCEED THE AMOUNT RECEIVED FROM CUSTOMER FOR THE HARDWARE, SOFTWARE LICENSE OR SERVICE GIVING RISE TO THE CLAIM. IN THE CASE WHERE NO AMOUNT WAS PAID, MENTOR GRAPHICS AND ITS LICENSORS SHALL HAVE NO LIABILITY FOR ANY DAMAGES WHATSOEVER. THE PROVISIONS OF THIS SECTION 8 SHALL SURVIVE THE TERMINATION OF THIS AGREEMENT.

## 9. THIRD PARTY CLAIMS.

- 9.1. Customer acknowledges that Mentor Graphics has no control over the testing of Customer's products, or the specific applications and use of Products. Mentor Graphics and its licensors shall not be liable for any claim or demand made against Customer by any third party, except to the extent such claim is covered under Section 10.
- 9.2. In the event that a third party makes a claim against Mentor Graphics arising out of the use of Customer's products, Mentor Graphics will give Customer prompt notice of such claim. At Customer's option and expense, Customer may take sole control of the defense and any settlement of such claim. Customer WILL reimburse and hold harmless Mentor Graphics for any LIABILITY, damages, settlement amounts, costs and expenses, including reasonable attorney's fees, incurred by or awarded against Mentor Graphics or its licensors in connection with such claims.
- 9.3. The provisions of this Section 9 shall survive any expiration or termination of this Agreement.

## 10. INFRINGEMENT.

- 10.1. Mentor Graphics will defend or settle, at its option and expense, any action brought against Customer in the United States, Canada, Japan, or member state of the European Union which alleges that any standard, generally supported Product acquired by Customer hereunder infringes a patent or copyright or misappropriates a trade secret in such jurisdiction. Mentor Graphics will pay costs and damages finally awarded against Customer that are attributable to such action. Customer understands and agrees that as conditions to Mentor Graphics' obligations under this section Customer must: (a) notify Mentor Graphics promptly in writing of the action; (b) provide Mentor Graphics all reasonable information and assistance to settle or defend the action; and (c) grant Mentor Graphics sole authority and control of the defense or settlement of the action.
- 10.2. If a claim is made under Subsection 10.1 Mentor Graphics may, at its option and expense: (a) replace or modify the Product so that it becomes noninfringing; (b) procure for Customer the right to continue using the Product; or (c) require the return of the Product and refund to Customer any purchase price or license fee paid, less a reasonable allowance for use.
- 10.3. Mentor Graphics has no liability to Customer if the action is based upon: (a) the combination of Software or hardware with any product not furnished by Mentor Graphics; (b) the modification of the Product other than by Mentor Graphics; (c) the use of other than a current unaltered release of Software; (d) the use of the Product as part of an infringing process; (e) a product that Customer makes, uses, or sells; (f) any Beta Code or Product provided at no charge; (g) any software provided by Mentor Graphics' licensors who do not provide such indemnification to Mentor Graphics' customers; (h) OSS, except to the extent that the infringement is directly caused by Mentor Graphics' modifications to such OSS; or (i) infringement by Customer that is deemed willful. In the case of (i), Customer shall reimburse Mentor Graphics for its reasonable attorney fees and other costs related to the action.
- 10.4. THIS SECTION 10 IS SUBJECT TO SECTION 8 ABOVE AND STATES THE ENTIRE LIABILITY OF MENTOR GRAPHICS AND ITS LICENSORS, AND CUSTOMER'S SOLE AND EXCLUSIVE REMEDY, FOR DEFENSE,

SETTLEMENT AND DAMAGES, WITH RESPECT TO ANY ALLEGED PATENT OR COPYRIGHT INFRINGEMENT OR TRADE SECRET MISAPPROPRIATION BY ANY PRODUCT PROVIDED UNDER THIS AGREEMENT.

**11. TERMINATION AND EFFECT OF TERMINATION.**

- 11.1. If a Software license was provided for limited term use, such license will automatically terminate at the end of the authorized term. Mentor Graphics may terminate this Agreement and/or any license granted under this Agreement immediately upon written notice if Customer: (a) exceeds the scope of the license or otherwise fails to comply with the licensing or confidentiality provisions of this Agreement, or (b) becomes insolvent, files a bankruptcy petition, institutes proceedings for liquidation or winding up or enters into an agreement to assign its assets for the benefit of creditors. For any other material breach of any provision of this Agreement, Mentor Graphics may terminate this Agreement and/or any license granted under this Agreement upon 30 days written notice if Customer fails to cure the breach within the 30 day notice period. Termination of this Agreement or any license granted hereunder will not affect Customer's obligation to pay for Products shipped or licenses granted prior to the termination, which amounts shall be payable immediately upon the date of termination.
- 11.2. Upon termination of this Agreement, the rights and obligations of the parties shall cease except as expressly set forth in this Agreement. Upon termination of this Agreement and/or any license granted under this Agreement, Customer shall ensure that all use of the affected Products ceases, and shall return hardware and either return to Mentor Graphics or destroy Software in Customer's possession, including all copies and documentation, and certify in writing to Mentor Graphics within ten business days of the termination date that Customer no longer possesses any of the affected Products or copies of Software in any form.
12. **EXPORT.** The Products provided hereunder are subject to regulation by local laws and European Union ("E.U.") and United States ("U.S.") government agencies, which prohibit export, re-export or diversion of certain products, information about the products, and direct or indirect products thereof, to certain countries and certain persons. Customer agrees that it will not export or re-export Products in any manner without first obtaining all necessary approval from appropriate local, E.U. and U.S. government agencies. If Customer wishes to disclose any information to Mentor Graphics that is subject to any E.U., U.S. or other applicable export restrictions, including without limitation the U.S. International Traffic in Arms Regulations (ITAR) or special controls under the Export Administration Regulations (EAR), Customer will notify Mentor Graphics personnel, in advance of each instance of disclosure, that such information is subject to such export restrictions.
13. **U.S. GOVERNMENT LICENSE RIGHTS.** Software was developed entirely at private expense. The parties agree that all Software is commercial computer software within the meaning of the applicable acquisition regulations. Accordingly, pursuant to U.S. FAR 48 CFR 12.212 and DFAR 48 CFR 227.7202, use, duplication and disclosure of the Software by or for the U.S. government or a U.S. government subcontractor is subject solely to the terms and conditions set forth in this Agreement, which shall supersede any conflicting terms or conditions in any government order document, except for provisions which are contrary to applicable mandatory federal laws.
14. **THIRD PARTY BENEFICIARY.** Mentor Graphics Corporation, Mentor Graphics (Ireland) Limited, Microsoft Corporation and other licensors may be third party beneficiaries of this Agreement with the right to enforce the obligations set forth herein.
15. **REVIEW OF LICENSE USAGE.** Customer will monitor the access to and use of Software. With prior written notice and during Customer's normal business hours, Mentor Graphics may engage an internationally recognized accounting firm to review Customer's software monitoring system and records deemed relevant by the internationally recognized accounting firm to confirm Customer's compliance with the terms of this Agreement or U.S. or other local export laws. Such review may include FlexNet (or successor product) report log files that Customer shall capture and provide at Mentor Graphics' request. Customer shall make records available in electronic format and shall fully cooperate with data gathering to support the license review. Mentor Graphics shall bear the expense of any such review unless a material non-compliance is revealed. Mentor Graphics shall treat as confidential information all information gained as a result of any request or review and shall only use or disclose such information as required by law or to enforce its rights under this Agreement. The provisions of this Section 15 shall survive the termination of this Agreement.
16. **CONTROLLING LAW, JURISDICTION AND DISPUTE RESOLUTION.** The owners of certain Mentor Graphics intellectual property licensed under this Agreement are located in Ireland and the U.S. To promote consistency around the world, disputes shall be resolved as follows: excluding conflict of laws rules, this Agreement shall be governed by and construed under the laws of the State of Oregon, U.S., if Customer is located in North or South America, and the laws of Ireland if Customer is located outside of North or South America or Japan, and the laws of Japan if Customer is located in Japan. All disputes arising out of or in relation to this Agreement shall be submitted to the exclusive jurisdiction of the courts of Portland, Oregon when the laws of Oregon apply, or Dublin, Ireland when the laws of Ireland apply, or the Tokyo District Court when the laws of Japan apply. Notwithstanding the foregoing, all disputes in Asia (excluding Japan) arising out of or in relation to this Agreement shall be resolved by arbitration in Singapore before a single arbitrator to be appointed by the chairman of the Singapore International Arbitration Centre ("SIAC") to be conducted in the English language, in accordance with the Arbitration Rules of the SIAC in effect at the time of the dispute, which rules are deemed to be incorporated by reference in this section. Nothing in this section shall restrict Mentor Graphics' right to bring an action (including for example a motion for injunctive relief) against Customer in the jurisdiction where Customer's place of business is located. The United Nations Convention on Contracts for the International Sale of Goods does not apply to this Agreement.
17. **SEVERABILITY.** If any provision of this Agreement is held by a court of competent jurisdiction to be void, invalid, unenforceable or illegal, such provision shall be severed from this Agreement and the remaining provisions will remain in full force and effect.
18. **MISCELLANEOUS.** This Agreement contains the parties' entire understanding relating to its subject matter and supersedes all prior or contemporaneous agreements. Any translation of this Agreement is provided to comply with local legal requirements only. In the event of a dispute between the English and any non-English versions, the English version of this Agreement shall govern to the extent not prohibited by local law in the applicable jurisdiction. This Agreement may only be modified in writing, signed by an authorized representative of each party. Waiver of terms or excuse of breach must be in writing and shall not constitute subsequent consent, waiver or excuse.