cadence®

Virtuoso ADE Verifier User Guide

Product Version IC23.1
November 2023

© 2023 Cadence Design Systems, Inc. All rights reserved.
Printed in the United States of America.

Cadence Design Systems, Inc. (Cadence), 2655 Seely Ave., San Jose, CA 95134, USA.

Open SystemC, Open SystemC Initiative, OSCI, SystemC, and SystemC Initiative are trademarks or registered trademarks of Open SystemC Initiative, Inc. in the United States and other countries and are used with permission.

Trademarks: Trademarks and service marks of Cadence Design Systems, Inc. contained in this document are attributed to Cadence with the appropriate symbol. For queries regarding Cadence's trademarks, contact the corporate legal department at the address shown above or call 800.862.4522. All other trademarks are the property of their respective holders.

Restricted Permission: This publication is protected by copyright law and international treaties and contains trade secrets and proprietary information owned by Cadence. Unauthorized reproduction or distribution of this publication, or any portion of it, may result in civil and criminal penalties. Except as specified in this permission statement, this publication may not be copied, reproduced, modified, published, uploaded, posted, transmitted, or distributed in any way, without prior written permission from Cadence. Unless otherwise agreed to by Cadence in writing, this statement grants Cadence customers permission to print one (1) hard copy of this publication subject to the following conditions:

1. The publication may be used only in accordance with a written agreement between Cadence and its customer.
2. The publication may not be modified in any way.
3. Any authorized copy of the publication or portion thereof must include all original copyright, trademark, and other proprietary notices and this permission statement.
4. The information contained in this document cannot be used in the development of like products or software, whether for internal or external use, and shall not be used for the benefit of any other party, whether or not for consideration.

Patents: The Cadence Products covered in this manual are protected by U.S. Patents

5,790,436; 5,812,431; 5,859,785; 5,949,992; 6,493,849; 6,278,964; 6,300,765; 6,304,097; 6,414,498; 6,560,755; 6,618,837; 6,693,439; 6,826,736; 6,851,097; 6,711,725; 6,832,358; 6,874,133; 6,918,102; 6,954,908; 6,957,400; 7,003,745; 7,003,749.

Disclaimer: Information in this publication is subject to change without notice and does not represent a commitment on the part of Cadence. Except as may be explicitly set forth in such agreement, Cadence does not make, and expressly disclaims, any representations or warranties as to the completeness, accuracy or usefulness of the information contained in this document. Cadence does not warrant that use of such information will not infringe any third party rights, nor does Cadence assume any liability for damages or costs of any kind that may result from use of such information.

Restricted Rights: Use, duplication, or disclosure by the Government is subject to restrictions as set forth in FAR52.227-14 and DFAR252.227-7013 et seq. or its successor.

Contents

1
Introduction to Virtuoso ADE Verifier 17
Key Elements of Virtuoso ADE Verifier 19
Requirements 20
Implementations 20
Verification Status and Reports 21
ADE Verifier in Design Verification Flows 21
ADE Verifier in a Top-Down Verification Flow 23
ADE Verifier in a Bottom-Up Verification Flow 24
Prerequisites to Using ADE Verifier 25
Launching Verifier 26
Verifier Cellviews and Sessions 28
Graphical User Interface of ADE Verifier 29
Tabs in ADE Verifier 30
Main Toolbar in ADE Verifier 31
File 31
Simulation 32
Edit 33
Move 34
Filter 34
Workspaces 35
Progress Bar 35
Assistants in ADE Verifier 36
Coverage Summary Assistant 36
Implementations Editor Assistant 37
Information Assistant 38
Requirements Editor Assistant 40
Setup Library Assistant 40
Snapshots Editor Assistant 41
Menu Bar in ADE Verifier 43
Setting Virtuoso ADE Verifier Preferences 45

© 2023 All Rights Reserved.

Logs and Automatic Backups in Virtuoso ADE Verifier . . . 46
Global Bindkeys in Virtuoso ADE Verifier . . . 47
Bindkeys for View Navigation and Interaction with Item Contents . . . 49

2
Requirements in Virtuoso ADE Verifier . . . 51

Details of a Requirement . . . 52
Shortcut Menu Options in the Requirements and Implementations Pane . . . 59
Adding Requirements . . . 61
Adding a Requirement Manually . . . 61
Import Requirements from External Sources . . . 63
Importing Requirements from File . . . 64
Important Points to Note when Importing Excel Files . . . 68
Importing Requirements from a Verifier Cellview . . . 69
Creating Requirements from Implementations . . . 70
Simulation Runs Using Auto-Synced Requirements . . . 72
Editing Requirements in the Requirements Hierarchy . . . 75
Editing Requirements Using the Requirements Editor Assistant . . . 76
Copying Requirements in the Hierarchy . . . 78
Copy Requirements from File . . . 79
Managing Imported Files . . . 81
Setting Up Requirement Owners . . . 82
Organizing the Requirements Hierarchy . . . 83
Reassigning IDs Hierarchically . . . 84
Exporting Requirements . . . 84
Deleting Requirements . . . 87
Filters in the Requirements Hierarchy . . . 88
Filtering Requirements Based on Text Criteria . . . 88
Filtering Requirements Based on Status . . . 89
Custom Fields . . . 90
Preparing ADE Verifier to Use Custom Fields . . . 91
Specifying Values for Custom Fields . . . 92
Description Editor for Requirements . . . 95

© 2023 All Rights Reserved.

3
Composite Requirements 101
Adding Composite Requirements 102
Overwriting Properties in Composite Requirements 104
Overwriting Mappings in Composite Requirements 105
Slicing Results According to the Assigned Verification Spaces 106
Exporting Composite Requirements 108

4
Implementations and Mappings in Virtuoso ADE Verifier . . 111
Details of an Implementation 112
Disabled Runs, Reliability Analyses, Tests, or Outputs for Implementation Cellviews . 118
Modification of the Enabled or Disabled State 120
Adding Implementations 122
Adding an Implementation Manually 123
Adding Multiple Implementations 123
Implementations with Wave Specifications 124
Implementations with ADE Assembler Run Plans 127
Customizing Simulations with ADE Assembler Cellviews 131
Creating Outputs in Implementations Using Requirements 131
Editing Outputs in Implementations 133
Importing Implementations from Another ADE Verifier Cellview 134
Implementation Run Modes 136
Mappings between Implementations and Requirements 137
Mapping Implementations to Requirements Using the Shortcut Menu 139
Mapping Implementations to Requirements Using Bindkeys 140
Mapping Implementations to Requirements Using the Mapping Mode 140
Managing Mappings 142
Exporting Mappings 143
Importing Mappings 143
Opening Implementations 144
Updates in Implementations 145
Restarting Implementation Jobs 146
Modifying Implementations 147

© 2023 All Rights Reserved.

Overriding Specifications 149
Deleting Implementations 151
Filters in the Implementation List 151
Filtering Implementations Based on Text Criteria 151
Filtering Implementations Based on Status 153
Mapped and MappedHier Properties 154
Typical Value Setup 157
Setting Up the Typical Value for Implementations 158
Project Directory Setup 161
Opening the Project Directory Setup 162
Adding and Removing a Project Directory 164
Modifying and Saving a Project Directory 165
Histories from the Project Directory Setup 166
Custom Implementations 167
Creating a Custom Implementation 168
Edit Custom Implementations 170
Editing a Single Custom Implementation 171
Editing Multiple Custom Implementations 172
Editing Output Expressions in Custom Implementations 174
Results of Modifying Expressions in Custom Implementations 177
Using Aliases in Custom Implementations 177

5
Implementation Runs in ADE Verifier 181
Implementation Simulation Flow 183
Setting Pre-run Scripts 183
Setting Up Run Preferences 186
Enabling or Disabling Implementation Runs from Verifier 187
Implementation Sets 188
Creating and Deleting Implementation Sets 191
Rearranging Implementation Sets 192
Simulation of Implementations 193
Running and Stopping Simulations 195
Simulation Status of Implementations 196
Report Identical Histories 198

© 2023 All Rights Reserved.

Simulation of Implementations Externally . . . 198
Running and Stopping Implementations in Batch Mode . . . 199
Setting Up Pre and Post Simulation Operations in Batch Scripts . . . 200
Distributing Batch Simulation Runs . . . 203
Simulation Runs based on Priority and Estimated Run Time . . . 204
Environment Setup to Run Simulations in ADE Verifier . . . 208
Creating a CSV File for Defining Environment Variables . . . 210
Adding and Removing Environment Variables . . . 211
Modifying Environment Variables . . . 212
Saving and Importing Pre-Defined Environment Variables from a CSV File . . . 212
Exporting Environment Variables to a CSV File . . . 213
Setting the Environment Variable Override Value . . . 214
Simulation Run Results . . . 215
Viewing Run Results Using the Information Assistant . . . 215
Results of Failed Outputs or Tests . . . 216
Viewing Results from an Incomplete History . . . 218
Viewing Job Policies for Implementations . . . 220
Viewing Simulation Logs . . . 221
Batch Scripts in ADE Verifier . . . 222
Generating a Batch Script . . . 223
Reviewing and Refining a Batch Script . . . 223
Running and Stopping a Batch Script . . . 226
Scheduling Runs . . . 227

6
Managing Verification Results . . . 229
Factors that Impact Verification Status . . . 230
Location of Run Summary Data . . . 232
Evaluation of Metric Prefix . . . 232
Reviewing the Current Overall Verification Status . . . 234
Overall Progress, Coverage, and Overall Coverage . . . 236
Verification Status of Requirements . . . 238
Viewing Verification Results in the Information Assistant and Requirements Editor . . . 243
Verification Status Indicators . . . 245
Requirements Signoff in Virtuoso ADE Verifier . . . 245

© 2023 All Rights Reserved.

Signing off Requirements 246
Updating the Completion Progress of Requirements Manually 248
Viewing Plotted Results 249
Detailed Verification Report 250
HTML Reports in Basic Format 250
Viewing the HTML Report in Basic Format 251
HTML Reports in Interactive Format 254
Viewing HTML Reports in Interactive Format 255
Viewing Additional Columns in the Verification Report 256
Customization of HTML Reports in Interactive Format 258
Customizing the Interactive HTML Report Template 259
Examples of Interactive HTML Report Customization 260

7
Verifying the Design Against the Specified Setup 265
Setup Library Assistant Workflow 267
Opening the Setup Library Assistant 269
Graphical User Interface of Setup Library Assistant 270
Setup Library Assistant Toolbar 270
Creating or Loading a Setup Library Cellview 271
Modification of the Setup Library Cellview 273
Adding New Sweep Setups 274
Adding New Corner Setups 275
Adding New Simulation Setups 278
Adding New Reliability Setups 281
Importing Sweeps and Corners in the Setup Library 284
Exporting and Importing SDB Setups 285
Deleting a Setup Library Cellview 287
Adding or Deleting Verification Spaces 288
Parametric Sets in Setup Library Assistant 290
Adding or Removing a Verification Space for Each Block 290
Running Simulations with Verification Spaces 292
Updating Implementation Cellviews with Verification Spaces 293
Coverage Calculation in Virtuoso ADE Verifier 296
Calculating the Coverage Percentage 297

© 2023 All Rights Reserved.

Calculation of Coverage Percentage for Statistical Outputs 299
Viewing the Coverage Summary 300
Exporting the Coverage Report 302
Setting ADE Verifier Preferences for Coverage 304

8
Snapshots in Virtuoso ADE Verifier 305
Snapshots User Interface in ADE Verifier 307
Snapshots Toolbar 307
Snapshots Editor 308
Snapshots Workspace 310
Snapshots Tab 311
Snapshot Management in ADE Verifier 313
Adding or Removing a Snapshot 313
Renaming a Snapshot 314
Setting a Reference Snapshot 314
Adding Comments to a Snapshot 314
Changing the Order of Snapshots 314
Restoring a Snapshot 315
Exporting Snapshots 315
Snapshot Comparison 317
Property Filters in Snapshots 319
Application of Tolerance Values in Snapshots 325
Modification of the Active Setup in the Snapshots Workspace 327
Comparison of Design and Simulation Setup Differences between Snapshots 328
Prerequisites for Snapshot Comparison 328
Comparing Design and Simulation Setup Differences 328
Ignoring Elements or their Order when Comparing Implementations from Snapshots . 330
Exporting Snapshot Comparison Differences 331

9
External References in Virtuoso ADE Verifier 333
Flow for Referencing External Cellviews 334
Importing or Accessing Cellviews as References 335

© 2023 All Rights Reserved.

Hierarchy for Referenced Cellviews . . . 338
Reassignment of Requirements . . . 339
Editing or Updating Broken External References . . . 340
Multiple Hierarchy Levels in Multi-User Verification Projects . . . 340
Synchronization Between the Master and Referenced Cellviews . . . 341
Updating Referenced Cellviews . . . 343
Simulation Runs on Referenced Cellviews . . . 343

10
Verification in ADE Verifier with Reliability . . . 345

Reliability Hierarchy in ADE Verifier . . . 346
Reliability Flow in ADE Verifier . . . 348
Reliability Outputs in ADE Verifier . . . 349
Reliability Outputs in Different Run Modes . . . 351
Validating Simulation Results for Reliability Outputs . . . 355
Viewing Simulation Logs for Reliability Outputs . . . 358

11
Mixed-Signal Design Verification . . . 361

ADE Verifier-vManager Flow . . . 362
ADE Verifier Architecture . . . 363
vManager Connection Settings in ADE Verifier . . . 364
Accessing the vManager Toolbar . . . 365
Prerequisites for Setting Up a vManager Connection in ADE Verifier . . . 366
Setting Up the vManager Connection . . . 366
Authentication for the vManager Server . . . 367
Download of vPlan Requirements on Initial Merge . . . 368
Uploading a Verification Plan to vManager . . . 369
Downloading a vPlan from vManager . . . 370
Editing the Verification Plan and vPlan Concurrently . . . 370
Verification Plans in ADE Verifier and vManager Plans . . . 371
Requirement IDs in an ADE Verifier-vManager Flow . . . 371
Posting Verifier Results to the vManager Server . . . 372
Creating a Batch and VSIF Script . . . 373
List of Mappings between Verifier Requirement Fields and vManager Attributes . . . 373

© 2023 All Rights Reserved.

ADE Verifier Results in vManager . . . 374
vManager Attributes - Domain and Scope . . . 376
Domain . . . 376
Scope . . . 377
Connecting Multiple Verifier Views to a vPlan . . . 378
Review of the Verification Results in vManager . . . 380
Referenced Cellviews in an ADE Verifier-vManager Session . . . 382
Connection between an ADE Verifier Cellview and a vManager Plan with External References . . . 383
Connection between an ADE Verifier Cellview with External References and a vManager Plan . . . 384

A
Virtuoso ADE Verifier Forms . . . 387

Add Implementation Cellviews Form . . . 389
Add Implementation Form . . . 390
Add Implementation Set Form . . . 391
Change Referenced Cellview Form . . . 392
Choose Verifier Cellview Form . . . 393
Edit Typical Value Setup Form . . . 394
Export Mappings and Results for Requirements in Imported Files Form . . . 395
Export Mappings Form . . . 397
Export Requirements to CSV File Form . . . 398
Export Requirements to Excel File Form . . . 399
Import Form . . . 400
Import from SDB File Form . . . 404
Import Verifier Cellview Form . . . 405
Manage Imported Files Form . . . 406
Preferences Form . . . 408
General . . . 409
Planning . . . 411
Run . . . 413
Distribution . . . 416
Reporting . . . 419
Save As Form . . . 422
Set Requirement Owners Form . . . 423

© 2023 All Rights Reserved.

Sign Off Form . . . 424
Typical Value Setup Form . . . 426
vManager Setup Form . . . 427

B
Verifier Environment Variables . . . 429

Environment Variables to Set Preferences . . . 430
allowEditingDisabledState . . . 431
appendDate . . . 432
appendTime . . . 433
autoCreateDataSheet . . . 434
autocreatereports . . . 435
autoUpdateAll . . . 436
batchhtmlreport . . . 437
binaryName . . . 438
checkdesigninreportidenticalhistory . . . 439
checksimfilesinreportidenticalhistory . . . 440
createbatchhtmlreport . . . 441
createRequirementsWithNoSpec . . . 442
dateTimeSuffixFormat . . . 443
displayAge . . . 444
displaycpk . . . 445
displayFresh . . . 446
displaymax . . . 447
displaymean . . . 448
displaymin . . . 449
displayRelOutputAges . . . 450
displaysigmatotarget . . . 451
displaystandarddeviation . . . 452
displayStress . . . 453
displayyield . . . 454
enablecrossselection . . . 455
estRunTimeDisplayFormat . . . 456
evaluateMetricPrefix . . . 457
historyNamePrefix . . . 458

© 2023 All Rights Reserved.

htmlReportDirectory . . . 459
impJobControlMode . . . 460
impjobpolicy . . . 461
impNetlistingJobPolicy . . . 462
importDisabledTests . . . 463
impresexpperiod . . . 464
impuimode . . . 465
linkdatasheets . . . 466
mappingonetoone . . . 467
numberOfRetries . . . 468
overwritereports . . . 469
retryTimeThreshold . . . 470
reqspecs . . . 471
reqspecscheckunits . . . 472
resultForSpaceCoverage . . . 473
simparallelcount . . . 474
tagsToOverrideWithVerificationSpace . . . 475
usereportidenticalhistory . . . 477
weightedAverageForCoverage . . . 478
Environment Variables to Manage Results . . . 479
allowLoadingIncompleteHistory . . . 480
autoMergeDefault . . . 481
calcStatisticsAcrossCorners . . . 482
calcStatisticsForNonMC . . . 483
ignoreError . . . 484
maxFailures . . . 485
Environment Variables to Manage File Export . . . 486
addSummaryInformation . . . 487
fields . . . 488
Environment Variables to Manage Requirements . . . 489
changeParentOnMerge . . . 490
columnHeadersForImport . . . 491
deadlineFormat . . . 492
extRefPrefix . . . 493
idCreationFunction . . . 494
updateInheritedMappings . . . 495

© 2023 All Rights Reserved.

Environment Variables to Manage Session Windows . . . 496
height . . . 497
width . . . 498
Environment Variables to Manage Verifier Runs . . . 499
clearAllTestJobPolicies . . . 500
distributeImpSet . . . 501
envVarConfigFile . . . 502
impReportDirectory . . . 503
impTimeoutScaleFactor . . . 504
retryUnfinishedOrErrorPoints . . . 505
simMode . . . 506
simulationTimeout . . . 507
stopBatchRunOnServerClose . . . 508
stopTimeout . . . 509
Environment Variables to Manage Verifier GUI Settings . . . 510
displayReqHierId . . . 511
filterOptions . . . 512
impColumnOrder . . . 513
impHiddenColumns . . . 514
resColumnOrder . . . 515
resHiddenColumns . . . 516
reqColumnOrder . . . 517
reqHiddenColumns . . . 518
runColumnOrder . . . 519
runHiddenColumns . . . 520
significantDigits . . . 521
snapEdColumnOrder . . . 522
snapEdHiddenColumns . . . 523
Environment Variables to Manage HTML Reports . . . 524
includeReferenceIDPrefix . . . 525
includeRequirementId . . . 526
useTemplate . . . 527
templateName . . . 528
Environment Variables to Manage Custom Fields . . . 529
csvfile (customFieldConfig) . . . 530
csvfile (customReqFieldConfig) . . . 531

© 2023 All Rights Reserved.

Environment Variables to Manage Snapshots . . . 532
ignoreDisabledElements . . . 533
ignoreImpSetupElements . . . 534
ignoreImpStateElements . . . 535
impElementOrderSensitivity . . . 536
includeImplementations . . . 537
toleranceProperties . . . 538
Environment Variables to Manage Implementations . . . 539
additionalProjectDirectory . . . 540
autoCreateRanOkRequirement . . . 541
autoSyncOutputFilter . . . 542
customCell . . . 543
customHistory . . . 544
customLib . . . 545
customView . . . 546
includeOnlyMappableOutputsInCount . . . 547
loadToActiveWhileViewAndReuseSession . . . 548
Environment Variables to Manage vManager Settings . . . 549
itemLimit . . . 550
hostname . . . 551
lookupHostnameTimeout . . . 552
password . . . 553
port . . . 554
project . . . 555
timeout . . . 556
username . . . 557
vPlan . . . 558

C
Guidelines for Debugging in Virtuoso ADE Verifier . . . 559

© 2023 All Rights Reserved.

© 2023 All Rights Reserved.

1

Introduction to Virtuoso ADE Verifier



With advanced nodes, and a multitude of performance and reliability requirements, analog designs and their verification flows are becoming more complex. The analog components mostly work together with highly complex digital components, creating a mixed-signal system-on-a-chip. As a result, the quality requirements of verification flows are escalating. For example, international standards mandate that companies record all the verification steps for automotive electronic and electrical safety-related systems. Other industry segments have similar standards and requirements. Therefore, managing a design verification project can be challenging, especially when multiple designers are involved in the project containing many design blocks.

With increasing verification requirements, project managers, verification leaders, and designers need a comprehensive and scalable verification solution for their analog-centric IPs. Such solutions are also essential for business success as re-spins are no longer acceptable for designs, such as the design of a mixed-signal system-on-a-chip.

To augment Virtuoso with features that offer a complete analog design verification solution, Cadence provides Virtuoso ADE Verifier (Verifier). Verifier is a comprehensive application to conduct and monitor requirements-driven verification flows for analog designs that can be used in a top-down, bottom-up, or mixed verification flow for your analog and mixed-signal designs. Verifier lets you set up a verification project with many blocks having multiple verification requirements owned by multiple team members. You can map these requirements to their corresponding simulation setups, which are referred to as implementations. You can simulate the implementations to gather the results. Then, you can review the overall verification status of the entire project, and drill down to the detailed status of each verification requirement in the requirements hierarchy. You can customize Verifier to suit your project. It is possible to reuse your verification setup for other projects to improve the verification quality.

© 2023 All Rights Reserved.

You can use Verifier to meet the following objectives for your design verification flows:

- Define the verification requirements for a design project.
- Map the verification requirements with their implementations, which can be ADE Assembler and ADE Explorer cellviews of the type `maestro`, their tests, and outputs.
- Specify references of requirements when multiple designers are involved in the project.
- Simulate multiple implementation cellviews individually, serially, or in parallel.
- Perform and monitor large, batch-based verification regression runs.
- Review the overall verification status of the entire project, and drill down to the detailed status of each verification requirement through various reports.

Verifier maintains verification data in cellviews of the type `verifier` that contain:

- The verification requirement plan, structured as a hierarchy of requirements
- Implementation links and the mapping between the requirements and their corresponding implementations
- Verifier settings, such as the default job policy to use, the source for verification specifications that Verifier uses to determine verification results, among other details

You can use the advanced features of Verifier to accommodate the unique needs of your verification project. For example, you can add custom fields that are specific to your project and manually sign off requirements. Verifier also lets you create owner cellviews of the type `verifier` based on the owners of the requirements. This feature helps project managers assign owners to design verification requirements and monitor the verification status of the owned requirements. Requirement owners can create implementations and map them to their requirements.

In addition to the graphical user interface, Verifier offers a configurable environment and full data access through an extensive set of SKILL functions. For details on these functions with examples, see *Virtuoso ADE Verifier SKILL Reference*.


Video

For a video overview of Verifier, see Introducing Virtuoso ADE Verifier on Cadence Learning and Support portal.

© 2023 All Rights Reserved.



RAK

Download the Virtuoso ADE Verifier Rapid Adoption Kit that contains the design database and instructions on how to implement the basic verification flow for a design.

Related Topics

Key Elements of Virtuoso ADE Verifier

ADE Verifier in Design Verification Flows

Key Elements of Virtuoso ADE Verifier

Key Elements of Virtuoso ADE Verifier

This topic provides the information you need to start and set Virtuoso ADE Verifier (Verifier). You can launch a Verifier session from the Virtuoso environment. Verifier has an intuitive graphical user interface, where the verification content is categorized into these tabs:

- *Setup* – For defining and mapping requirements and implementations.
- *Run* – For managing and reviewing simulations of the implementations.
- *Results* – For reviewing the overall and detailed verification results.
- *Snapshots* – For comparing the detailed project status based on property values at different verification stages.

You can customize the behavior of Verifier to meet the unique objectives of your verification project. For example, you can add custom fields for storing additional information specific to your verification project. Verifier logs useful information, which you can access for obtaining additional details.

Verifier uses the following key elements to provide the design verification infrastructure.

- Requirements
- Implementations
- Verification Status and Reports

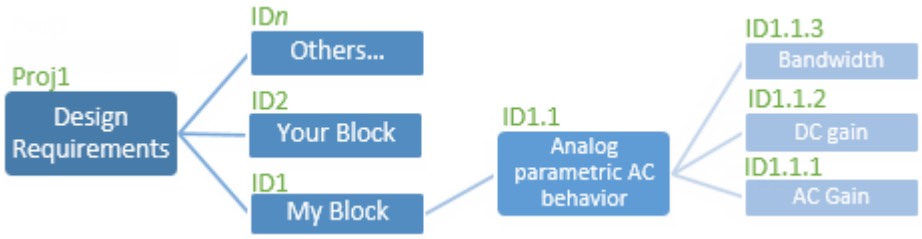
© 2023 All Rights Reserved.

Requirements

A design verification plan includes a set of verification requirements structured in a hierarchical tree. The terms *verification plan* and *requirements hierarchy* are interchangeable. A verification requirement is an abstract description of an aspect of a design that needs to be verified, along with its details. It includes details like the title, description, and verification type with the applicable range of acceptable output values. In a multi-user design environment, requirements can be referenced. You can also specify custom fields to include additional information.

Verifier lets you capture the verification requirements of a design in a hierarchy and provides the mechanism to use them as the base to analyze the verification results. This approach helps implement a requirements-driven verification flow.

Example: Consider that one of the first verification requirements of a design is to check the analog parametric AC behavior of a block. Under this, you can check the AC gain, DC gain, and bandwidth.



Implementations

A verification implementation represents the simulation setup, testbench, and the measured output of a requirement. Implementation data is stored in cellviews of the type `maestro`.

You map requirements to the corresponding implementations. To determine if a design verification requirement has been verified or has failed verification, you run the simulation tests associated with the mapped implementations. Verifier compares the outputs of these tests with the specification values set in the related implementations or requirement and determines the verification results.

Verifier also supports the checks and asserts defined in cellviews as implementations.

Example: Consider that the test output of the `maestro` cellview of the block that captures the DC gain you want to verify is as follows:

`opamp090/full_diff_opamp_AC/maestro/Active.opamp090:full_diff_opamp_AC:1.`**`DCGain`**

© 2023
All Rights Reserved.

You can map this implementation of the output to the requirement `ID1.1.2` for DC gain verification.



Also, the output result must meet a specification to pass verification. You can set this specification in the requirement or the implementation. Consider that the specification is that the result should be more than 25 units (>25).

Verification Status and Reports

Verifier presents the overall requirements-based design verification status and detailed verification results in HTML format. It also displays simulation details of implementations.

Example: Once you run the implementation cellview, you can check if the DC gain requirement that you set failed or passed. The following figure represents how you can review the status of the DC gain requirement directly from Verifier. You can also access details from the other reports that Verifier provides.

Cellview	opamp090/full_diff_opamp_AC/maestro/Active
History Name	verifier_run.2
Test	opamp090:full_diff_opamp_AC:1
Output	DCGain
Specification Status	Pass
Specification	>25
Min Value	25.1416

Related Topics

Introduction to Virtuoso ADE Verifier

ADE Verifier in Design Verification Flows

ADE Verifier in Design Verification Flows

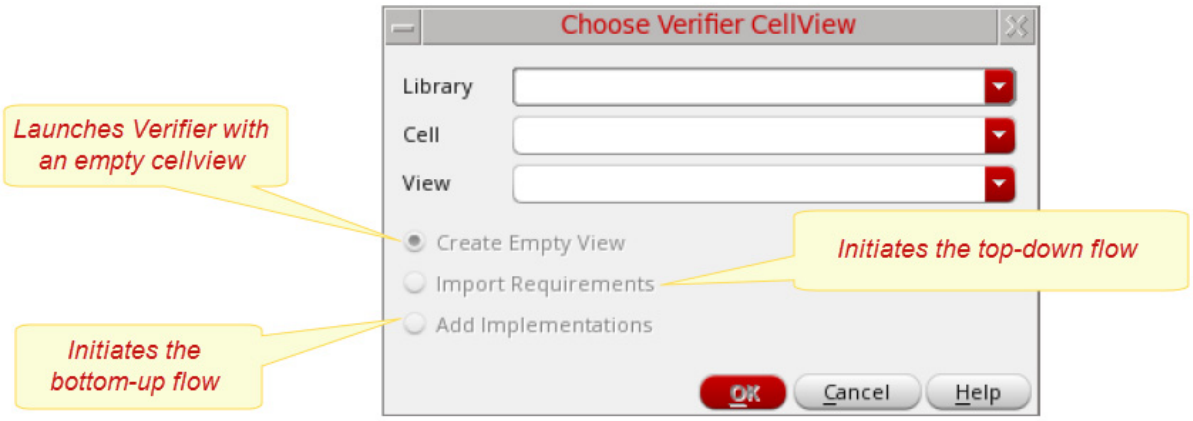
You can customize the use of Verifier according to your design verification needs. For example, you can use Verifier in a top-down flow where you define requirements before you add their implementations, which are stored in cellviews of the type `maestro`. Alternatively, you can use Verifier in a bottom-up flow where you add implementations before you start

© 2023 All Rights Reserved.

developing the verification plan in the form of requirements hierarchy. You can also use Verifier in a meet-in-the-middle flow that has a mix of the top-down and bottom-up elements.

The top-down flow is recommended when you want to reuse a verification plan in different projects. Use the bottom-up flow if the design project is in progress and implementations already exist.

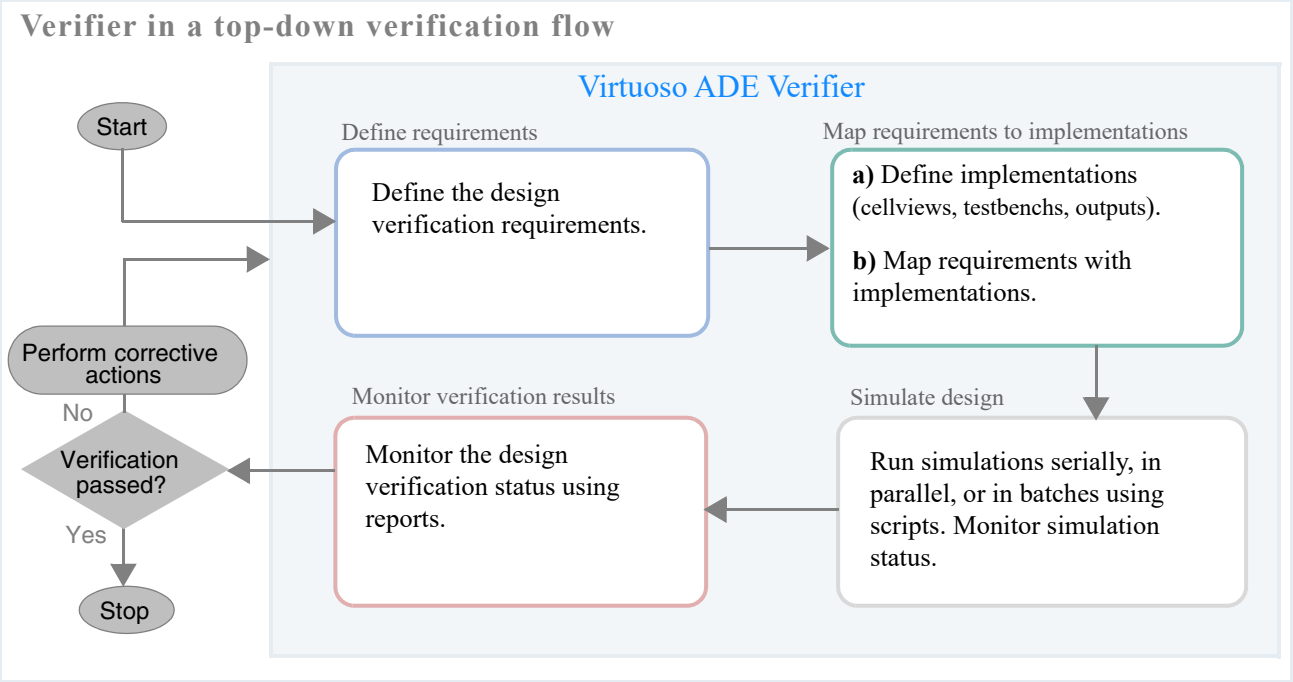
When you launch a new Verifier setup, Verifier initially prompts you to open an existing Verifier cellview. If you specify a cellview name that is not available in the selected library, Verifier prompts you to include implementations for a bottom-up flow, include requirements for a top-down flow, or start Verifier without any data.



© 2023
All Rights Reserved.

ADE Verifier in a Top-Down Verification Flow

The following flowchart illustrates how you can use Verifier in a generic top-down verification flow.



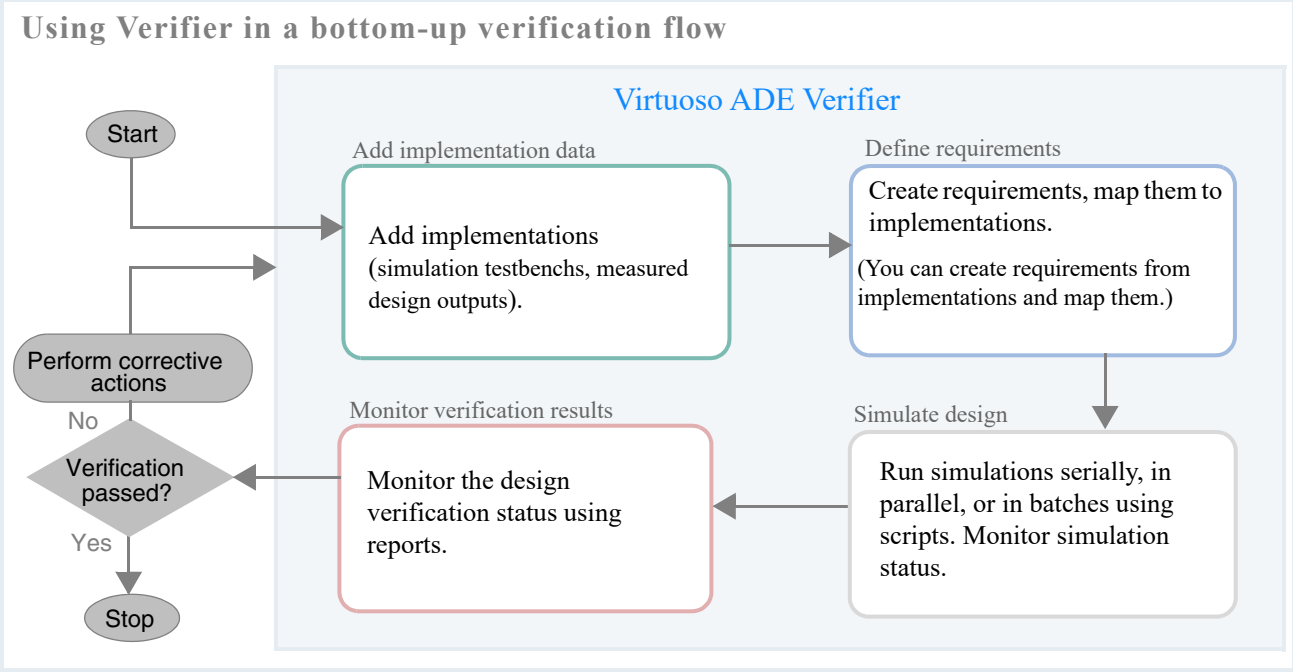
Verifier in a generic top-down verification flow includes the following stages:

1. Defining design verification requirements.
2. Adding implementations, in the form of `maestro` cellviews, and their tests and outputs; then map them with the corresponding requirements.
3. Initiating the simulation of implementation cellviews from Verifier, monitor status, and review results. You can also start the simulation from another application, including ADE Assembler and ADE Explorer, and load results into Verifier.
4. Monitoring the design verification status using reports. Based on predefined criteria set in the requirements and implementations, Verifier analyzes the simulation results and presents the design verification status.

© 2023
All Rights Reserved.

ADE Verifier in a Bottom-Up Verification Flow

The following flowchart illustrates how you can use Verifier in a generic bottom-up verification flow.



Verifier in a generic bottom-up verification flow includes the following stages:

1. Adding implementations to Verifier, which can be cellviews of the type `maestro`, their testbenches, and measured outputs.
2. Creating requirements corresponding to the implementations.

   Verifier lets you create requirements using the data stored in implementations. These requirements are also mapped to their corresponding implementations.
3. Initiating the simulation of implementation cellviews from Verifier, monitor the status, and review results. You can also start the simulation from other applications, ADE Assembler and ADE Explorer, and load results into Verifier.

   You can specify the project requirements and map them to their implementations at any time in the flow, even after simulating the implementations. Verifier presents requirements-based verification results. Therefore, you must map the requirements with their implementations to obtain the verification details.

© 2023
All Rights Reserved.

4. Monitoring the design verification status using reports. Based on predefined criteria set in the requirements and implementations, Verifier analyzes the simulation results and presents the design verification status.

Related Topics

Introduction to Virtuoso ADE Verifier

Key Elements of Virtuoso ADE Verifier

Prerequisites to Using ADE Verifier

To use Verifier, ensure that you have the correct Virtuoso version and license.

Licensing Requirements

Verifier is available with the license 95270 (Virtuoso_ADE_Verifier) in the IC6.1.7 and higher releases.

Appropriate licenses are required to simulate implementations from Verifier. These include the licenses for Virtuoso ADE Assembler and the appropriate simulator licenses for simulating designs and using the appropriate simulation environment.

A single license is checked out for ADE Verifier, even if you run more than a single session or an external run on the same host. To know the correct number of licenses checked out, validate the number of checked out licenses directly on the server.

TIMEOUT

To have ADE Verifier return its license to the license pool after a specified duration, you can add the TIMEOUT option to the license options file. The license is reclaimed when the product has been idle for sometime using this option.

By default, the value of TIMEOUT is infinity, that is, Verifier releases the license only when a session is closed.

Once the license is released to the pool during an inactive session, Verifier tries to regain the license when the session becomes active again. However, if it is unsuccessful in regaining the license, close the current session, and restart the tool. For details, refer to How to Configure Licensing.

© 2023 All Rights Reserved.

With the TIMEOUT option, you can specify how long a license can be inactive before being available for another user. For example, if you want ADE Verifier to time out in two hours of inactivity, add the following command in the options file:

```
TIMEOUT Virtuoso_ADE_Verifier 7200
```

Related Topics

Launching Verifier

Key Elements of Virtuoso ADE Verifier

ADE Verifier in Design Verification Flows

Launching Verifier

You can launch Verifier in multiple ways from Virtuoso for different purposes.

Location	Process to Launch Verifier
	Open an existing cellview of type `verifier`
Virtuoso Library Manager	➡ Double-click an existing cellview.
Virtuoso CIW	**1.** Choose *File – Open*. **2.** Select the library, cell, and view of the type `verifier`. **3.** Click *OK*.
	Create a cellview of type `verifier`
Virtuoso Library Manager	**1.** Choose *File – New – Cell View*. **2.** Select `verifier` from the *Type* list. Specify the library, cell, and view where you want to save the cellview. **3.** Click *OK*.
Virtuoso CIW	**1.** Choose *File – New – Cellview*. **2.** Select `verifier` from the *Type* list. Specify the library, cell, and view where you want to save the cellview. **3.** Click *OK*.

© 2023 All Rights Reserved.

Location	Process to Launch Verifier
	Launch Verifier without any cellview information, which you can add and save later
Virtuoso CIW	➡ Choose *Tools – ADE Verifier*. (Shortcut: `Alt+T` and `F`)

If you specify a new cellview name, Verifier displays options to load implementations, import requirements, or to open an empty Verifier cellview.

After setting up your verification project, save your `verifier` cellview by clicking *Save* on the toolbar. When you save your Verifier setup for the first time or choose to save the opened `verifier` cellview as another cellview, the Save As form appears to let you specify the library, cell, and view name.

Related Topics

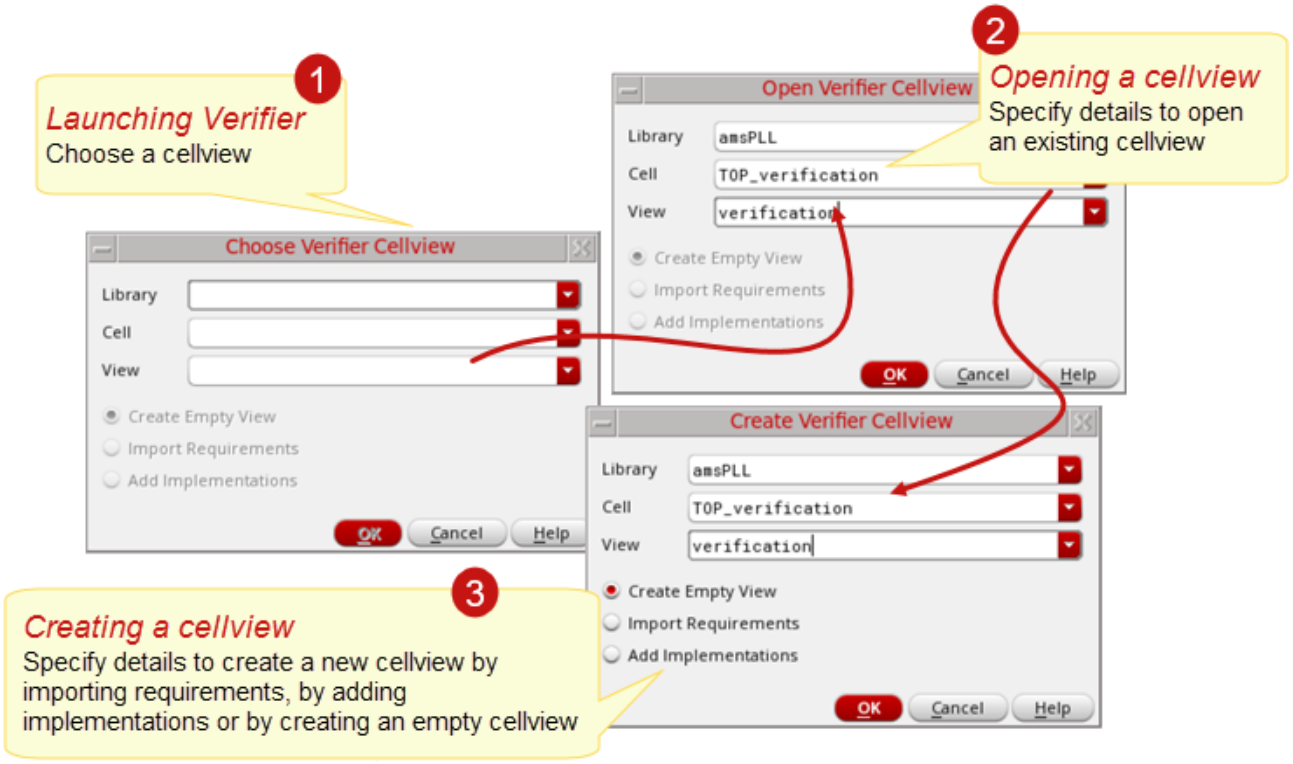
Prerequisites to Using ADE Verifier

Verifier Cellviews and Sessions

© 2023 All Rights Reserved.

Verifier Cellviews and Sessions

The Choose Verifier Cellview form is displayed when Verifier is launched by choosing *Tools – ADE Verifier* from Virtuoso CIW.



You can open only one cellview in a Verifier session running in a Virtuoso session. It is possible to open multiple Verifier sessions in a Virtuoso session. You can use the Verifier SKILL functions to manage Verifier sessions. For details and examples, see *Virtuoso ADE Verifier SKILL Reference*.

If you are using a design management environment with Virtuoso, you must check out the Verifier cellview that you want to edit. To commit your changes, check in the cellview. When you choose to open a Verifier cellview that is not checked out, the environment prompts you to check it out. If you do not do so, the cellview opens in read-only mode. For details on the design management environment, see Generic Design Management Facility.

Related Topics

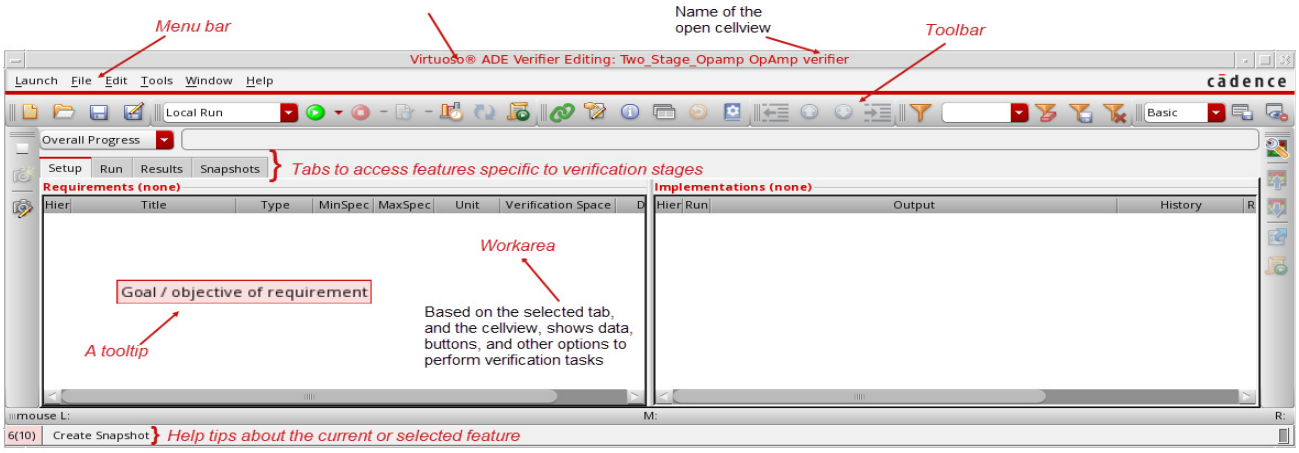
Launching Verifier

Virtuoso ADE Verifier SKILL Reference

© 2023 All Rights Reserved.

Graphical User Interface of ADE Verifier

The main interface of Verifier opens after you specify settings in the Choose Verifier Cellview Form form by selecting *Tools – ADE Verifier* from Virtuoso CIW.



The graphical interface includes the following main components.

- Tabs in ADE Verifier
- Main Toolbar in ADE Verifier
- Assistants in ADE Verifier
- Menu Bar in ADE Verifier

To know more about copying and pasting text or fields, see Copying Requirements in the Hierarchy.

© 2023
All Rights Reserved.

Tabs in ADE Verifier

The following table describes the various tabs available in ADE Verifier.

Tab	Description
Setup	■ Defines design verification requirements, and optionally set their owners. ■ Adds implementations in the form of cellviews of the type `maestro`, their runs, tests, and measured outputs. ■ Maps requirements with implementations, as needed by your design verification project. See Requirements in Virtuoso ADE Verifier and Implementations and Mappings in Virtuoso ADE Verifier.  *Tip* You can copy the selected text or the text in a selected field on this tab using the standard shortcut key `Ctrl+C`.
Run	Starts and stops the simulation of implementation cellviews. You can set the run options in the Preferences form to simulate multiple cellviews simultaneously. You can also specify the settings for Run Mode on the Verifier toolbar. See Implementation Runs in ADE Verifier.
Results	Lets you review the detailed verification status of the complete design verification project and sign off the failures. You can generate reports in HTML format. See Managing Verification Results.
Snapshots	Lets you compare the property values of requirements at different stages in the verification cycle. See Snapshots in Virtuoso ADE Verifier.

To know about creating custom shortcut menus in the above tabs, see Function to Customize shortcut Menus. Custom shortcut menus are not available in the *Snapshots* tab.

Related Topics

Graphical User Interface of ADE Verifier

© 2023 All Rights Reserved.

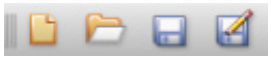
Main Toolbar in ADE Verifier

The main toolbar in Verifier consists of the following sub-toolbars:

- File
- Simulation
- Edit
- Move
- Filter
- Bookmarks *
- Go *
- Workspaces
- Progress Bar

Note: The Bookmarks * and Go * toolbars are displayed in compliance with the Virtuoso framework. Currently, these toolbars are not available for use.

File

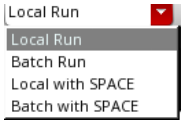


Command	Tooltip	Description
	*New*	Creates a cellview of the type `verifier`.
	*Open*	Opens a cellview of the type `verifier`.
	*Save*	Saves the changes you made to the current cellview.
	*Save As*	Saves the current cellview with a different name. The new cellview is saved at a specified location and the Verifier session keeps the current cellview. Verifier displays the cellview browser when you have to specify a cellview.

© 2023 All Rights Reserved.

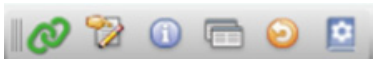
Simulation



Command	Tooltip	Description
	*Run Modes*	Runs the simulation locally, in batch mode, locally with SPACE, or in batch mode with SPACE. When you save the cellview, the run mode is saved to the `settings.v3` file. You can specify the default run mode for any new cellviews by using the simMode environment variable.
 	*Run All / Selected Simulations*	Starts all or selected simulation runs associated with implementations.
 	*Stop All / Selected Simulations*	Stops all or selected simulation runs associated with implementations.
 	*Check for Changes in All / Selected Implementations and Simulation Results*	Checks if all or the selected implementation cellviews and their simulation results have changed since the last run.
	*Calculate Coverage*	Calculates the coverage percentage for the selected implementation and also check which project-specific setup values are not run for the selected design block.
	*Reload Simulation Results*	Reloads the latest simulation results for all implementations in the current setup.
	*Create Batch Scripts*	Creates a script to use with Virtuoso for running simulations and loading simulation results. This script can also be used to only load the results.

© 2023
All Rights Reserved.

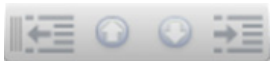
Edit



Command	Tooltip	Description
	*Open / Close Requirements Editor*	Displays or hides the requirement editor assistant in the *Setup* and *Results* tab screens. This editor displays details of the selected requirement to let you edit those details.
	*Enter / Exit Mapping Mode*	Enters or exits the Mapping Mode in the *Setup* tab screen.
	*Open / Close Information assistant*	Displays or hides the Information assistant in the *Setup*, *Run* and *Results* tab screens. This assistant displays details of the selected requirement or implementation in a hierarchically organized format.
	*Open / Close Coverage Summary Assistant*	Displays or hides the *Coverage Summary Assistant* in the *Setup*, *Run* and *Results* tab screens. This assistant displays the coverage details of the selected implementations.
	*Update Implementations or referenced cellviews that are out of date*	Synchronizes the out-of-date implementations or the external references in the session with the current ADE Assembler implementations or Verifier cellviews.
	*Open / Close Setup Library Assistant*	Displays or hides the *Setup Library Assistant* in the *Setup*, *Run* and *Results* tab views. This assistant displays details of the selected corners, sweeps, and verification spaces.

© 2023
All Rights Reserved.

Move



Command	Tooltip	Description
	*Decrease indent of selected items* (`Shift+Left`)	Decreases the indent of selected items in the requirement hierarchy. You can also decrease the indent by using the keyboard shortcut `Shift+Left`.
	*Move selected items up in the tree* (`Shift+Up`)	Moves the selected items up in the requirements hierarchy and the implementations hierarchy. You can also move the items up in the hierarchy by using the keyboard shortcut `Shift+Up`.
	*Move selected items down in the tree* (`Shift+Down`)	Moves the selected items down in the requirements hierarchy and the implementations hierarchy. You can also move the items down in the hierarchy by using the keyboard shortcut `Shift+Down`.
	*Increase indent of selected items* (`Shift+Left`)	Increases the indent of selected items in the requirement hierarchy. You can also increase the indent by using the keyboard shortcut `Shift+Left`.

Filter



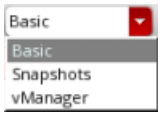
Command	Tooltip	Description
	*Show Filter Row/ Hide Filter Row*	Shows or hides the filter row in the requirement and the implementation panes.
	*Enter filter name*	Specifies a name for the filter and saves it for reuse.
	*Clear the contents of the filters*	Clears all the filter criteria specified in the filter rows.
	*Save Filter*	Saves the current filter in the Verifier setup.

© 2023
All Rights Reserved.

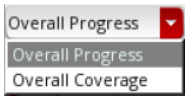
Command	Tooltip	Description
	*Delete Filter*	Deletes the specified filter from the Verifier setup.

Workspaces



Command	Tooltip	Description
	*Workspace Configuration*	Selects a workspace with the default or custom configuration. The available values are *Basic*, *Snapshots* and *vManager*.
	*Save Workspace*	Saves the currently selected workspace.
	*Toggle Assistants Visibility*	Hides or shows the selected assistants.

Progress Bar

Command	Tooltip	Description
	*Overall Coverage/ Overall Progress*	Displays the status of *Overall Progress*, which shows the overall results as *Pass*, *Fail*, *No Results*, and *Not Mapped*. Additionally, it also displays the status of *Overall Coverage*, which measures the coverage of all project-specific setup requirements in the simulation results.

Related Topics

Graphical User Interface of ADE Verifier

© 2023 All Rights Reserved.

Assistants in ADE Verifier

The following assistant panes are available in Verifier:

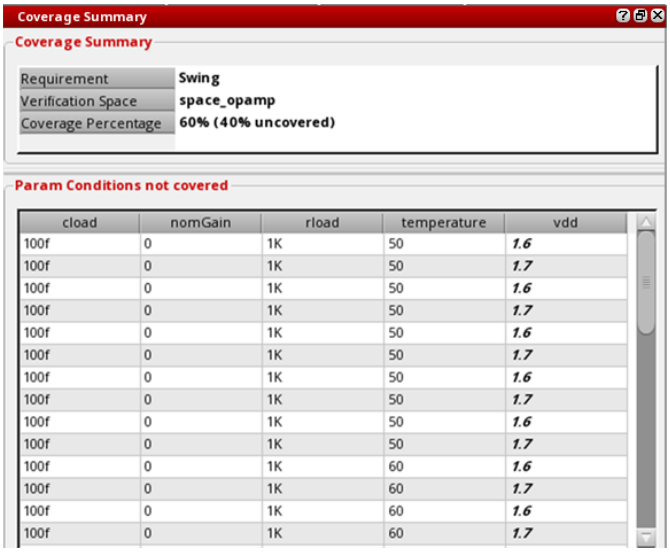
- Coverage Summary Assistant
- Information Assistant
- Requirements Editor Assistant
- Setup Library Assistant
- Snapshots Editor Assistant

Coverage Summary Assistant

The Coverage Summary assistant allows you to view the coverage summary and parameter conditions that are not covered for the requirement.

To view the Coverage Summary Assistant, do one of the following:

- Click the  icon on the toolbar.
- Choose *Window – Assistants – Coverage Summary*.
- Right-click on the toolbar or the menu area, and choose *Assistants – Coverage Summary*.



For information about using the Coverage Summary assistant pane, see Viewing the Coverage Summary.

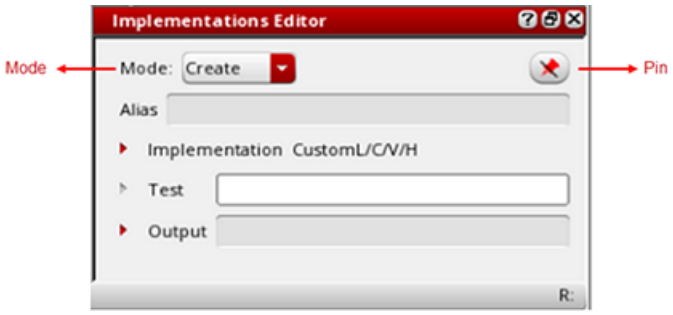
© 2023 All Rights Reserved.

Implementations Editor Assistant

The Implementations Editor assistant allows you to view the currently selected implementation and view or modify a custom implementation. This assistant is helpful in the *Setup* tab.

To view the Implementations Editor, do one of the following:

- Right-click the toolbar or the menu area, and choose *Assistants – Implementations Editor*.
- Choose *Window – Assistants – Implementations Editor*.



The Implementations Editor assistant follows the current selection in the *Implementations* pane. To pin the editor on an implementation, select the implementation and click the *Pin* button on the top right corner of the assistant pane. This is especially useful during drag-and-drop operations.

Depending on the current selection in the *Implementations* pane, one or more of the following modes might be available in the Implementations Editor:

- *Create*: This mode is activated when you choose *Create Custom Implementations* from the shortcut menu in the *Implementations* pane. New custom implementations are created based on the specified values.
- *Modify:* This mode is activated when you choose *Modify* in the *Mode* list or when you choose *Modify Custom Implementations* from the shortcut menu in the *Implementations* pane. When this mode is selected, the Implementations Editor lets you make changes to the values or names assigned to tests, outputs, aliases, or specifications.
- *View*: This mode is activated when you choose *View* in the *Mode* list or when you select a maestro implementation. In this mode, the Implementations Editor performs as a viewer of information, and disables any changes in the implementation details.

© 2023
All Rights Reserved.

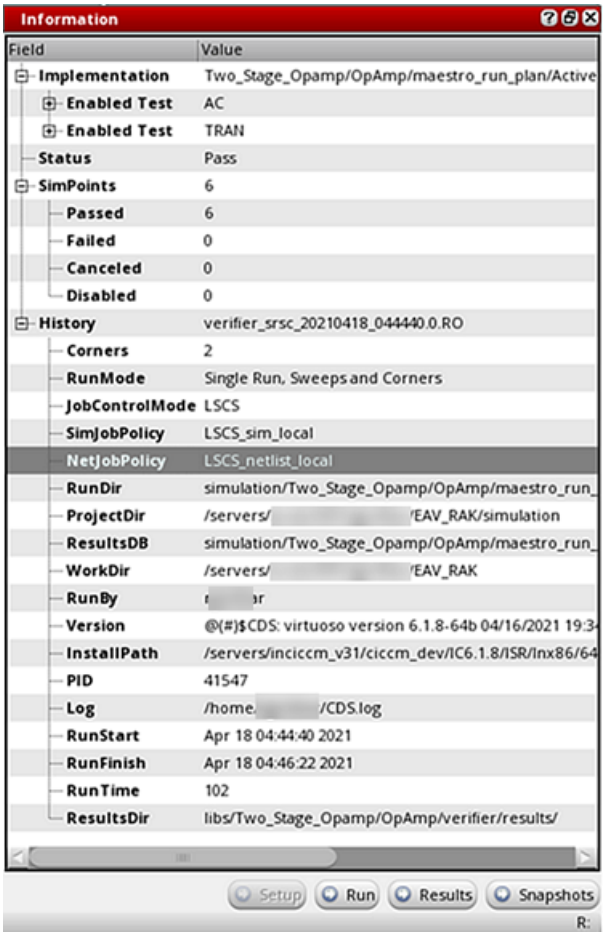
For information about using the Implementations Editor assistant, see Edit Custom Implementations.

Information Assistant

The Information assistant allows you to view the information associated with the current selection in a hierarchically organized format. These selections can be a requirement, an implementation, a simulation run, or the results of an implementation run.

To display the Information assistant, do one of the following:

- Click the  button on the toolbar.
- Right-click the toolbar or the menu area, and choose *Assistants – Information*.
- Choose *Window – Assistants – Information*.



© 2023
All Rights Reserved.

When you select a requirement, the Information assistant shows the following sections of information:

- Requirement (properties)
- OverallStatus (numbers and result data age)
- Signoff (signoff data)
- Mappings (mapped implementation cellview, run, reliability, test, or output)

When you select an implementation cellview, run, reliability, test or output, the Information assistant shows the following sections of information:

- <Type> (setup or properties)
- Status
- *SimPoints*
- History (the run mode, job control mode, simulation job policy, net job policy, simulation information for implementation or run, design cellview and simulator used for the test, specs, unit, min/typical/max values, parameters and failures for output)

Clicking the *Setup*, *Run*, *Results*, and *Snapshots* buttons on the assistant pane allow you to navigate to the corresponding page to view the related information and select the item that was shown in the assistant, if it exists on that page.

For example, when you click an implementation in the *Setup* page and open the Information assistant, the assistant displays the details of the implementation, the simulation points, and the history. Now, when you click the *Results* button in the assistant pane, the *Results* page is displayed and the requirement that is mapped to the implementation is selected. The assistant now displays the details related to the requirement, overall status, and mappings.

In contrast, when you manually select the *Setup*, *Run*, *Results*, and *Snapshots* tabs to open their corresponding pages, the information displayed in the assistant changes depending on the last selection on that page. For example, you select a requirement with ID `1.1` on the *Setup* page and the Information assistant shows its corresponding details. Now, if you switch to the *Results* page, the assistant might show the details of a different requirement that was selected the last time you worked on the *Results* page. If you switch to the *Setup* page, the Information assistant now displays the details of requirement `1.1` again.

For information about using the Information assistant pane, see Viewing Run Results Using the Information Assistant and Bindkeys for View Navigation and Interaction with Item Contents.

© 2023 All Rights Reserved.

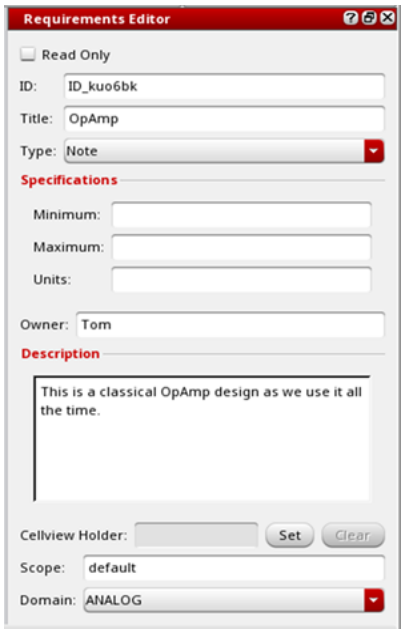
Requirements Editor Assistant

The Requirements Editor assistant allows you to view and edit the currently selected design verification requirements. This assistant is helpful in the *Setup* and the *Results* tab.

To view the Requirements Editor, do one of the following:

- Click the  icon on the toolbar.
- Right-click the toolbar or the menu area, and choose *Assistants – Requirements Editor*.
- Choose *Edit – Open Requirements Editor*.
- Choose *Window – Assistants – Requirements Editor*.

  For information about using the Requirements Editor assistant, see Editing Requirements Using the Requirements Editor Assistant.



Setup Library Assistant

The Setup Library assistant helps you view the information associated with the current simulation run and the implementation run results.

To view the Setup Library assistant, do one of the following:

- Choose *Window – Assistants – Setup Library*.

© 2023 All Rights Reserved.

- Right-click on the toolbar or the menu area, and choose *Assistants – Setup Library*.



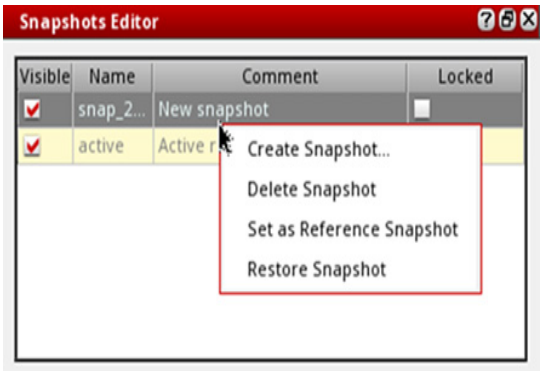
For information about using the Setup Library assistant, see Verifying the Design Against the Specified Setup.

Snapshots Editor Assistant

The Snapshots Editor assistant allows you to view the information associated with the current verification setup and compare it with previous setups, that are captured as snapshots.

To view the Snapshots Editor assistant, do one of the following:

- Choose *Window – Assistants – Snapshots Editor*.
- Right-click on the toolbar or the menu area, and choose *Assistants – Snapshots Editor*.
- Choose *Window – Workspace – Snapshots*.
- Choose *Snapshots* in the workspace list on the Verifier toolbar.



For information about using the Snapshots Editor assistant pane, see Snapshots Editor.

For more information about the various assistants in Virtuoso, see Toolbars and Assistant Panes.

© 2023 All Rights Reserved.

Related Topics

Graphical User Interface of ADE Verifier

© 2023 All Rights Reserved.

Menu Bar in ADE Verifier

The following table describes the various menus available on the menu bar in ADE Verifier.

Menu	Types of Options
File	Lets you do the following: ■ Create, open, and save cellviews of the type `verifier` ■ Open the cellview in read-only or editable mode **Note:** In read-only mode, you cannot access some features that require changes in the cellview, such as mapping requirements to implementations. However, features like viewing verification reports are available in read-only mode because they do not require changes in the cellview. ■ Export requirements to a CSV, Excel, or imported file ■ Import implementations and requirements from external sources ■ Export mappings to, or import mappings from a CSV or Excel file ■ Access previously opened cellviews and exit Verifier
Edit	Lets you do the following: ■ Set your preferred general options, run options, and report options for Verifier. Also set the values of the custom project fields, if enabled. ■ Show or hide the Requirement Editor assistant

© 2023 All Rights Reserved.

Menu	Types of Options
Tools	Lets you do the following: ■ Add multiple implementation cellviews, and optionally create corresponding requirements ■ Define the Typical Value Setup for the implementations in the cellview ■ Start and stop simulation runs of implementation cellviews ■ Create a batch script ■ Check whether the implementation has changed since the last run or reload in Verifier if the age of simulation results of implementation is older than the specified expiration period ■ Reload implementation simulation results when the simulation is initiated from an external application ■ Access verification reports in HTML format
Window	Lets you do the following: ■ Select the assistants to display in Verifier and toggle their visibility on or off ■ Select the toolbars to display in Verifier and toggle their visibility on or off ■ Select, save, delete or load workspaces ■ Select the tabs to close

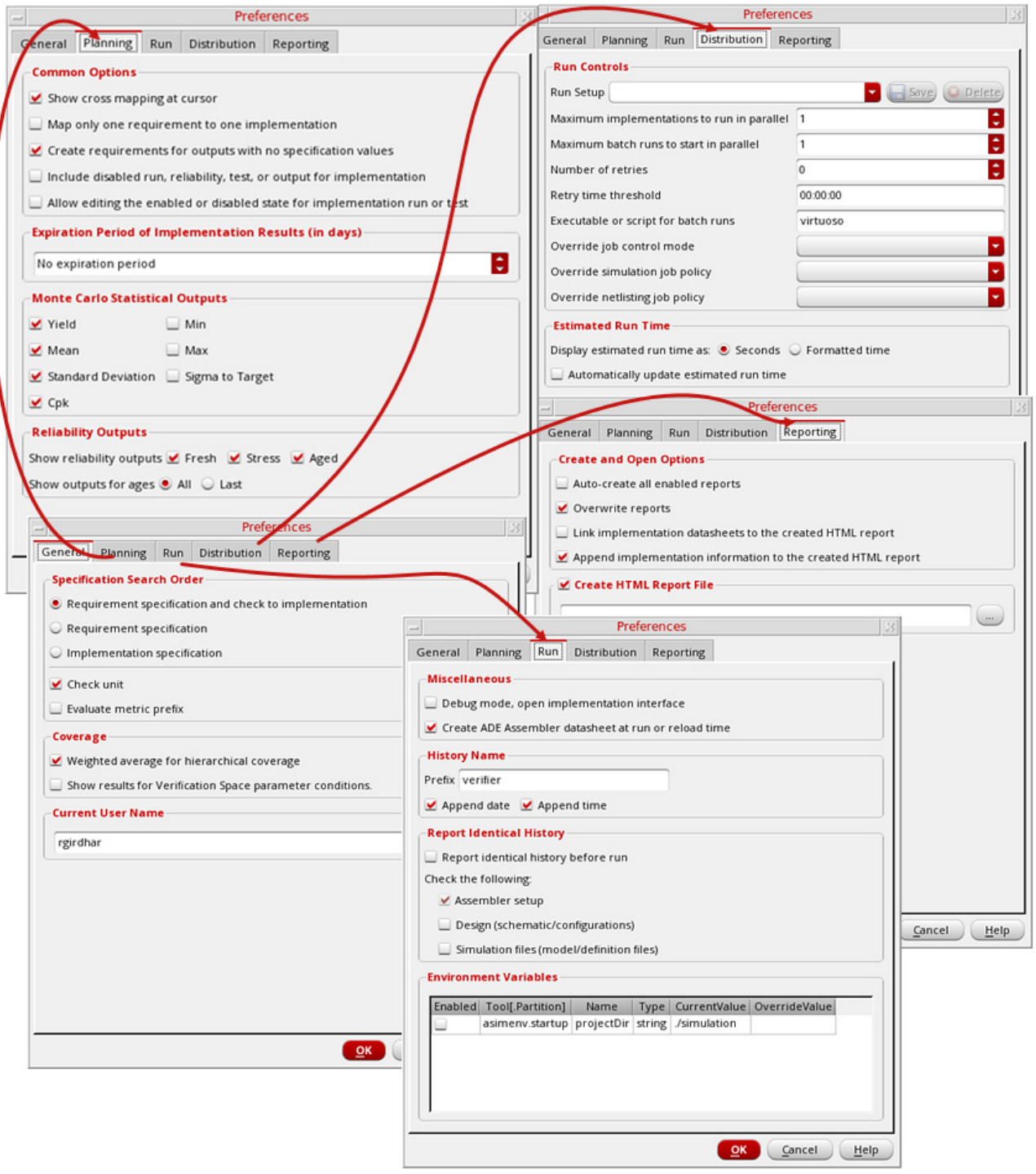
Related Topics

Graphical User Interface of ADE Verifier

© 2023 All Rights Reserved.

Setting Virtuoso ADE Verifier Preferences

You can configure the general, planning, run, distribution, and report options of Verifier using the Preferences form.



© 2023
All Rights Reserved.

To set your Verifier preferences:

1. Choose *Edit – Preferences*.

   The Preferences Form appears.

2. Click the tab that provides the options you want to set.

   For example, to specify the number of simulation jobs that can run in parallel, click the *Run Options* tab.

3. Set the required options.

   If you have configured Verifier to use custom fields, see Custom Fields to set the project-level custom fields.

4. Click *OK*.

The current Verifier session implements your settings. When you save the setup in a cellview, these settings are also saved.

You can also configure Verifier by setting environment variables. For details, see Verifier Environment Variables.

Related Topics

Preferences Form

Environment Variables to Set Preferences

Logs and Automatic Backups in Virtuoso ADE Verifier

Verifier logs useful information of the current session. It indicates the reason for the failure of a specification check in the tooltips. All important messages are displayed in the CIW.

Verifier saves the current setup settings in the `settings.v3` file. When saving a session, Verifier creates a backup of the `settings.v3` file as `settings.v3~`. When you open a new session, Verifier checks if the current `settings.v3` contains any setup information. If the file is empty, Verifier then looks for an existing backup with data, which is the `settings.v3~` file, and displays a dialog box to confirm whether you want to restore from the backup found, or continue with a new setup.

© 2023 All Rights Reserved.

Related Topics

Launching Verifier

Prerequisites to Using ADE Verifier

Global Bindkeys in Virtuoso ADE Verifier

The following table lists the bindkeys applicable in Verifier.

Bindkey	Function
Access to File menu options	
`Ctrl + N`	New...
`Ctrl + O`	Open Cellview...
`Ctrl + S`	Save Cellview
`Shift + S`	Save As...
`Ctrl + W`	Close
Access to Edit menu options	
`Ctrl + P`	Preferences...
Access to Tools menu options	
`Ctrl + L`	Add Implementation Cellviews...
`Ctrl + R`	Run Simulations
`Ctrl + T`	Stop Simulation Runs
`Ctrl + B`	Create Batch Script...
`Ctrl + X`	Check for Changes in Implementations and Simulation Results
`Shift + R`	Reload All Simulation Results
`Ctrl + U`	Update Implementations and References
`Ctrl + I`	Manage Imported Files
`Ctrl + H`	Publish HTML Report
Access to visibility options	
`F11`	Toggle the display of open Assistants

© 2023 All Rights Reserved.

Bindkey	Function
`Shift + F11`	Toggles the display of open assistants and toolbars
`Ctrl + F11`	Toggles the display of open toolbars
Access to Assistants	
`Ctrl + Shift + E`	Opens the Requirements Editor
`Ctrl + Shift + I`	Opens the Information
`Ctrl + Shift + O`	Opens the Coverage Summary
`Ctrl + Shift + S`	Opens the Setup Library
Access to the Requirements hierarchy	
`Ctrl + C`	Copies the requirement
`Ctrl + V`	Pastes the requirement
`Ctrl + Shift + C`	Copies the requirement hierarchically
Access to mappings in the Requirements and Implementations hierarchy	
`M`	Enters the mapping mode
`M/ESC`	Exits the mapping mode
`Ctrl + D`	Deletes the mapping
`Ctrl + M`	Maps the selected requirement and implementation
`Shift + C`	Copies the mapping
`Shift + V`	Pastes the mapping
Access to the selection in the hierarchy	
`Space`	Selects the current item
`Ctrl + Space`	Deselects the selected items
`Up`	Selects the previous item
`Down`	Selects the next item
`Left`	Selects the parent item or collapse the expanded item
`Right`	Selects the first child item or expand the collapsed item

© 2023 All Rights Reserved.

Bindkey	Function
Make changes in the hierarchy	
`Ctrl + Shift + Up`	Moves the selected item up by one level
`Ctrl + Shift + Down`	Moves the selected item down by one level in the hierarchy
`Ctrl + Shift + Left`	Decreases the indentation in the hierarchy
`Ctrl + Shift + Right`	Increases the indentation in the hierarchy

Related Topics

Bindkeys for View Navigation and Interaction with Item Contents

Bindkeys for View Navigation and Interaction with Item Contents

All hierarchies in a view support a set of key bindings that enable the user to navigate in the view and interact with the contents of the items.

Bindkey	Function
`Up`	Moves the cursor up to the item in the same column on the previous row. If the parent of the current item has no more rows to navigate to, the cursor moves to the corresponding item in the last row of the preceding node.
`Down`	Moves the cursor to the item in the same column on the next row. If the parent of the current item has no more rows to navigate to, the cursor moves to the relevant item in the first row of the sibling that follows the parent.
`Left`	Hides the children of the current item (if present) by collapsing a branch.
`Minus`	Hides the children of the current item (if present) by collapsing a branch.
`Right`	Reveals the children of the current item (if present) by expanding a branch.
`Plus`	Reveals the leaf nodes of the current item (if present) by expanding a branch.

© 2023 All Rights Reserved.

Bindkey	Function
`Asterisk`	Expands the current item and all its children (if present).
`PageUp`	Moves the cursor up by one page.
`PageDown`	Moves the cursor down by one page.
`Home`	Moves the cursor upward in the same column to the first row of the first top-level item in the model.
`End`	Moves the cursor to an item in the same column of the last row of the last top-level item in the model.

Related Topics

Global Bindkeys in Virtuoso ADE Verifier

© 2023 All Rights Reserved.

2

Requirements in Virtuoso ADE Verifier

Requirements are central to a design verification flow implemented using Virtuoso ADE Verifier (Verifier). Requirements form the hierarchical verification plan of your design verification project. You map requirements to their implementations. Verifier determines the verification status of the requirements using this mapping information, along with other settings and results. For details on implementations, see Implementations and Mappings in Virtuoso ADE Verifier

In addition to adding requirements manually, you can copy or reference requirements from a spreadsheet, a comma-separated values (CSV) file, or another Verifier cellview. It is also possible to export requirements to a CSV file for reference and reuse. Verifier lets you edit local requirements directly in the requirements hierarchy or through the Requirement Editor Assistant. To edit referenced requirements, you must modify its source.

Related Topics

Details of a Requirement

Organizing the Requirements Hierarchy

Exporting Requirements

Import Requirements from External Sources

Filters in the Requirements Hierarchy

Custom Fields

© 2023 All Rights Reserved.

Details of a Requirement

A requirement is an abstract description of a design aspect that needs to be verified. You organize the requirements of a verification project in a hierarchy, which is also referred to as the verification plan. When defining the requirements for a design verification project, focus on what aspects of the design you want to verify. You can later define how those aspects must be verified by mapping them with their corresponding implementations.

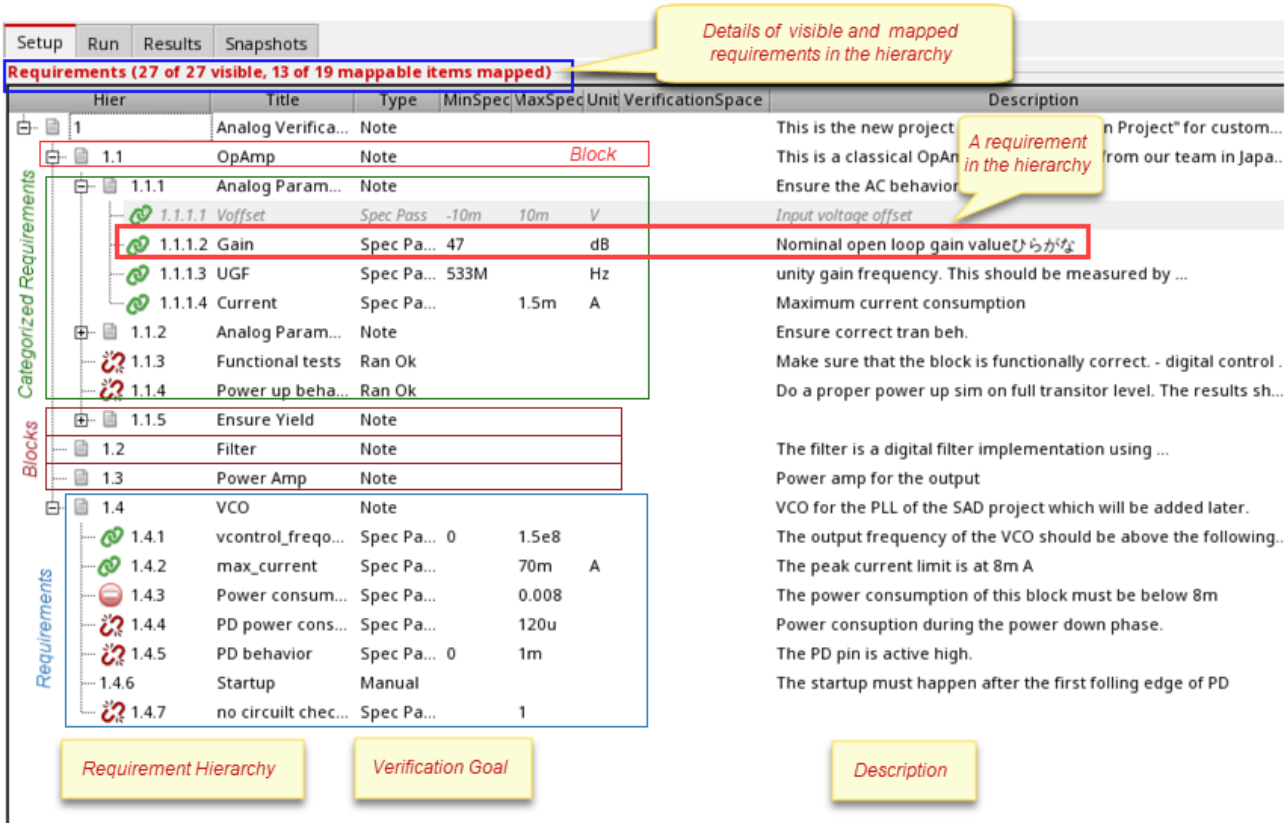
It is important to appreciate the difference between requirements and implementations. Having separate sets of requirements and implementations can help in refining the verification project and quality. For example, when you review the implementations, you can identify aspects that are missing from the verification plan. Such differences can help you to better understand and capture the verification challenges and find the best use of your verification resources.

In your verification project, you can treat the verification plan as a living document, which you can refine throughout the verification cycle. It is also useful to continue to use the verification plan to ensure that the plan accommodates the unique needs of your verification project.

To determine the status of the verification project, Verifier uses the information stored in the requirements, along with other related information and settings. For example, you can set a requirement to pass verification if the mapped implementation is simulated successfully. In this case, Verifier checks the simulation status of the mapped implementation to determine the verification status of the requirement. The overall verification status largely depends on the verification plan. For example, if a set of requirements are missing from the verification plan, the design verification results will be inaccurate.

© 2023 All Rights Reserved.

The following figure illustrates a hierarchy of requirements. Also see Requirements.

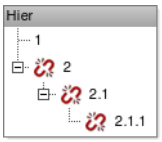


You can show or hide the columns of the requirements hierarchy. For this, right-click anywhere on the column headings in the requirements area and select the columns you want to show. Verifier hides the columns that are not selected.

To sort columns, you can use the drag-and-drop method and change the display of columns. Additionally, you can specify the default column order and hidden columns by using the environment variables reqColumnOrder and reqHiddenColumns.

© 2023
All Rights Reserved.

The following table describes the details of a requirement.

Column	Description
Hier	Unique hierarchical number that Verifier assigns to a requirement to identify its placement in the requirements hierarchy. You cannot edit this number. When you change the requirements hierarchy, the hierarchical number of the requirements update according to the new hierarchy. The *Hier* column is always displayed and includes the mapping status icon with the hierarchical number for each requirement.  Mapping status icons displayed with the hierarchical number: The requirement type is set to *Note* or *Manual*. The requirement is mapped with one or more implementations. The requirement is mapped with one or more implementations where one of them is a custom implementation. The requirement needs to be mapped, but is not mapped with any implementations. The specification checks failed, which indicate the following: ■ Verifier is set to use the preference option *Requirement specifications and check to implementation*. Verifier checks if the specifications set in the mapped requirements and implementations match. If they match, Verifier uses the specifications for validating the requirement. ■ The requirement meets the following conditions: ❑ Is mapped with one or more implementations ❑ Has the verification type *Spec Pass* ❑ The specification in the requirement does not match the specification in the mapped implementation. This status is the same as the implementation status *Spec Check Fail*. The requirement is referenced from an existing Verifier cellview. The requirement that is referenced from an existing Verifier cellview, is out-of-date.

© 2023
All Rights Reserved.

Column	Description
ID	Unique ID of the requirement. The ID column is hidden by default. When you add a new requirement, Verifier automatically assigns to it a Universally Unique Identifier (UUID), such as 00f9a512-0154-4592-b0fa-7b3767f716fd. The *ID* column is hidden by default. To assign a different requirement ID, you can use idCreationFunction. It defines a public SKILL function for creating a new requirement ID. Ensure that you define IDs of a reasonable size. For example, if 100 unique IDs are to be created, specify an ID that contains at least three to four random characters. When a user-defined SKILL function returns an ID that is not unique, an incremental suffix is added to the ID to make it unique.
Title	Descriptive title of the requirement. ■ The following title naming rules are applied on a requirement title: ❑ The title cannot be empty ❑ The title must be a single-line text ❑ The title must not contain any slashes ❑ The title must be unique within the current hierarchical level ■ Verifier reports an error if you specify an invalid title: ❑ When you import requirements from a CSV or Excel file ❑ When you add a requirement with specified title using SKILL functions ❑ When you modify the requirement title in the hierarchy, using the Requirements Editor assistant or using SKILL functions ■ Verifier adds the suffix '_<num>' to the title to make it unique for one parent automatically when: ❑ Loading old settings.v3 ❑ Copying and pasting requirements ❑ Dragging and dropping requirements ❑ Moving requirements

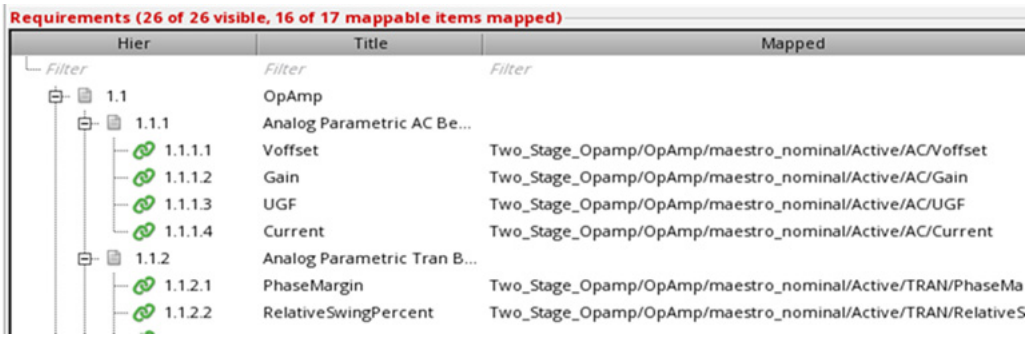
© 2023 All Rights Reserved.

Column	Description
Type	Verification type of the requirement. The type can be one of the following: ■ *Note*: The requirement is treated as a note, without any need to check or pass any verification goals. Typically, this type represents a hierarchical node in the tree, a comment, or a placeholder item. You cannot map such a requirement with an implementation. ■ *Spec Pass*: The requirement passes when the measured output for all the sweeps and corners that are run meet the selected specifications set in the requirements or the implementations. You can provide the specification values in the *MinSpec*, *MaxSpec*, and *Unit* fields of the requirement. Use this option when mapping a requirement with an output. **Note:** The type of requirement changes to `Spec Pass` from `Ran OK` whenever a valid value is specified for the *Min Spec* or *Max Spec* value. ■ *Ran OK*: The requirement passes when the simulation is completed successfully. However, the simulation results are not checked automatically. Use this option when mapping a requirement to an implementation cellview or a test. When you map a requirement of this type to an output, Verifier checks if the measurement worked by evaluating the expression to some result, but does not check its value. ■ *Manual*: The requirement is marked for manual verification. You can sign off the requirement manually from the *Results* tab. You can map such a requirement with an implementation. The mapping is typically done to help find the related items that should be reviewed before signing off the requirement. ■ *ExtRef*: The requirement is imported in the current cellview as an external reference to an existing verifier cellview. When you import the requirement, its mapped implementations and results are also imported. You can edit the ID, Title, Specifications, and Description in the Requirement Editor for the main node of the external reference tree in the requirement hierarchy. For details, see External References in Virtuoso ADE Verifier
MinSpec	Minimum specification value. It is used to determine if the requirement passed when the type is set to *Spec Pass*. You can specify a numeric as well as string value. The string value is shown as the blue-colored text.
MaxSpec	Maximum specification value. It is used to determine if the requirement has passed when the type is set to *Spec Pass*. You can specify a numeric as well as string value. The string value is shown as in blue-colored text. *MaxSpec* must always be greater than *MinSpec*.

© 2023 All Rights Reserved.

Column	Description
Unit	Unit of measurement of the specified minimum and maximum specification values.
VerificationSpace	Verification space assigned to a requirement.
Owner	Owner of the requirement when multiple people are involved in the design verification project. You can specify a list of comma-separated owners for a requirement. The current user is the default owner of the new requirements you add in the Verifier session. The *Owner* column is hidden by default.
CellviewHolder	Specifies which Verifier view an item belongs to.
Scope	Specifies a 'tag' to define a group of requirements or results that belong together. You can filter the hierarchical plan and the results based on the scope by using this tag.
Domain	Specifies if the requirement is analog (`ANALOG`) or digital (`DIGITAL`).
Description	Detailed description of the requirement. See Description Editor.
ReadOnly	Select to make the requirement read-only. The read-only requirements appear in gray italic text. The *ReadOnly* column is hidden by default.
Deadline	Specifies the deadline or due-date for a requirement in the predefined date format. Environment variable: deadlineFormat.

© 2023 All Rights Reserved.

Column	Description
Mapped	Shows a comma-separated list of items that are mapped to the current requirement. A mappable item can be an implementation, run, reliability setup, test or an output. The values in this column allow you to quickly validate if all the requirements are mapped to the correct implementation. The *Mapped* column is hidden by default. 

Row	Description
Filter Row	This option is available when you right-click anywhere on the column headings and select *Filter Row*. This row contains individual filters for every column in the requirement hierarchy. You can enter the required criteria in a column filter to display only the requirements matching the criteria. You can also refine the filtered requirements by specifying criteria in other columns. This row is hidden by default. See Filtering Requirements Based on Text Criteria.

Related Topics

Adding Requirements

Deleting Requirements

© 2023
All Rights Reserved.

Shortcut Menu Options in the Requirements and Implementations Pane

The following table describes the shortcut menu options available in the *Requirements* and *Implementations* panes:

Option	Description
Edit Implementation Cellview	Opens the implementation cellview in edit mode. Displayed for Requirement type Spec Pass.
View Implementation Results	Opens the implementation cellview in read-only mode and load the last history that was active during the last Verifier run. Displayed for Requirement type Spec Pass. If the implementation cellview uses the integrated history management, you receive a popup asking you to convert the cellview to use the separate history management scheme.
View Simulation Log of Implementation Test	Lets you view the simulation log for the specified test in the implementation cellview. Displayed for Requirement type Spec Pass. If the implementation cellview uses the integrated history management, you receive a popup asking you to convert the cellview to use the separate history management scheme.
Add Requirement	Adds a requirement in the hierarchy. By default, this option adds a requirement of the type *Note*.
Import Cellview	Imports an existing Verifier cellview as a copy or an external reference.
Delete Requirements	Deletes the selected requirements.
Reassign ID Hierarchically	Reassigns the ID for the selected requirements hierarchically.
Set Owners for Selected Requirements	Specifies the owner of the requirement when multiple people are involved in the design verification project. You can specify a list of comma-separated owners for a requirement. The current user is the default owner of the new requirements you add in the Verifier session. For details, see External References in Virtuoso ADE Verifier. The *Owner* column is hidden by default.

© 2023 All Rights Reserved.

Option	Description
Set Selected Requirements Read Only	Makes the requirement read-only. The read-only requirements appear in gray italic text.
Edit Description	Makes changes to the description of the selected requirement. For details, see Description Editor for Requirements.
Set Cellview Holder	Makes the current cellview the holder for the vPlan.
Clear Cellview Holder	Clears the cellview holder for the vPlan.
Copy	Copies the selected requirement. Press the `Ctrl` key to select multiple requirements.
Copy Hierarchically	Copies the selected requirement hierarchically.
Paste	Pastes the copied requirements.
Map	Maps the selected requirement and implementation.
Delete Mapping	Deletes all mappings of the selected requirement.
Copy Mapping	Copies the mapping of the selected requirement.
Apply Mapping	Applies the copied mapping to the selected requirement.
Assign Space to Requirements	Assigns a verification space to the specified requirements and then use them to calculate the coverage.
Remove Space from Requirements	Removes the assignment of the verification space to the specified requirements.
Expand	Expands the selected nodes in the hierarchy. If no node is selected, all nodes are expanded.
Collapse	Collapses the selected nodes in the hierarchy.
Show	Shows the requirements that are Mapped, Unmapped or have Spec Check Fail.

Related Topics

Details of a Requirement

Details of an Implementation

© 2023 All Rights Reserved.

Adding Requirements

Verifier lets you add requirements in multiple ways. In a top-down flow, you can create a blank cellview and import requirements, or you can create requirements manually. In a bottom-up flow, you can create requirements from implementations.

While creating requirements, the parent requirement should always be assigned with type `Note`. While creating requirement from an implementation cellview or an implementation test, two requirements are created where the parent requirement is assigned with type `Note`, and the child requirement is assigned with type `Ran OK`.

Related Topics

Adding a Requirement Manually

Import Requirements from External Sources

Creating Requirements from Implementations

Editing Requirements in the Requirements Hierarchy

Editing Requirements Using the Requirements Editor Assistant

Adding a Requirement Manually

You can add unique requirements manually.

To add a requirement:

1. Click the *Setup* tab to display the requirements area.
2. Do one of the following:
   - Right-click in the requirement area and choose *Add Requirement*.
   - Click *Add*.

   A new requirement is added.



© 2023
All Rights Reserved.

- Add the requirement details in the Requirements Editor. See Adding Requirements using the Requirements Editor Assistant.

You can edit the requirement details, except for the auto-generated hierarchical number under the column *Hier*.

Additional Information

- The default type of a new requirement is *Note*.
- The default owner of a new requirement is the current Verifier user.
- It is possible to change the current user name that Verifier uses for requirements. For this, choose *Edit – Preferences* and type the user name in the *Current User Name* field. This field displays the login name by default. For details, see External References in Virtuoso ADE Verifier.
- When you select a requirement and add a new requirement, the type of selected requirement influences the position of the new requirement.
  - When the selected requirement is of type 'Note', the new requirement is added as a child requirement at the end.
  - When the selected requirement is a type other than 'Note', the new requirement is added after the selected requirement.
- The hierarchical number is auto-generated for the requirements based on the updated hierarchy. You cannot edit this number manually.
- By default, Verifier assigns to each new requirement an alphanumeric UUID of a format similar to `00f9a512-0154-4592-b0fa-7b3767f716fd`. To specify a requirement ID of a user-defined format, set the idCreationFunction environment variable.

  The regular expression for the valid IDs is: ^[-a-zA-Z0-9_.]+$. This means that the valid characters include only the alphabets in uppercase and lowercase, digits and the characters '-', '_' or '.'.

Related Topics

Adding Requirements

Import Requirements from External Sources

Creating Requirements from Implementations

Editing Requirements in the Requirements Hierarchy

© 2023 All Rights Reserved.

Editing Requirements Using the Requirements Editor Assistant

Import Requirements from External Sources

You can maintain a master design verification plan that contains a hierarchy of requirements and reuse it in different design verification projects. For example, you can maintain the master verification plan in a spreadsheet. You can then import this plan in different verification projects managed through Verifier, with or without variations. In a top-down verification flow, defining or reusing the verification plan is the first stage.

Verifier lets you import requirements from the following sources:

- Spreadsheet (`.xlsx`, `.XLSX`, `.xlsm`,`.XLSM`)
- Comma-separated values file (`.csv`, `.CSV`)
- Cellview of the type `verifier`

You can import requirements from an Excel file saved in the format `.xlsx`,`.XLSX`, `.xlsm`, or `.XLSM`. Files that are saved in the `.xls` or `.XLS` format are not supported.

For details on working with requirements imported from CSV or Excel file, see Copy Requirements from File and Important Points to Note when Importing Excel Files.

For Verifier cellview source, you can import requirements into a Verifier session by copying or referencing them from the source. Import requirements by copying them when you want to edit the requirements in Verifier. Import requirements by referencing them when you want to maintain the source as the requirements master. In this case, Verifier displays the referenced requirements in read-only mode. To change requirements imported by reference, you must edit the source. Verifier displays the reference source filename as the topmost node.

For details on working with the requirements copied from the imported cellview, see Importing Requirements from a Verifier Cellview.

For details on working with multiple referenced cellviews, see External References in Virtuoso ADE Verifier.

For information on editing and deleting referenced requirements, see Flow for Referencing External Cellviews and Deleting Requirements, respectively.

Related Topics

Adding Requirements

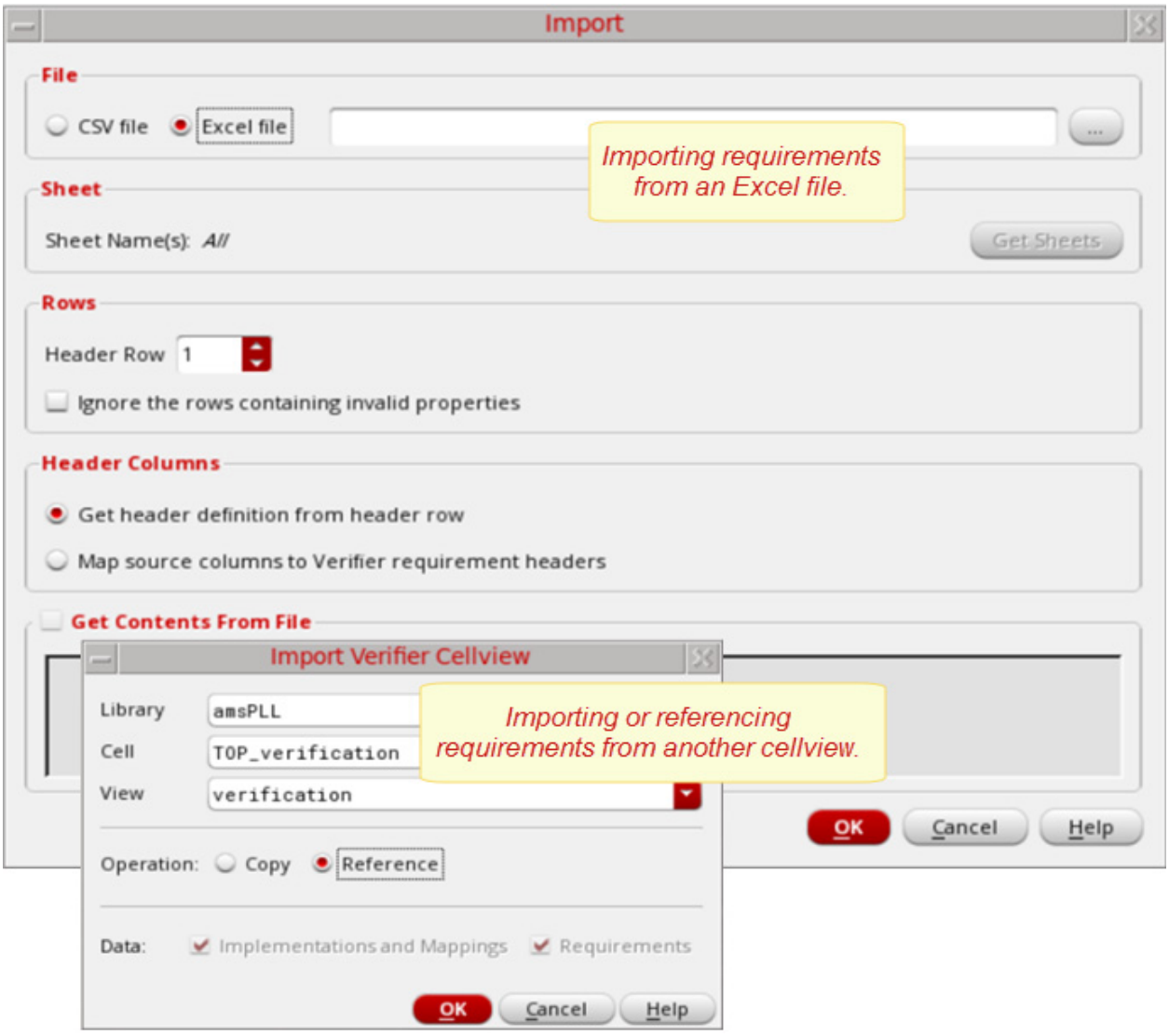
© 2023 All Rights Reserved.

Importing Requirements from File

Importing Requirements from a Verifier Cellview

Importing Requirements from File

You can use the Import form to import requirements from a CSV file or a Microsoft Excel file. The fields in this form depend on the type of the requirements source.



© 2023
All Rights Reserved.

To import requirements into an existing Verifier session:

1. Start the import process.

   - To start importing requirements in the opened Verifier session, choose *File – Import – File*.
   - To start the import process in a new Verifier session:

     a. Click *Tools – ADE Verifier* from Virtuoso CIW.

        The Choose Verifier Cellview Form form is displayed.

     b. Specify a *Cell* and a new *View*.

     c. Click *Import Requirements*.

   The Import Form is displayed.

2. Select the source type.

   - To import the requirements stored in a `.csv` or `.CSV` file, click *CSV file*.
   - To import the requirements stored in a Microsoft Excel `.xlsx`,`.XLSX`, `.xlsm`, or `.xlsm` file, click *Excel file*. For more information, see Important Points to Note when Importing Excel Files.

3. Specify the source.

   For example, specify the location and filename in the *File Name* field.

   Excel file names can only contain the following characters:
   `a-z,A-Z,0-9,!,@,#,$,%,&,(,),-,_,+,=,[,],.,,, and space.`

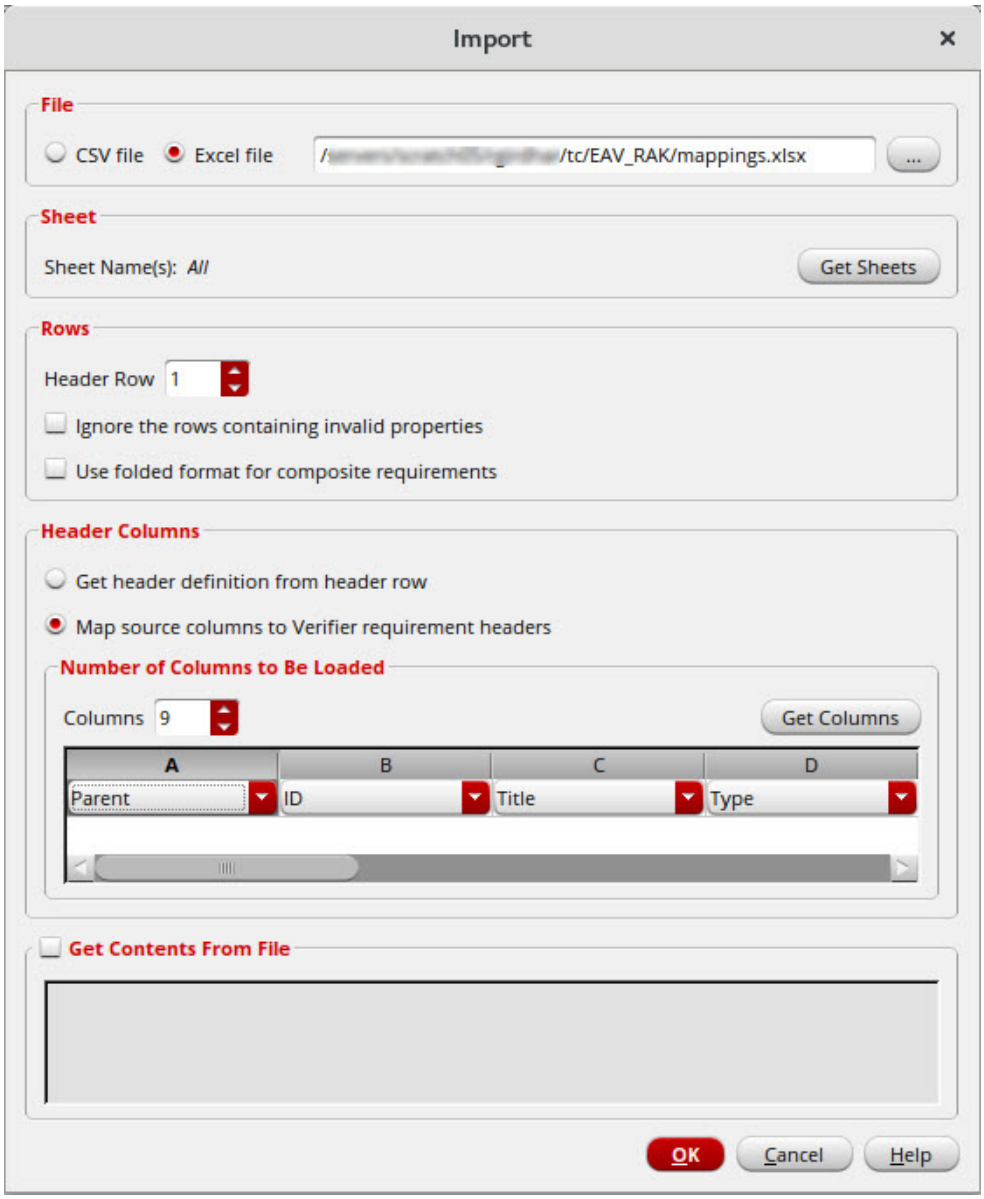
4. If your source is a Microsoft Excel file with multiple sheets, specify the sheet that contains the requirements you want to import. For this, click *Get Sheets* and select one or more sheet names or *All*, as required.

   Ensure the following:

   - All the blank spaces are replaced with underscore in the sheet name.
   - A unique requirement ID without spaces is created for file or sheet.
   - All the sheets are included in the group box.
   - By default, all the sheets in a file are selected. Click *Select All* to select all the sheets and *Clear All* to clear selection of all the sheets.

© 2023 All Rights Reserved.

- If all the sheets are selected, `All` is shown with *Sheet Name(s)*.



5. Select the *Ignore the rows containing invalid properties* check box to ignore all the errors while importing from an Excel or CSV file. By default, it is deselected.
6. Set the *Header Columns* group box as follows:

© 2023
All Rights Reserved.

To use the requirement column definitions from the source file, click *Get header definition from row* and specify the header row number. Ensure that the headers in the source use the following text. The mandatory header is `Title`.

```
Parent,ID,Title,Type,MinSpec,MaxSpec,Unit,VerificationSpace,Owner,CellviewH
older,Scope,Domain,Deadline,Description
```

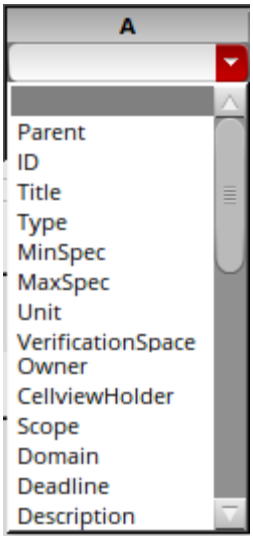
If an invalid value is specified for these columns, the corresponding row of specifications is not imported as a requirement and a message containing the recommended format for these values is displayed.

7. Map the source columns to the Verifier requirement columns by doing the following.

   a. Click *Map source columns to Verifier requirement headers*.

   b. In the *Number of Columns to be Loaded* group box, specify the number of columns you want to import. The column mapping table in the *Header Columns* group box updates accordingly.

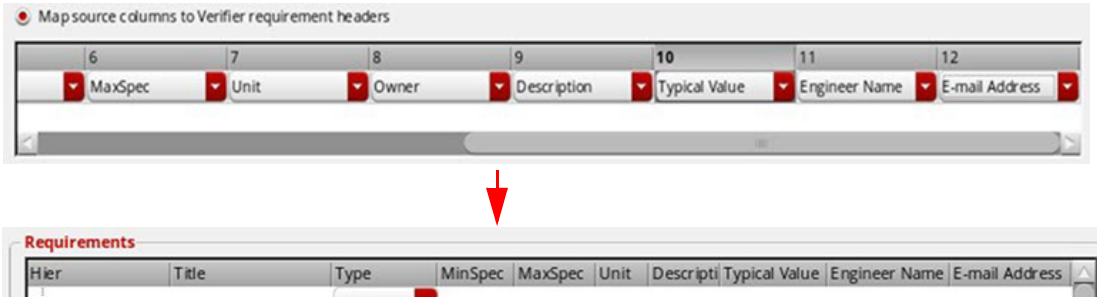
   c. Map the columns in the source file to the Verifier requirement columns. You can change the sequence of the columns based on the sequence in which they exist in the source file. For example, if the third column in the source file is the description instead of the title, select *Description* under *3*.



You can merge multiple cells by assigning the same target to multiple columns. For example, if the columns 10 and 11 of the file contain information that you want to add as the description of the imported requirements, assign `Description` to both these columns.

© 2023 All Rights Reserved.

- The hierarchy number (*Hier*) is auto-assigned by Verifier and is not part of the import process.
- You can import multiple custom fields from Excel and CSV files.The custom fields are added to the files by using the environment variable. When the file is imported, the custom requirement fields are displayed in the drop-down list boxes in the *Header Columns* group box. For details, refer to Custom Fields.



8. Select *Get Contents From File* to view some rows in the source file to ensure that you have specified the correct file and mapping details. The number of rows that will be shown in this field is *Top rows* + `1`.
9. Click *OK*.

The requirements are imported in the Verifier session.

You can choose to load requirements and implementations from a cellview of the type `verifier`. For this, choose *File – Import – Verifier Cellview*, and specify the other required details, including the library, cell and view. You can also right-click on a requirement of the type *Note* and select the shortcut menu option *Import Cellview*, to add the cellview below the selected requirement in the hierarchy. For details, see Importing Implementations from Another ADE Verifier Cellview.

Important Points to Note when Importing Excel Files

- You can import requirements from an Excel file saved in the format `.xlsx`, `.XLSX`, `.xlsm`, or `.XLSM`. The `.xls` or `.XLS` file format is not supported.
- In addition to plain text, you can import UTF-8/Unicode characters from an Excel file, including text with symbols outside the ASCII format, and text in different languages, such as Chinese, Japanese, and Korean.
- You can manually hide the columns of the source Excel file. However, to hide a column in Verifier, it should meet the following conditions:
  - The column is hidden in the source Excel sheet.

© 2023 All Rights Reserved.

- The column is not mapped to any Verifier requirement header yet.

  Verifier gets the real column count and hidden column count from the first selected sheet. After obtaining the information, set the current or maximum value of the Columns drop-down list box to the real column count to ensure that the hidden columns are not displayed in the header columns table.

- If you import an Excel file that contains merged cells, the details are imported into individual cells, where each cell contains the same value as the merged cell. For example, if the file contains two cells merged into one cell and contain the value <*any_value*>, after import, the merged cell is imported as two individual cells, where each cell contains the value <*any_value*>. This is especially useful when you want to import a large number of requirements. For example, you can merge the Owner column with a value and import the requirements. After import, the correct owner will be set for each requirement.

Related Topics

Import Requirements from External Sources

Importing Requirements from a Verifier Cellview

Adding Requirements

Adding a Requirement Manually

Importing Requirements from a Verifier Cellview

You can use the Import Verifier Cellview form to import requirements from a Verifier cellview.

To import requirements into a Verifier session:

1. Start the import process.

   To start importing requirements in the opened Verifier session, choose *File – Import – Verifier Cellview* or choose the shortcut menu option *Import Cellview...* in the requirement hierarchy.

   The Import Verifier Cellview Form form opens.

2. Specify the details of the Verifier Cellview to import.

   - If you specify valid details for a Verifier cellview, the Verifier session imports the cellview as a *Copy* or external *Reference*.

© 2023 All Rights Reserved.

- If you specify invalid details for the Verifier Cellview, a hierarchical node is created in the *Requirements* pane with an error icon.

For details, see Importing Implementations from Another ADE Verifier Cellview.

3. Specify the import *Operation*.

Note: The *Add Implementations* command opens the Add Implementation Cellviews Form form and the *Import Requirements* command displays the Import Form form.

Related Topics

Import Requirements from External Sources

Importing Requirements from File

Adding Requirements

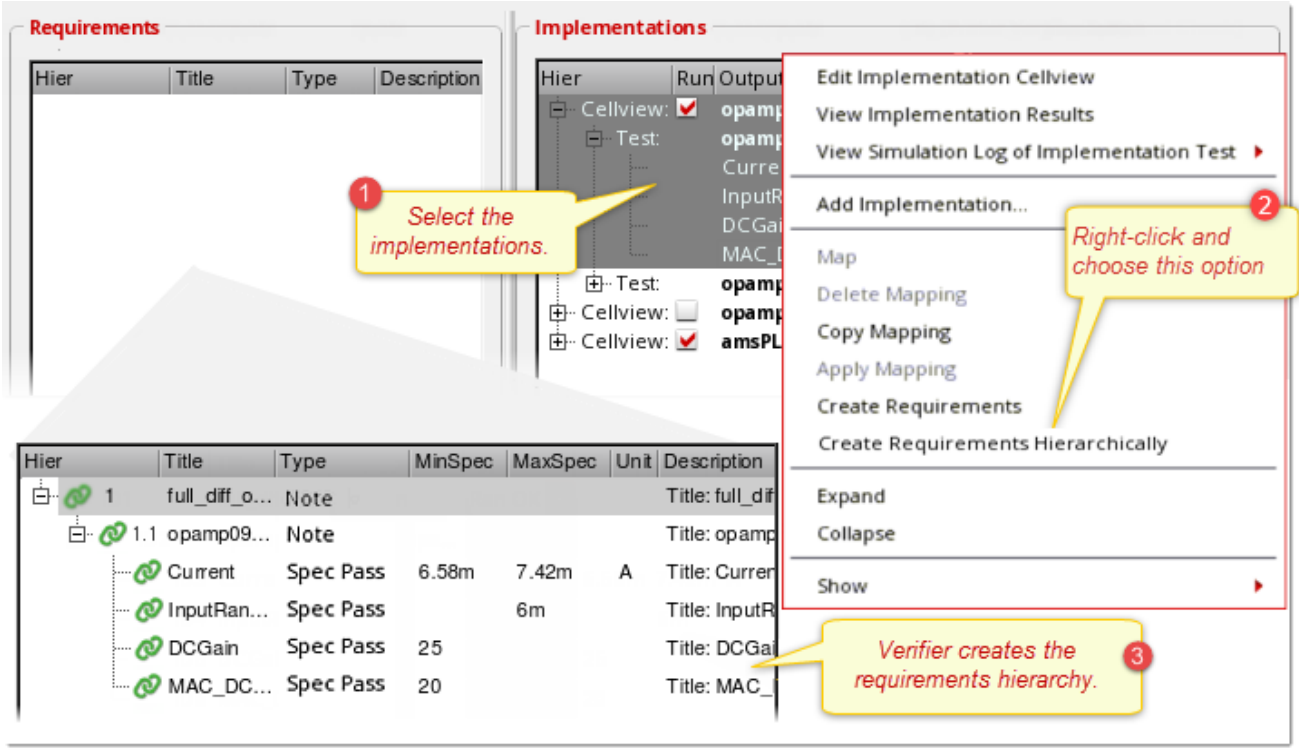
Adding a Requirement Manually

Creating Requirements from Implementations

If you do not have a requirement plan, but have included implementations in Verifier, you can create a requirement plan corresponding to the implementations. Verifier also maps the new requirements to the corresponding implementations.

© 2023 All Rights Reserved.

For details on implementations, see Implementations and Mappings in Virtuoso ADE Verifier



To create requirements from implementations:

1. In the *Setup* tab, select the implementations.
2. Right-click and do one of the following:
   - ❑ Choose *Create Requirements* to create a single requirement corresponding to the selected implementation specifications.
   - ❑ Choose *Create Requirements Hierarchically* to create a requirements hierarchy that is identical to the hierarchy of the selected implementation.

Verifier generates the respective requirement or requirements hierarchy and maps the requirements to their corresponding implementation. You can edit the requirements, as needed.

Additional Information

- Verifier creates requirements only for unmapped implementations. The shortcut menu command *Create Requirements Hierarchically* ignores the specifications hierarchy of mapped implementations.

© 2023
All Rights Reserved.

- The hierarchical number of the requirements are auto-generated based on their hierarchy levels.
- The title of the requirements indicate the cellview name, test name, or output name of the corresponding implementation. In the requirement titles, '/' replaces '_' and ' ' is used as a separator.
- The default type of the requirements of cell and tests is *Ran OK* and the default type of the requirements of outputs is *Spec Pass*.
- You can create *Ran OK* requirements for a required subset of an implementation, run, reliability, test, by setting the autoCreateRanOkRequirement environment variable to the only the required set. For example, to create hierarchical requirements only for the implementation, run, and test, set `autoCreateRanOkRequirement` to "Implementation, Run,Test".
- Verifier adds the implementation specifications to the requirements.

Related Topics

Adding Requirements

Import Requirements from External Sources

Importing Requirements from File

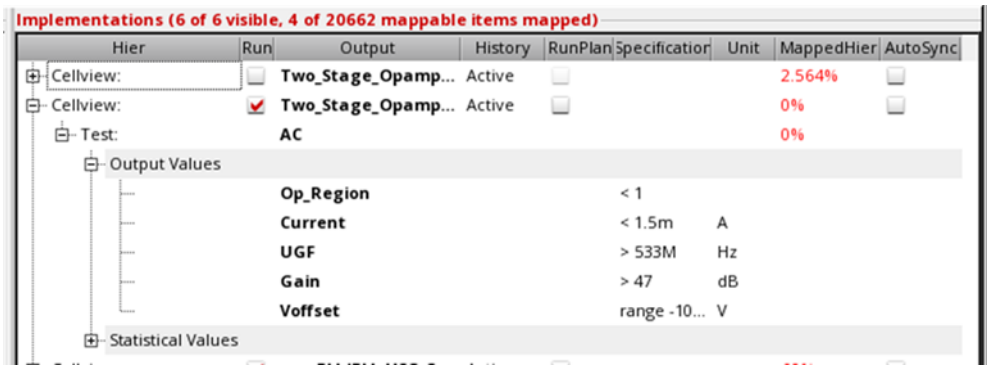
Importing Requirements from a Verifier Cellview

Simulation Runs Using Auto-Synced Requirements

You can use ADE Verifier only as a regression tool for running simulations. If you want to run simulations using implementations, without the need to manually create requirements and mappings, you can display the *AutoSync* column from the header area in the

© 2023 All Rights Reserved.

Implementations pane. This column is hidden by default. After the column is displayed, you need to select the *AutoSync* check box for the selected implementation.



Important

AutoSync does not support multiple mappings to requirements or referenced implementations. It also ignores the settings specified by the autoCreateRanOkRequirement environment variable.

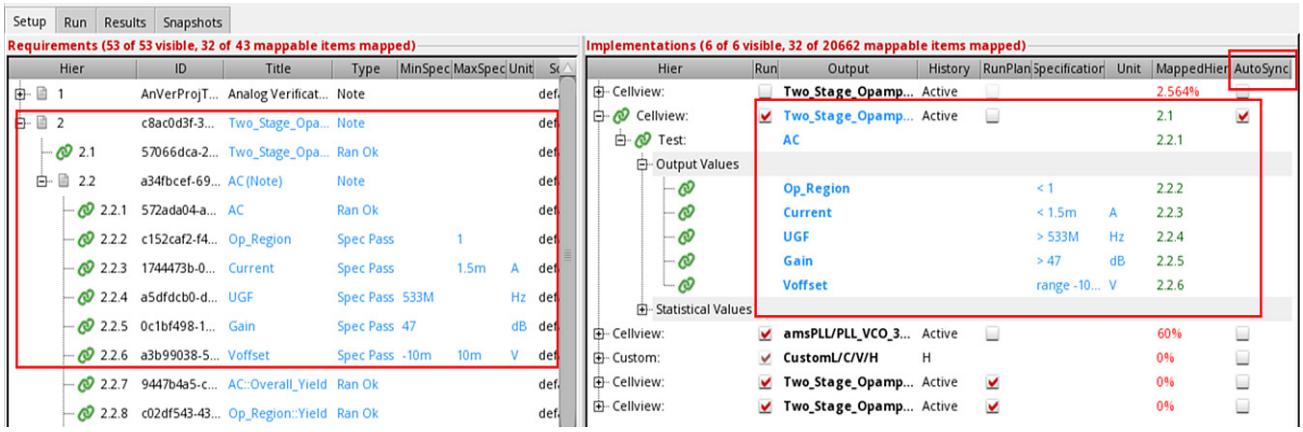
When you select the *AutoSync* check box for the first time, the implementation is updated with the setup data from the `maestro` cellview. Additionally, requirements and mappings are created automatically for the selected implementation and displayed in blue text, as long as the *AutoSync* check box is selected. The requirements hierarchy created is identical to the implementation hierarchy.

Whenever there is a change in the `maestro` view, the implementation is automatically updated to match these changes, and then the requirements are updated as well. The requirements and mappings remain in sync with the implementation outputs. Updates in an implementation are triggered when you do the following:

- Open the Verifier cellview
- Change the run mode in ADE Assembler from a previous value to *Single Run, Sweeps and Corners*, or *Monte Carlo Sampling*

© 2023
All Rights Reserved.

- Add, delete, or modify a run, reliability analysis, test, or output in ADE Assembler

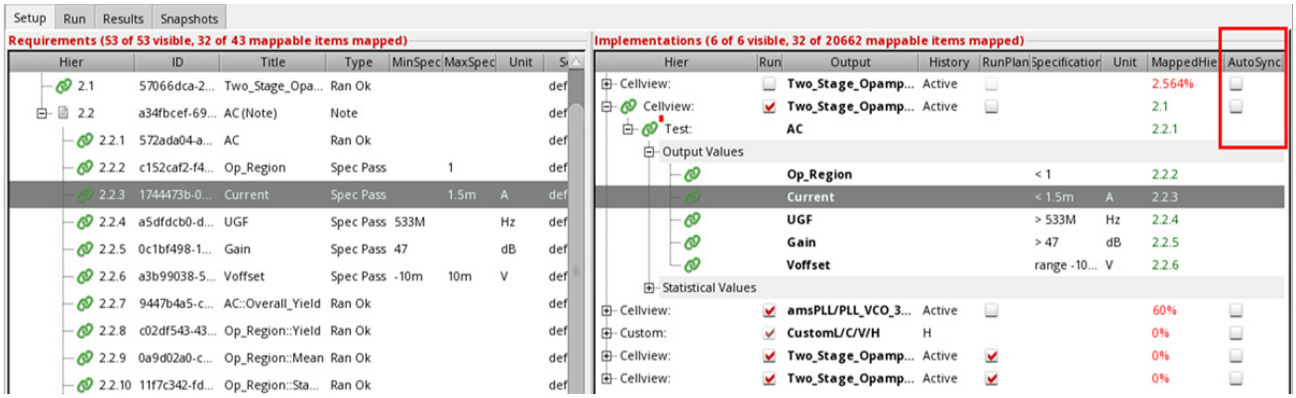


Except the top auto-synced requirement, which is of type *Note*, you cannot move the requirements up or down in the hierarchy. Other modifications, such as changes to mappings or requirements through the GUI are also not available.

The values in the *Title*, *Type*, *MinSpec*, *MaxSpec*, and *Unit* fields of the *Requirements* pane, and in the *Output*, *Specifications*, and *Unit* fields of the *Implementations* pane, are read-only and displayed in blue text. All other fields of the auto-synchronized requirements and implementation are editable.

You cannot move other items into the hierarchy displayed in blue text. For example, if you drag any other requirement which appears in black text, you cannot drop it onto the auto-synchronized section. Specifically, the requirements in blue text cannot be mixed or matched with the usual requirements.

If you deselect the *AutoSync* check box, the synchronization between the requirements and the implementations stops. In such a case, the requirements hierarchy and the mappings that were created for the implementation stay the same but become editable.



© 2023
All Rights Reserved.

Any changes that you make to the requirements or mappings persist unless you select the *AutoSync* check box which removes all changes and synchronizes the requirements hierarchy again.

When you select the *AutoSync* check box, the simulation results might show the results status as out-of-date. This happens because ADE Assembler runs simulations in editable mode, and Verifier picks up the writable history, thereby considering the cellview as changed. With *AutoSync* selected, Verifier reloads the implementation, which again marks the results as out-of-date. Typically, it is recommended to set `disableReadOnly` to its default value `nil`.

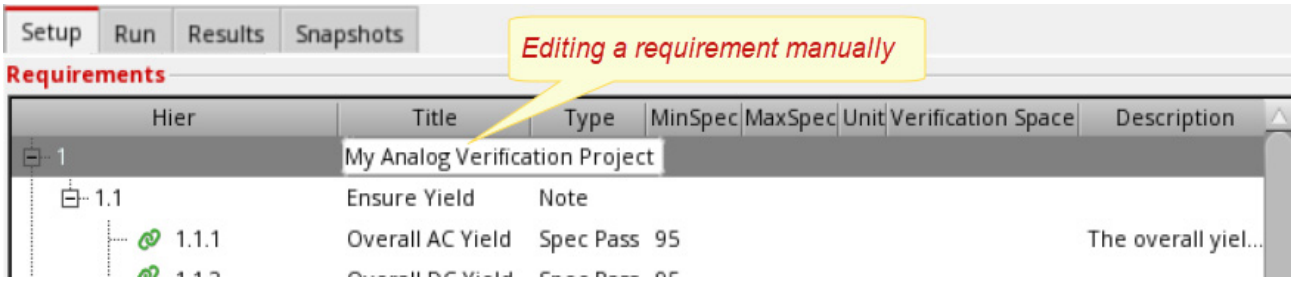
To ignore specific outputs when creating auto-synced requirements, you can use the Create requirements for outputs with no specification values preference option or the `autoSyncOutputFilter` environment variable. You can use either one or both of these at the same time. For example, to create auto-synced requirements for all outputs that start with `X`, set the `autoSyncOutputFilter` environment variable using the required regular expression string.

Related Topics

autoSyncOutputFilter (Environment variable)

Editing Requirements in the Requirements Hierarchy

You can edit a local requirement directly in the requirements hierarchy, except the *Description* field. The following figure illustrates how you edit a requirement in the hierarchy.



To edit a requirement directly in the requirements hierarchy

1. Click the *Setup* tab and select the requirement.
2. To edit the ID, title, and specifications, click the field and change its value.
3. To change the type, select your option from the drop-down list.

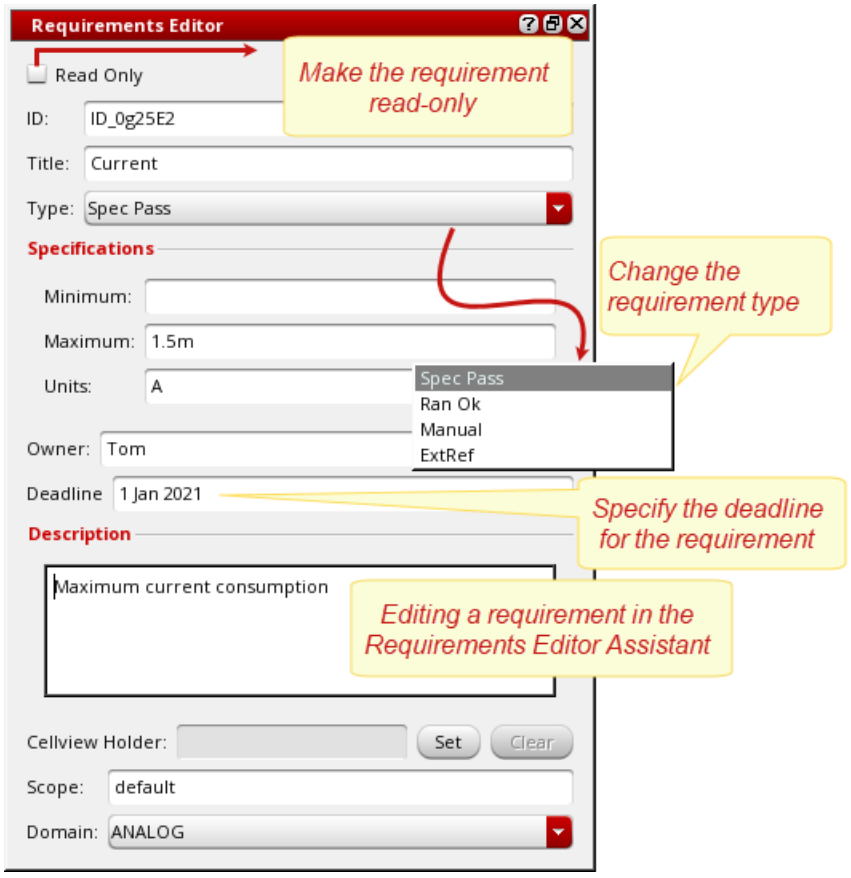
© 2023
All Rights Reserved.

4. To make the requirement read-only, select the *Read Only* check box.
5. To make the requirement editable again, deselect the *Read Only* check box. You cannot change the hierarchical number.

Verifier updates the requirement as you edit it.

Editing Requirements Using the Requirements Editor Assistant

If you want to update the description or other details of a requirement, choose to edit it in the Requirements Editor assistant. You can edit all the fields in the Requirement Editor Assistant.



To edit a requirement using the Requirement Editor Assistant:

1. Open the Requirements Editor Assistant.

   The assistant appears in the right pane.

© 2023
All Rights Reserved.

2. Select a requirement in the *Requirements* pane.
3. Edit the requirement details using the assistant.

To close the requirement editor, see Requirements Editor Assistant.

You can choose to display the Requirements Editor Assistant from the *Edit* menu, *Window* menu, and the Verifier toolbar, under the *Setup* and *Results* tabs. It is not required under the *Run* tab.

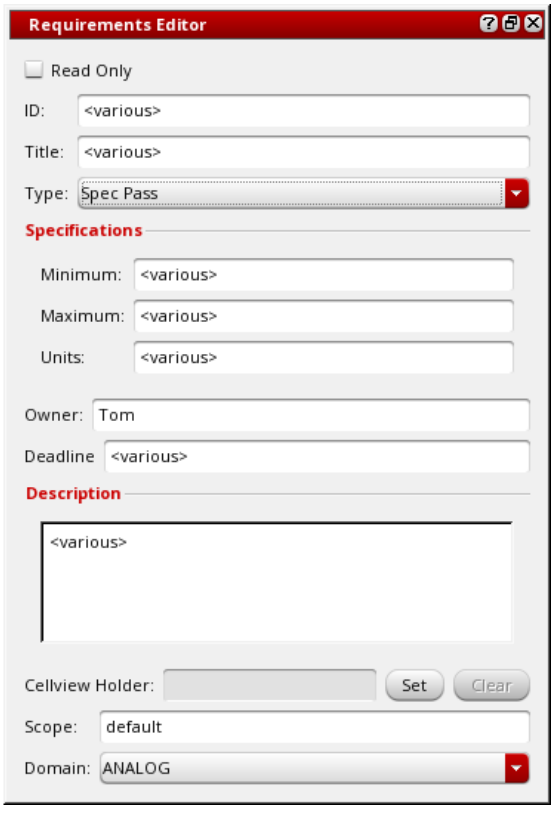
To edit multiple requirements using the Requirement Editor Assistant:

1. Open the Requirements Editor Assistant.

   The assistant displays in the right pane.

2. Select multiple requirements in the *Requirements* pane while keeping the *Shift* or *Ctrl* key pressed.

   The fields in the Requirements Editor display *<various>* as shown below for all requirement properties that have different values:



3. Edit the requirement details using the assistant.

© 2023
All Rights Reserved.

All selected requirements are updated with the values that you specify in the assistant. The values of those fields that display *<various>* are not changed.

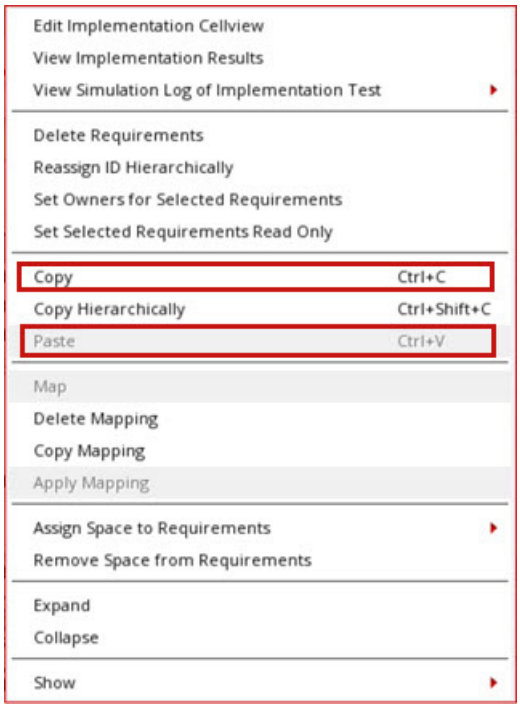
To close the requirement editor, see Requirements Editor Assistant.

Copying Requirements in the Hierarchy

You can copy the requirements as a single entry or hierarchically and paste them using multiple ways in the requirements hierarchy.

To copy and paste requirements, do one of the following:

- Use the shortcut menu commands to copy and paste at the desired location.



- Use the bindkeys `Ctrl+C` (Copy), and `Ctrl+Shift+C` (Copy Hierarchically) to copy from, and `Ctrl+V` (Paste) to paste requirements.

When read-only requirements are copied, the read-only feature remains intact while pasting. However, in all cases, new IDs are assigned when pasting the requirements.

© 2023 All Rights Reserved.

Copy Requirements from File

While importing requirements from a CSV or an Excel file, you can copy the requirements from the source file, but the information about the reference source is retained. You can compare to check if the source information and copied information are in sync. If the information is not in sync, you can merge the requirements so that requirements from the source file can be added. This flow reduces the risk of purely referencing an external file, such as renaming, corruption, or unreadability of the source file.

The requirements in the CSV or Excel file are compared with the requirements in Verifier to list the *Different*, *Only*, *Same*, or *Ignore* requirements in a tree:

- **Different** indicates the requirements in both the source file and Verifier that have different properties;
- **Only** indicates the requirements only in the source file;
- **Same** indicates the requirements in the source file and Verifier with same properties;
- **Ignore** indicates the requirements in the source file but without ID or the specified properties of the requirement contain invalid data.

When you merge the requirements in CSV or Excel to Verifier, the following happens:

Different requirements are updated, **Only** are added in the cellview, and **Same** are kept as is.

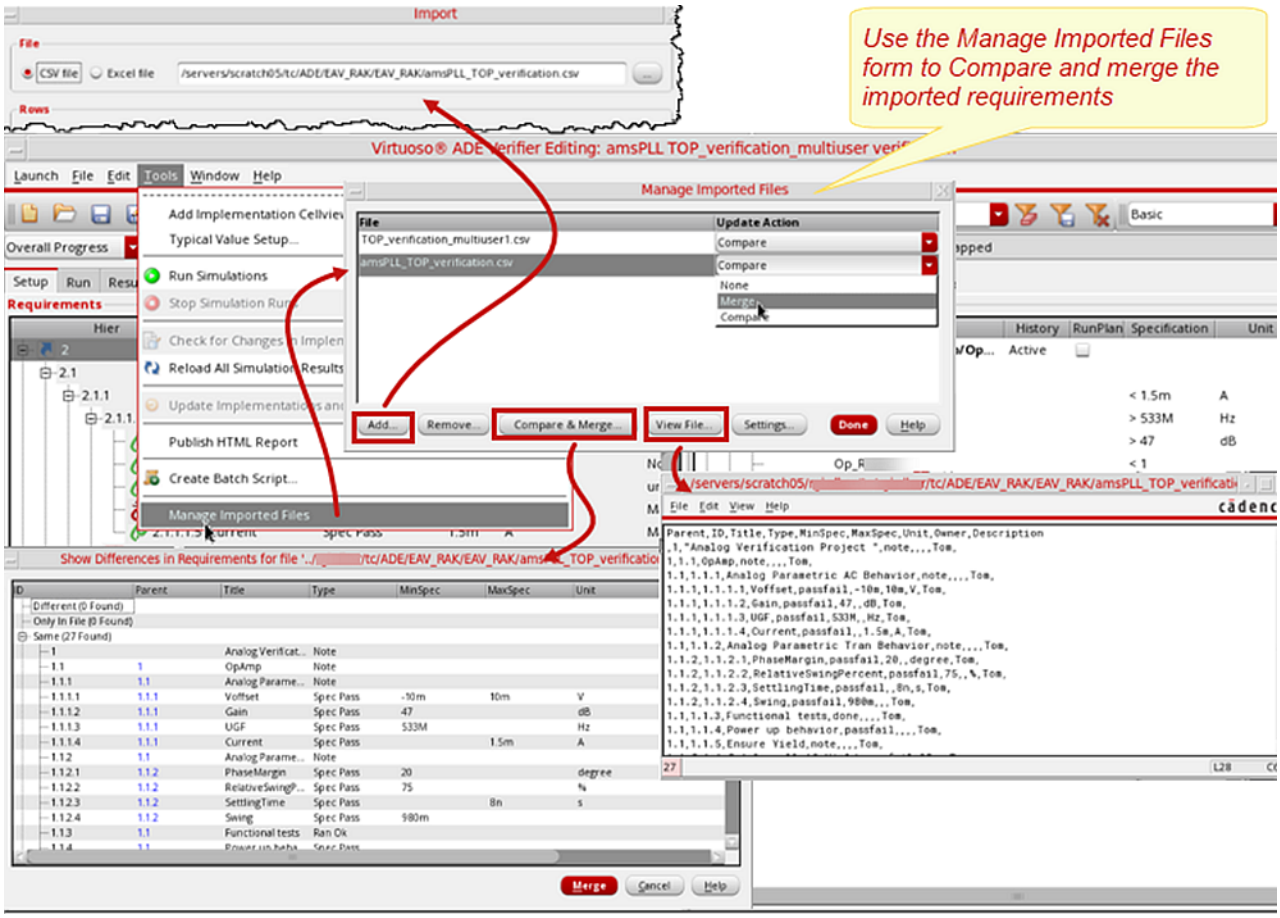
Important

The properties of requirements, which are found only in Verifier without corresponding columns in the source CSV or Excel file, are ignored during the Compare & Merge flow.

The extra top-level requirements of type *Note* for the imported file or data sheets are ignored while comparing and merging the requirements. Click *Cancel* on the Show Differences in Requirements for File form to ignore the differences and keep the requirement as it is.

© 2023 All Rights Reserved.

The following figure illustrates how you manage the requirements imported from a file.



To edit a copied requirement in the hierarchy:

1. Right-click a referenced requirement and click *Open Reference*.
2. Edit the requirement in the source.
3. Save the source.

If your source is an Excel file, edit and save the changes outside Verifier.

To import requirements from a file:

1. Start a new Verifier session.
2. Open the Import form using one of these:
   - *File – Import – File*
   - *Tools – Manage Imported Files – Add*

© 2023 All Rights Reserved.

3. Set the options in the Import form.
4. Click *OK* to import the data from the source file as local requirements.

Managing Imported Files

To manage imported files, you can perform the following tasks:

Add

- Click *Tools – Manage Imported Files* to open the form.
- Click *Add* in the *Manage Imported Files* form to add a new imported file.

Remove

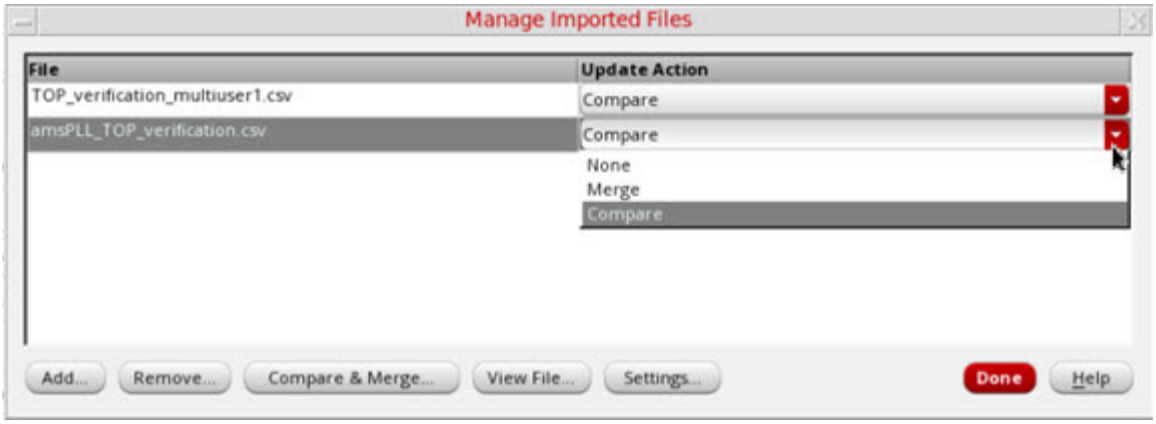
- Select a file item.
- Click *Remove* in the *Manage Imported Files* form to delete the selected imported file.

Other Tasks

- Select a file item and click *Settings* in the *Manage Imported Files* form to edit the setup data for the imported file.
- Select a CSV file item and click *View* in the *Manage Imported Files* form to open the file in a dialog box.
- Select the required option from the following three options for automated tasks in Verifier by using the *Auto* drop-down list box in the Manage Imported Files form.
  - *None*: Nothing is being done during the setup sync.
  - *Merge*: Automatically merges any changes to the requirements in the file to Verifier.

© 2023 All Rights Reserved.

- *Compare*: Compares the requirements automatically in the file with the requirements in Verifier. The differences are reported, if any. You are prompted to use *Compare & Merge*.



Click *Merge* in the Show Differences in Requirements for File form to merge all the selected requirements from the source file to Verifier.

Setting Up Requirement Owners

When you add a requirement manually, Verifier sets its default owner as the current Verifier user. When you include requirements from a CSV or Microsoft Excel file, Verifier sets the owners as specified in the source file. For details, see Adding Requirements.

The *Owner* column is hidden by default. To display this column, click the *Show* button for the requirements and select *Owners*.

To specify the owner of a requirement, do one of the following

- Right-click the requirement, choose *Set Owners for Selected Requirements*, type the owner, and click *OK*.

  You can use this method to set the same owners for multiple selected requirements.

- Edit the owner name directly in the requirements hierarchy.

- Display the requirement editor and specify the owner in the *Owner* field.

  For details on editing the name in the hierarchy and using the requirement editor, see Editing Requirements in the Requirements Hierarchy and Editing Requirements Using the Requirements Editor Assistant.

© 2023 All Rights Reserved.

Consider the following important points when you assign the owners of the requirements in the master cellview.

- The default owner of a new top-level requirement is the current Verifier user name. The default owner of a new sub-level requirement is the same as the owner of its parent requirement.
- You can change the current Verifier user name, which is used as the default owner name of new high-level requirements. For this, choose *Edit – Preferences,* type the user name in the *Current User Name* field, and click *OK*.

  When you add a new top-level requirement in the master cellview manually, Verifier leaves the current user name as blank.
- All requirement owners can map their requirements with implementations in their local cellviews only.

Related Topics

Set Requirement Owners Form

Organizing the Requirements Hierarchy

Verifier lets you organize requirements in a hierarchy for improved usability. You can change the hierarchy level of a local requirement, and move it up or down the hierarchy. If you want to change the hierarchy of referenced requirements, modify the source.

When you change the hierarchy, Verifier updates the hierarchical number of the requirements to reflect the current hierarchy.

To rearrange a local requirement in the hierarchy:

➡ Change the hierarchical order of the requirement.

- ❑ Select the requirement and click *Decrease Indent of selected items (Shift + Left)* or *Increase Indent of selected items (Shift + Right)*, as required.
- ❑ Drag the requirement to its new hierarchy level.

➡ Change the sequential order of the requirement:

- ❑ Select the requirement and click *Move selected items up in the tree (Shift + Up)* or *Move selected items down in the tree (Shift + Down)*, as required.
- ❑ Drag the requirement to its new sequential order.

© 2023 All Rights Reserved.

Reassigning IDs Hierarchically

As a project manager, you may want to rename your requirement ID. This ensures that the IDs are easily identifiable and unique.

Verifier lets you rename the requirement IDs when you choose *Reassign ID Hierarchically* from the shortcut menu. *Reassign ID Hierarchically* changes the name of selected ID and its child requirement ID names to random unique IDs.

To rename the requirements ID in the hierarchy:

➥ Right-click the requirement and choose *Reassign ID Hierarchically*.

The selected requirement and the corresponding child requirements are renamed hierarchically and the mapping information is retained.

- *Reassign ID Hierarchically* is shown only when the selected requirement is editable. This is not available for referenced requirements.
- To retain the mapping information, ensure that the preference option *Copy mapping to renamed requirement* is enabled. By default, this is enabled.

Exporting Requirements

Verifier lets you export the requirements hierarchy in the current Verifier session to a new CSV or Excel file. You can store the exported requirements hierarchy for future reference and reuse it in other verification projects.

Customized requirement fields can also be exported to the destination files. The customized requirement field names should not be same as the available field names in the destination files.

You can also export the mappings and results for requirements in the current Verifier session to imported files.

The following properties can be exported from a Verifier session:

```
Parent,ID,Title,Type,MinSpec,MaxSpec,Unit,Owner,Scope,Domain,Description,Deadline
,Overall Status,Mapping,History,Result Data Age,Min Value,Typical Value,Max
Value,Passed,Failed,No Results,Unmapped,Verification
Space,Coverage,CellviewHolder,Disabled,FailCoverage,Hier,NoneCoverage,PassCoverag
e,ReadOnly,ReadOnly,Signoff,StatusDescription,Yield,Min,Max,Mean,StandardDeviatio
n,SigmaToTarget,Cpk
```


© 2023 All Rights Reserved.

The `StatusDescription` properties in the exported file contains the tooltip information that is displayed on the *OverallStatus* column of the *Results* tab.

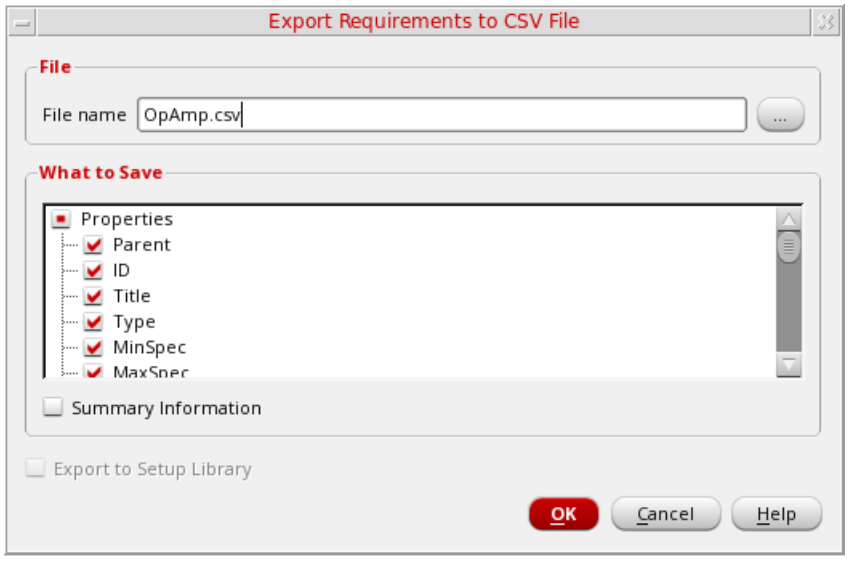
The following snippet shows the format used in Verifier to store the fields and their order in a CSV file. The snippet includes the fields and an example requirement.

```
Parent,ID,Title,Type,MinSpec,MaxSpec,Unit,Owner,Description,Hierarchy,Overall
Status,Mapping,History,Result Data Age,Min Value,Typical Value,Max
Value,Passed,Failed,No Results,Unmapped,Verification Space,Coverage,
AnVerProjTop,"Analog Verification Project ",note,,,,Tom,"This is the new project
""Analog Verification Project"" for customer XYZ. It is based on project
...",52.63%,,,,,,,,,,,,
```

To export the requirements hierarchy:

1. Choose *File – Export* and select from the following formats of exporting requirements:
   - *CSV*

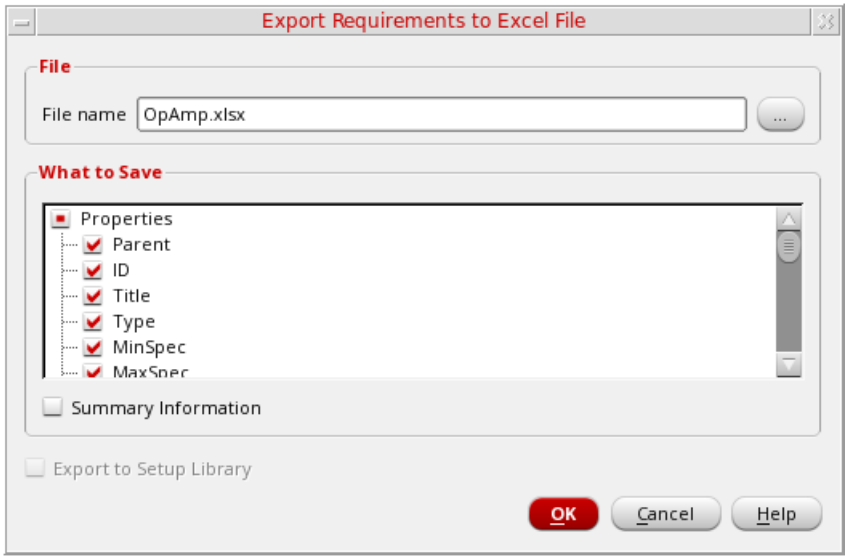
     The Export Requirements to CSV File form appears.



© 2023
All Rights Reserved.

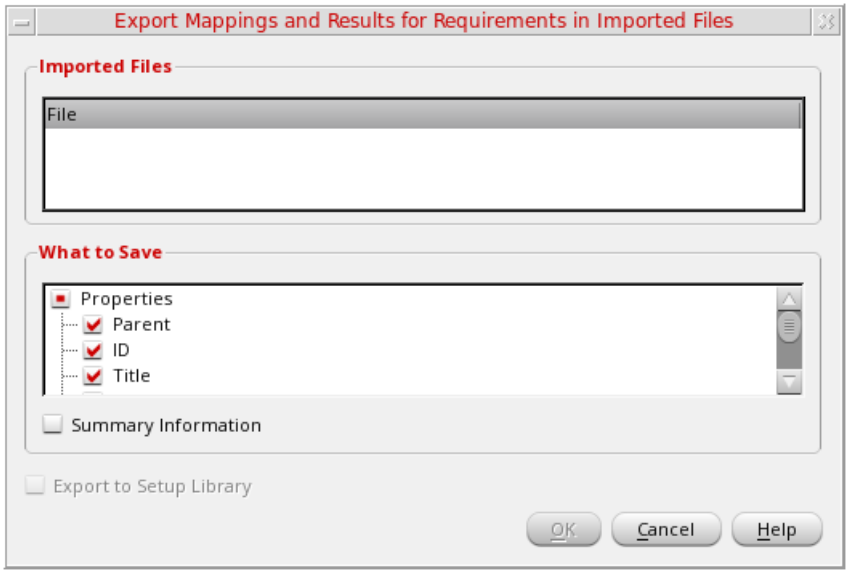
- *Excel*

  The Export Requirements to Excel File form appears.



- *Imported File*

  The Export Mappings and Results for Requirements in Imported Files form appears.



  The available options let you export the same values as the *Results* page in the Verifier user interface.

2. Specify the name and location of the destination file.

© 2023
All Rights Reserved.

If the specified directory does not exist, it is created automatically.

Excel file names can only contain the following characters:
`a-z,A-Z,0-9,!,@,#,$,%,&,(,),-,_,+,=,[,],.,,, and space.`

3. Select from the *What to Save* options.

   The form lets you reorder the exported fields. You can drag the *What to Save* options and drop them at the desired location in the displayed list.

4. Click *OK*.

The selected hierarchy is stored in the specified file.

Choosing *Summary Information* in these forms adds the following details in the exported file, as shown below. This information is added above the field information.

```
General Information:,Header Row,7,,,,,,,,,,,,,,,,,,
,Author,Tom,,,,,,,,,,,,,,,,,,
,Creation Date,Jun 2 11:13:39 2018,,,,,,,,,,,,,,,,,,
,Results Summary,Passed(12/18);Failed(1/18);No Results(0/18);Unmapped(5/
18),,,,,,,,,,,,,,,,,,
,,,,,,,,,,,,,,,,,,,,
,,,,,,,,,,,,,,,,,,,,
```

When you save your cellview after exporting requirements, the selected fields are saved under the `customizedExportFields` tag in the `settings.v3` file. For any subsequent export operation using any of the Export forms, the fields are pre-filled with the settings saved in the settings.v3 file.

Additionally, you can set default settings for exporting a set of fields. For more information, see the environment variables fields and addSummaryInformation.

Deleting Requirements

You can delete the local requirements that are not needed in your verification plan. When you delete a requirement mapped to an implementation, the mapping is removed.

To delete a local requirement, do one of the following:

- Right-click the requirement and choose *Delete Requirements*.
- Select the requirement and press `Delete`.

© 2023 All Rights Reserved.

You can select multiple requirements for deletion. Select a parent requirement if you want to delete it and its child requirements.

The selected requirements are removed.

When you choose to delete requirements, Verifier deletes the selected requirements and their associated mappings. It also updates the hierarchical number of the requirements to represent the new hierarchical structure.

Filters in the Requirements Hierarchy

It is possible that you have a large requirements hierarchy, where locating specific requirements is time-consuming. You can add a filter row in the requirements hierarchy to display only specific types of requirements. The filter row is applied to both, the requirements hierarchy and the implementations list.

You can also locate requirements using the filter functionality of Verifier.

Related Topics

Filtering Requirements Based on Text Criteria

Filtering Requirements Based on Status

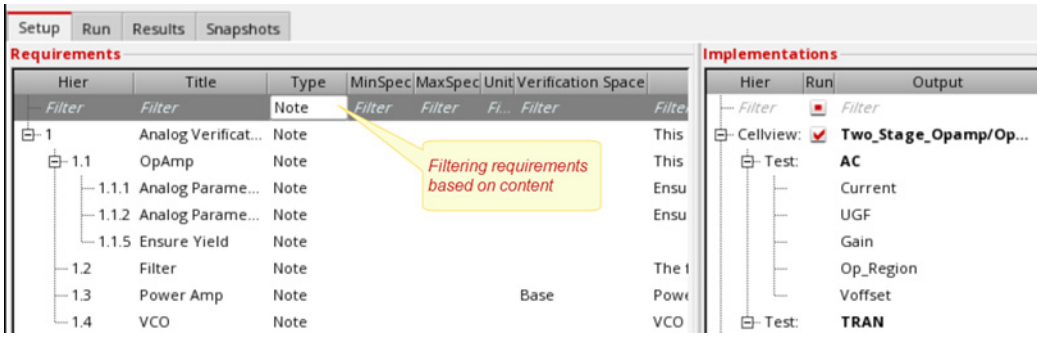
Filtering Requirements Based on Text Criteria

To search requirements with specific text in the hierarchy:

1. Do one of the following:
   - Click the *Show Filter Row* icon on the toolbar.
   - Right-click anywhere on the column heading row hierarchy and choose *Filter row*.

© 2023 All Rights Reserved.

The Filter row is displayed below the column headings in the *Setup*, *Run* and *Results* tab.



2. Type the filter string or number in the required hierarchy column.

As you type, the filter functionality searches the selected field of the requirements or implementations and displays the results in the requirements pane. You can also use the shortcut menu options and choose criteria such as *All of the words*, *Starts with*, *Case Insensitive*, and more.

If you filter the requirements with the *Deadline* property, it is recommended that you specify the same deadline for a parent requirement and its child requirements. Specifying conflicting deadlines will display misleading filtered requirements.

To use the other features of the filter functionality:

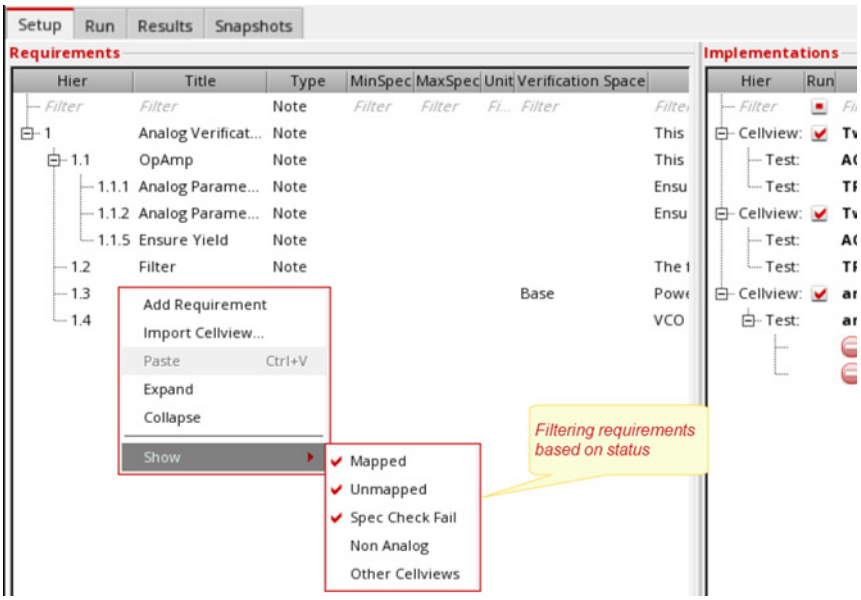
- Enter a name for the filter and click *Save Filter* to store the search string for later use in the session.
- Click the *Clear the contents of the filter icon on the toolbar,* to remove all filters and show all requirements or implementations.
- Click *Delete Filter* to delete a saved filter.

Filtering Requirements Based on Status

To filter the requirements and implementations based on their status in the hierarchy:

© 2023
All Rights Reserved.

1. Right-click the requirements area and choose *Show*.



2. Set the filter.

   The available filter options are:

   - *Mapped*
   - *Unmapped*
   - *Spec Check Fail*
   - *Non Analog*
   - *Other Cellviews*

   Ensure that the requirement or implementation types you want to hide are not selected.

Verifier filters the requirements hierarchy according to the specified criteria.

Custom Fields

Verifier lets you include additional information for your design verification project, and for each requirement in the project. For example, you can include the name and e-mail address of the verification project manager. For each requirement in the project, you can include the name and e-mail address of the engineer.

Verifier only stores the custom field values for reference. It does not use the custom fields for any other purpose.

© 2023
All Rights Reserved.

This section provides information on how you can use the custom fields functionality of Verifier.

Related Topics

Preparing ADE Verifier to Use Custom Fields

Specifying Values for Custom Fields

Preparing ADE Verifier to Use Custom Fields

To use custom fields in Verifier, you must define the fields in a CSV file and set the Verifier environment to include those fields.



Important

Before you set Verifier for custom fields, determine if you want different custom fields for the project and its requirements. While the custom fields for the project are global settings for the Verifier setup and are also available in the *Results* page, the custom fields for the requirements are custom entries for each requirement in the verification plan. If you want different fields, you must maintain two CSV files, one for the custom project fields and one for the custom requirement fields. If your custom fields are common for the project and its requirements, you can define them in a single CSV file.

To prepare Verifier for using custom fields:

1. Specify the custom field tags, names, and tool tips in the CSV files in the following format:

   Field tag name, Field tag description, Field tool tip

   See the following examples:

   The `infoProject.csv` file contains the following custom field details for the project:

   ```
   Manager,Manager Name,The name of the project manager.
   Email,E-mail Address,The e-mail address of the project manager.
   ```

   The `infoReq.csv` file contains the following custom field details for the requirements:

   ```
   Engineer,The name of the lead engineer.
   Email,The e-mail address of the engineer.
   ```


© 2023 All Rights Reserved.

Note: In case of existing `infoReq.csv` files that have three columns, the *`Field tag description`* is ignored, and the *`Field tag name`* is used instead.

2. Set the Verifier environment variable csvfile (customFieldConfig) and csvfile (customReqFieldConfig) to indicate the CSV files of the custom project fields and the custom requirement fields, respectively.

   You can set the variables in one of the `.cdsenv` file, `.cdsinit` file, or Virtuoso CIW. The following snippet illustrates how you set these variables in the `.cdsinit` file stored in the root directory of your project.

   The filename and location entry can contain the character ~ for user home directory, and *`$UNIX_environment_variable`* for the value of an environment variable.

```
envSetVal("verifier.customFieldConfig" "csvfile" 'string "~/infoProject.csv")
envSetVal("verifier.customReqFieldConfig" "csvfile" 'string "~/infoReq.csv")
```

The custom fields become available to the Verifier sessions.

Specifying Values for Custom Fields

After you set up Verifier, the custom fields become available to the Verifier sessions. The custom project fields are available in the *Custom Data* tab of the Change Referenced Cellview Form. The custom requirement fields are available in the Requirements Editor Assistant and requirements tree. Verifier updates the table in Requirements Editor assistant if you edit the custom requirement fields in requirements' tree. Conversely, the requirements' tree is automatically updated if you edit the table in Requirements Editor Assistant.

You set the value of the custom fields in a Verifier session. These values are saved in the Verifier cellviews.

To set the values of the custom project fields

1. Choose *Edit – Preferences*.

   The Preferences form displays.

2. Click the *Custom Data* tab.

   The custom fields display in a table.

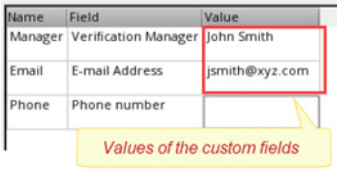
© 2023 All Rights Reserved.

The *Custom Data* tab becomes available when you set the Verifier environment variable `customFieldConfig`.



3. Double-click the *Value* cell of a custom field and type the value.

   Repeat this step to specify the value of the remaining custom fields.



4. Click *OK*.

You can save your changes in the cellview.

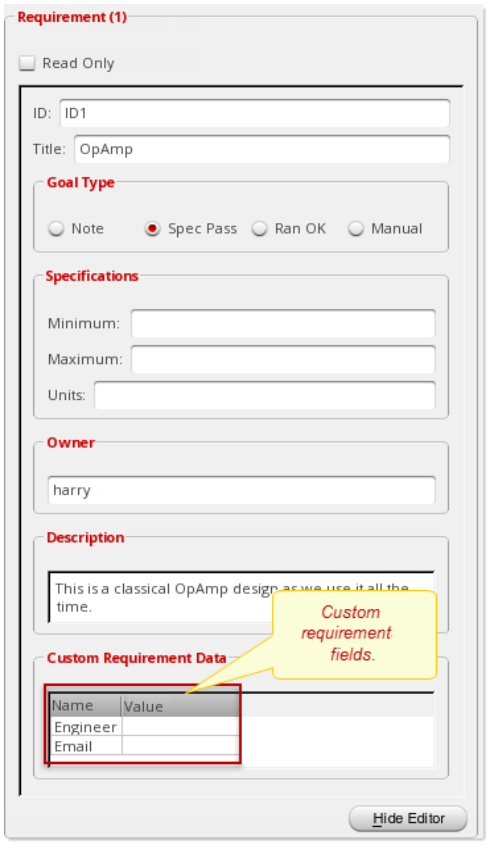
To set the values of the custom fields of a requirement

1. Click the *Setup* tab, if it is not the current tab.
2. Select the requirement.
3. Do one of the following:
   - Click the *Edit* icon on the toolbar: 
   - Choose *Edit – Open Requirement Editor*.

   The Requirement Editor assistant displays the custom fields in the *Custom Requirement Data* group box, as illustrated in the following figure.

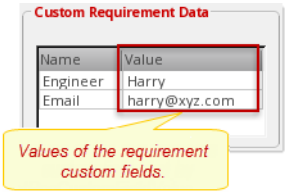
© 2023 All Rights Reserved.

The *Custom Requirement Data* group box becomes available when you set the Verifier environment variable `customReqFieldConfig`.



4. Double-click the *Value* cell of a custom field and type the value.

   Repeat this step to specify the value of the remaining custom fields.



You can save your changes in the cellview.

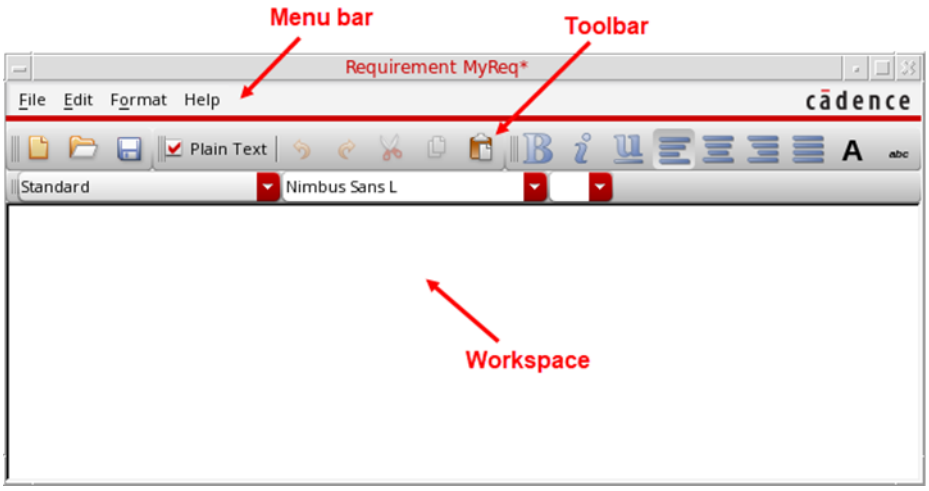
© 2023
All Rights Reserved.

Description Editor for Requirements

The description editor is a compact but powerful utility that is available in the *Description* column of the *Requirements* pane on the *Setup* page. Use this editor to edit your requirement descriptions. The edited description is saved to the requirement.

Using the description editor, you can add various HTML formats to text, add links to documents and cellviews, open the added documents and cellviews by clicking the links, as well as add images and display them inline. When you save the requirement description, the modifications are added to the requirement in the Verifier cellview. The documents, links, and formatting changes are saved in the `documents` directory and the images are saved in the `images` directory. These directories are available in the Verifier cellview directory structure. Therefore, when you use the *SaveAs* command from the Verifier toolbar, the documents and images included with the requirements are exported, and when you import a cellview, these documents and images are also imported.

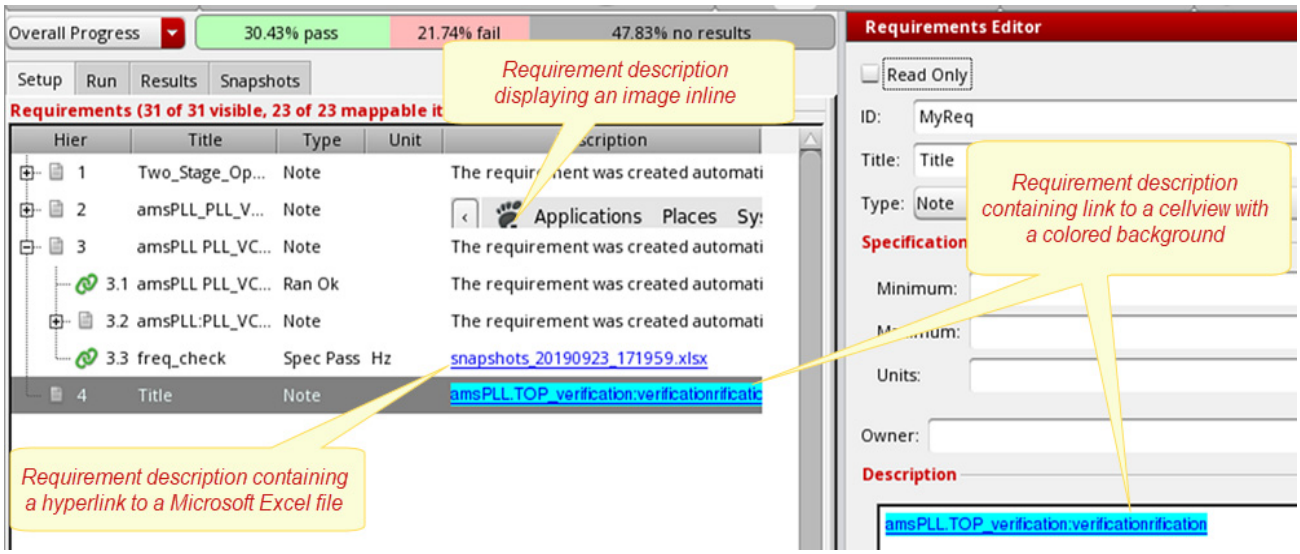
To access the description editor, right click a requirement in the *Requirements* pane of the *Setup* page, and select *Edit Description*. The editor window appears and displays the name of the selected requirement as the window title in the format *Requirement <name>*.



The editor window always displays the name of the selected requirement in the format *Requirement <name>*. The edited description is displayed in the Requirements Editor assistant as well as the Information assistant.

© 2023
All Rights Reserved.

The following illustration shows the Requirements Editor assistant displaying the changes made to the requirement description.



The following table describes the main controls available on the description editor toolbar for a requirement.

Command	Description
New	Creates a new description for the selected requirement and saves it as plain text or HTML. GUI Command: *File – New* Shortcut key: `Ctrl+N`
Open	Opens an existing HTML file. GUI Command: *File – Open* Shortcut key: `Ctrl+O`
Save	Saves the edited description. GUI Command: *File – Save* Shortcut key: `Ctrl+S`
Plain Text	If selected, removes HTML formatting of text and sets the text type to Plain Text.

© 2023
All Rights Reserved.

Command	Description
Undo	Reverses the last changes made to the description. GUI Command: *Edit – Undo* Shortcut key: `Ctrl+Z`
Redo	Repeats the last changes made to the description. GUI Command: *Edit – Redo* Shortcut key: `Ctrl+Shift+Z`
Cut	Removes the selection temporarily and pastes it on the clipboard. GUI Command: *Edit – Cut* Shortcut key: `Ctrl+X`
Copy	Copies a selection to the clipboard. GUI Command: *Edit – Copy* Shortcut key: `Ctrl+Ins`
Paste	Pastes a copied selection from the clipboard. GUI Command: *Edit – Paste* Shortcut key: `Ctrl+V`
Bold	Increases the font weight by making the text thicker than the surrounding text. GUI Command: *Format – Bold* Shortcut key: `Ctrl+B`
Italic	Changes the text to italic format. GUI Command: *Format – Italic* Shortcut key: `Ctrl+I`
Underline	Adds an underline beneath the text. GUI Command: *Format – Underline* Shortcut key: `Ctrl+U`

© 2023 All Rights Reserved.

Command	Description
Left	Aligns the selection to the left in the editor workspace. GUI Command: *Format – Left* Shortcut key: `Ctrl+L`
Center	Aligns the selection to the center in the editor workspace. GUI Command: *Format – Center* Shortcut key: `Ctrl+E`
Right	Aligns the selection to the right in the editor workspace. GUI Command: *Format – Right* Shortcut key: `Ctrl+R`
Justify	Justifies the selection in the editor workspace. GUI Command: *Format – Justify* Shortcut key: `Ctrl+J`
Color	Changes the color of the selected text to the specified color. GUI Command: *Format – Color*
Background Color	Changes the background color of the selected text to the specified color. GUI Command: *Format – Background Color*
List	Sets the text to the standard, ordered list, or unordered list format.
Font	Sets the text font to the selected font.
Size	Sets the text to the selected size.

The following table describes the additional shortcut menu commands available in the description editor that help you manage your requirement descriptions in a Verifier cellview.

Command	Description
Insert Document	Saves an existing document in the `documents` directory and creates a hyperlink to it. GUI Command: *Edit – Insert Document*

© 2023 All Rights Reserved.

Command	Description
Link Document	Adds a link to an existing document from the `documents` directory. When you save the description and click the link in the *Description* column, the document opens in an appropriate application.
	GUI Command: *Edit – Link Document*
Insert Image	Saves an existing image in the `images` directory and creates a hyperlink to it. When you save the description, the image is displayed inline.
	GUI Command: *Edit – Insert Image*
Link Image	Adds a link to an existing image from the `images` directory.
	GUI Command: *Edit – Link Image*
Insert Cellview	Inserts a link to an existing cellview in the requirement description.
	GUI Command: *Edit – Insert Cellview*
Delete	Deletes the current selection.
	Shortcut key: `Delete`
Select All	Selects everything in the editor workspace.
	Shortcut key: `Ctrl+A`

© 2023 All Rights Reserved.

© 2023 All Rights Reserved.

3

Composite Requirements

In Verifier, a typical requirement includes the ID, name, description, and other properties, which are displayed in a single row. In complex cases, a single requirement cannot be used to verify simulation results in different operating conditions. To meet the needs of such complex cases, Verifier provides a special type of requirement, known as *Composite*.

The key features of a composite requirement are as follows:

- The results for the same measurement can be verified under different conditions
- A composite requirement can have only one sub-level of hierarchy, with one parent and its child requirements
- The child requirements inherit the properties and mappings of the parent requirement of the Composite type, unless they are overridden
- The child requirements can be assigned with different verification spaces that define the special simulation conditions

Typically, when you have a few requirements for which you want to verify measurements under different conditions, the requirements are recorded or logged in a tabular format. For example, data for verifying requirements `Gain` and `UGF` for different temperatures can be represented in a tabular format, as follows:

Req	Temp = 27		Temp = 100	
	Min	Max	Min	Max
Gain	34	38	33	36
UGF	100M	120M	96M	115M

In this example, the conditions include a set of sweeps, corners and simulation settings, and are defined in a verification space in the Setup Library assistant. In this table, the group `Temp=27` in the first row is the condition group and is used as a composite header for composite requirements. Similarly, you can create other condition groups as needed.

© 2023 All Rights Reserved.

Adding Composite Requirements

In Verifier, composite requirements can be added in the following ways:

- Adding a Composite Requirement Manually
- Importing Composite Requirements from File

Adding a Composite Requirement Manually

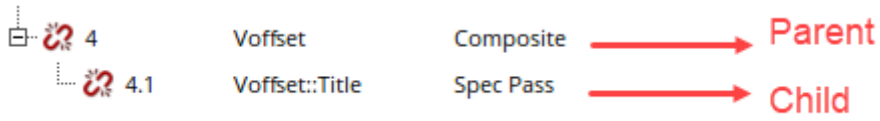
To manually add a composite requirement:

1. In the *Requirements* pane of the *Setup* page, right-click and select the *Add Requirement* shortcut command to add a new requirement.

   A new requirement is added.

2. In the *Type* column for this requirement, select *Composite*.
3. Right-click this requirement and choose *Add Requirement*.

   A child requirement of type `Spec Pass` is added. Observe that the *Title* for the child requirement appears as `parent_title::child_title`.



4. Select the *Composite* requirement and add more child requirements as needed.

Importing Composite Requirements from File

To create composite requirements in Verifier by importing data from a MS Excel file:

1. Choose *File – Import – File*.

   The Import Form appears.

2. In the *File* group box, select *Excel file*.
3. Click the *Browse* button next to the *Excel file* text box and select the required Excel file.
4. In the *Rows* group box of the Import form, select Use folded format for composite requirements.

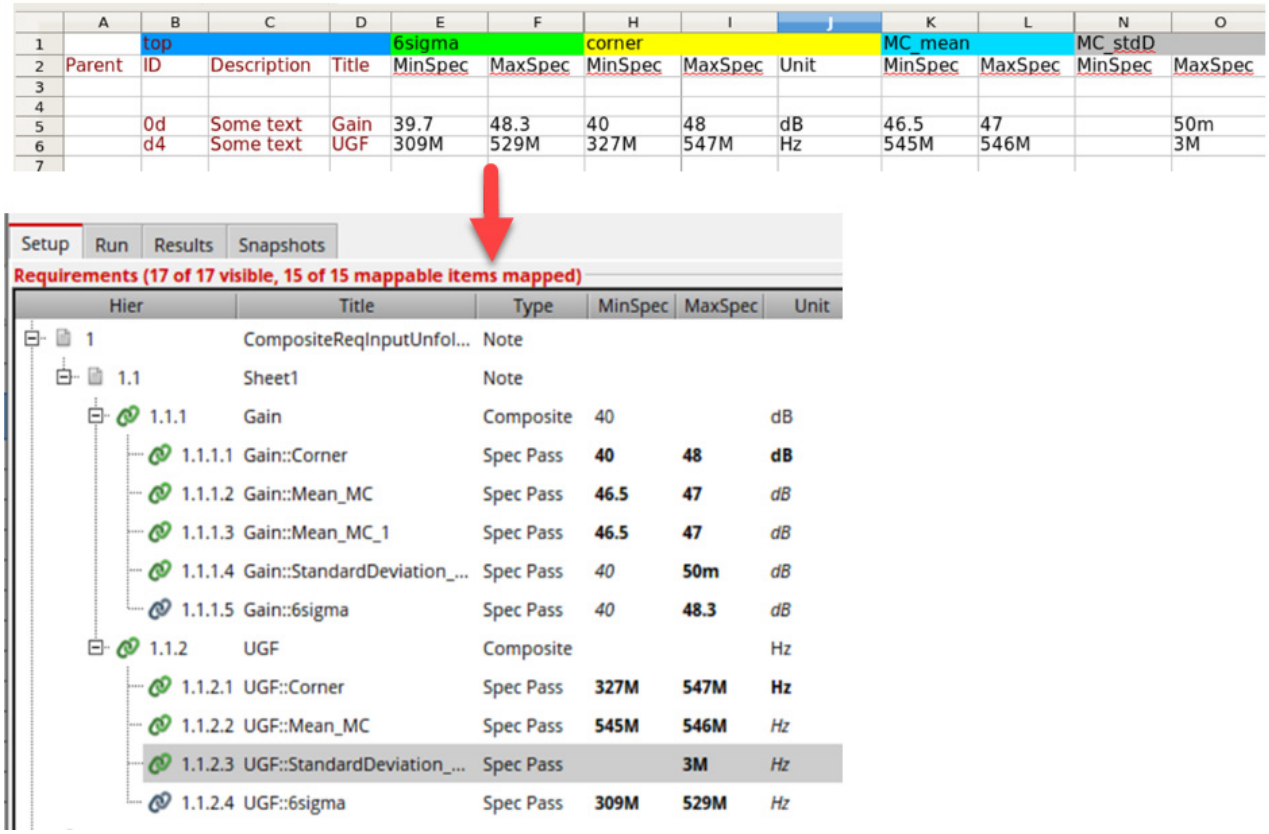
© 2023
All Rights Reserved.

In the folded format, the information is provided in the composite requirement structure based on the composite header specified in the file.

Enable this option to use the row preceding the header as the composite header. The Excel data in folded format will be converted into the composite requirement structure based on the composite header specified in the file.

5. Click *OK* to import the file.

The tabular data from the Excel file is imported and the data is converted into composite requirements in Verifier.



	A	B	C	D	E	F	H	I	J	K	L	N	O
1		top			6sigma		corner			MC_mean		MC_stdD	
2	Parent	ID	Description	Title	MinSpec	MaxSpec	MinSpec	MaxSpec	Unit	MinSpec	MaxSpec	MinSpec	MaxSpec
3													
4													
5		0d	Some text	Gain	39.7	48.3	40	48	dB	46.5	47		50m
6		d4	Some text	UGF	309M	529M	327M	547M	Hz	545M	546M		3M
7													

Setup | Run | Results | Snapshots

Requirements (17 of 17 visible, 15 of 15 mappable items mapped)

Hier	Title	Type	MinSpec	MaxSpec	Unit
1	CompositeReqInputUnfol...	Note			
1.1	Sheet1	Note			
1.1.1	Gain	Composite	40		dB
1.1.1.1	Gain::Corner	Spec Pass	**40**	**48**	**dB**
1.1.1.2	Gain::Mean_MC	Spec Pass	**46.5**	**47**	*dB*
1.1.1.3	Gain::Mean_MC_1	Spec Pass	**46.5**	**47**	*dB*
1.1.1.4	Gain::StandardDeviation_...	Spec Pass	*40*	**50m**	*dB*
1.1.1.5	Gain::6sigma	Spec Pass	*40*	**48.3**	*dB*
1.1.2	UGF	Composite			Hz
1.1.2.1	UGF::Corner	Spec Pass	**327M**	**547M**	**Hz**
1.1.2.2	UGF::Mean_MC	Spec Pass	**545M**	**546M**	*Hz*
1.1.2.3	UGF::StandardDeviation_...	Spec Pass		**3M**	*Hz*
1.1.2.4	UGF::6sigma	Spec Pass	**309M**	**529M**	*Hz*

Alternatively, you can also create composite requirements by importing data in unfolded format from an Excel file. The unfolded format has all requirements in a flat structure, where all parent and child requirements are listed individually in a single row. You can also choose to keep your Excel data in the same structure as the Verifier hierarchy, so that after an import, the imported requirements appear the same as in the Excel file.

© 2023
All Rights Reserved.

Overwriting Properties in Composite Requirements

Composite requirements follow a parent-child hierarchy structure. This means that any property defined for the parent is automatically inherited by its child requirements. When you make any changes to the property value of a composite requirement, all properties inherited by the child requirements are updated automatically. However, you can modify the properties of a child requirement individually.

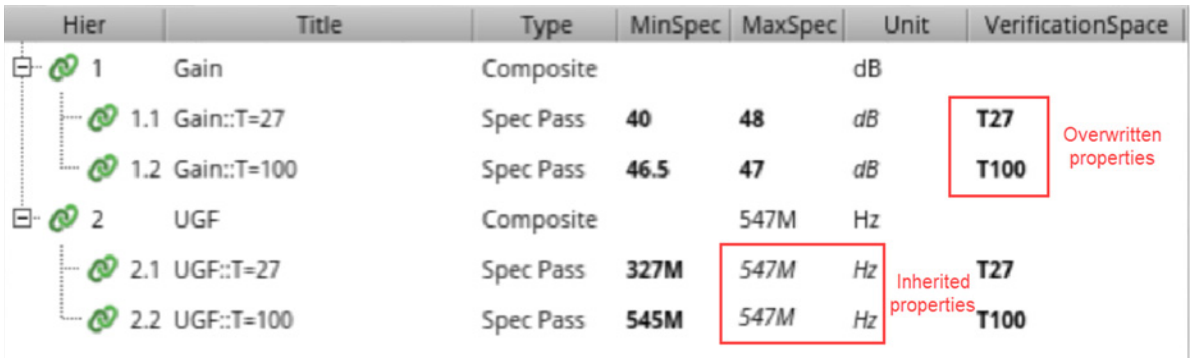
To modify the properties of a child requirement

1. Double-click the required property value to make it editable.
2. Specify a new value.

   This overwrites the property value inherited from the composite requirement.

Inherited property values are indicated by italicized text; overwritten values are shown in bold.

The following figure shows how Verifier displays inherited and overwritten property values:



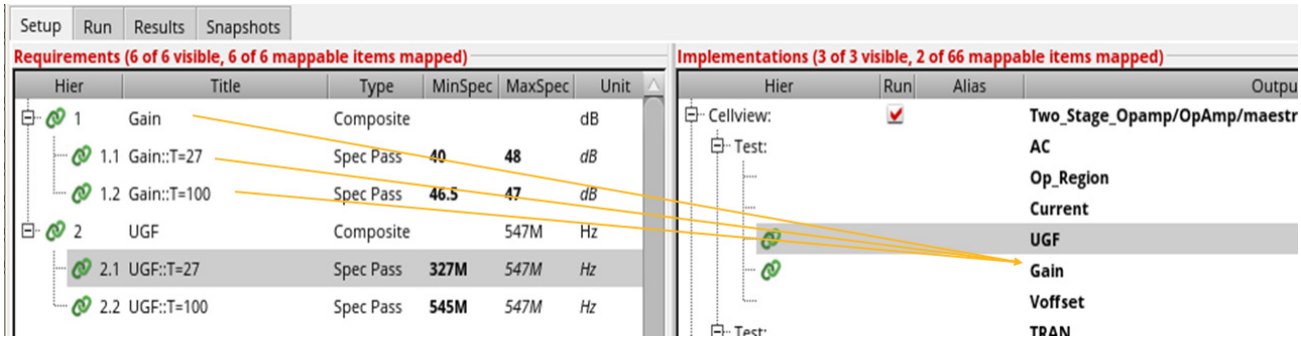
Observe that the requirement ID and title of the composite parent are not inherited by the child requirement. The title of the child requirement is created in the `parent_title::child_title` format, where the `child title` is the name of the condition group.

When you do not specify any value for the specification of a child requirement, the child inherits the specification values from the parent. However, if you want to present the child requirement from inheriting the specification values of its parent, you can specify a string value, such as `"-"`. This makes the tool ignore this specification value without failing the simulation.

© 2023
All Rights Reserved.

Overwriting Mappings in Composite Requirements

The inheritance of mappings is similar to inheritance of property values. When you map a composite requirement to an implementation, the same mapping is inherited by the child requirement. When you make any changes to the mapping of a composite requirement, all mappings inherited by the child requirements are updated automatically.



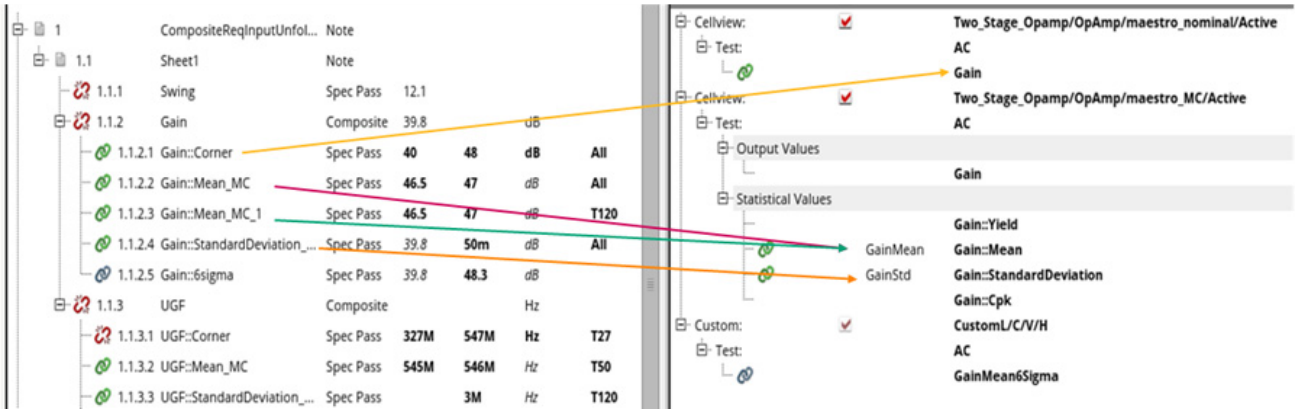
If you want to change the mappings for a child requirement, you need to overwrite the mapping individually.

To modify the mapping of a child requirement:

1. Select the child requirement.
2. Right-click and choose *Delete Mapping*.
3. Select the child requirement once again.
4. Press the `Ctrl` key and select the required implementation to be mapped.
5. Right-click and select *Map*.

© 2023
All Rights Reserved.

This overwrites the mapping inherited from the composite requirement and creates the individual mapping for the child requirement. If the composite requirement is not mapped, all of its child requirements need to be mapped individually.



Slicing Results According to the Assigned Verification Spaces

Composite requirements let you use different verification spaces that define the different conditions to run simulations, and then slice the data on the basis of these verification spaces.

Running Implementation with Multiple Verification Spaces

Verification spaces are a combination of sweeps, corners, and simulation settings that you specify in the Setup Library assistant. When you assign multiple verification spaces to requirements that are mapped to the same implementation, you can get results for all conditions with a single composite requirement by running the implementation with a superset of all assigned verification spaces.

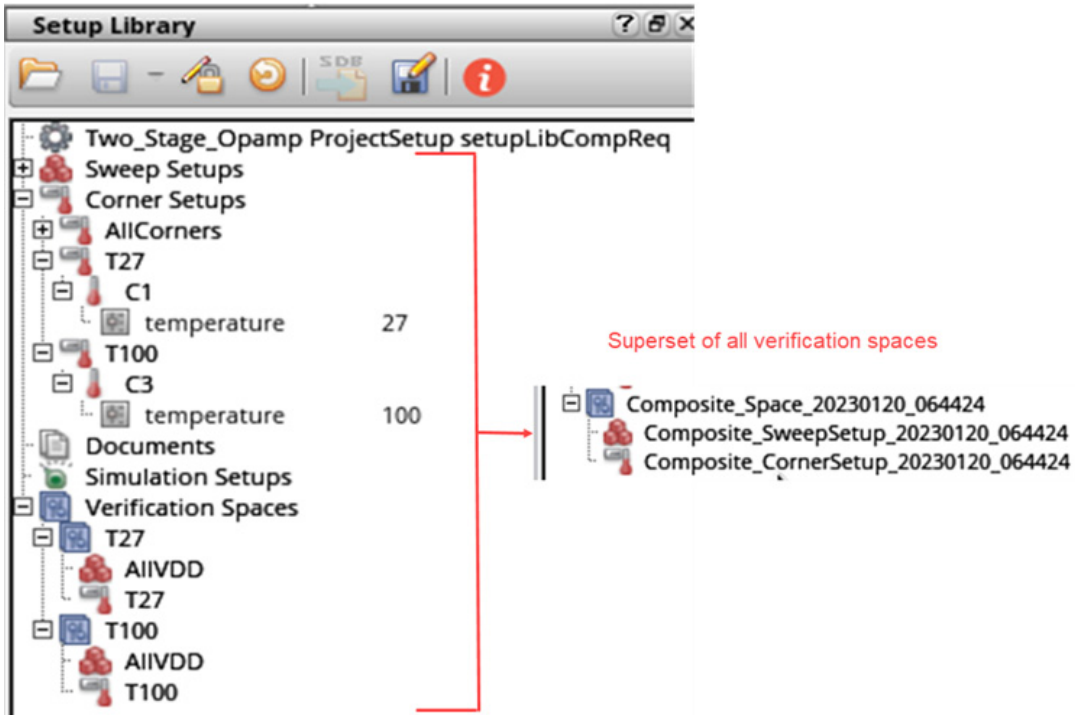
Consider a scenario where the user wants to run a single implementation and the conditions to be checked are `Temp=27` and `Temp=100`.

Req	Temp = 27		Temp = 100	
	Min Spec	Max Spec	Min Spec	Max Spec
Gain	40	48	46.5	47
UGF	327M	547M	545M	547M

In this case, the user can specify corners with different temperatures in a verification space and then assign these spaces to the required child requirement and map this requirement to an implementation.

© 2023 All Rights Reserved.

During a *Local with SPACE* or *Batch with SPACE* run, when multiple verification spaces are detected on an implementation, the assigned verification spaces are temporarily merged and the implementation is run with the merged verification space.



If more than one simulation setup is found, the temporarily merged verification space picks the first simulation setup found from the assigned verification spaces. This means that simulation setup data is not merged but only the first of all verification spaces is used. You can save the merged verification space by using:

```
envSetVal("maestro.sla" "saveCompositeSetup" 'boolean t)
```

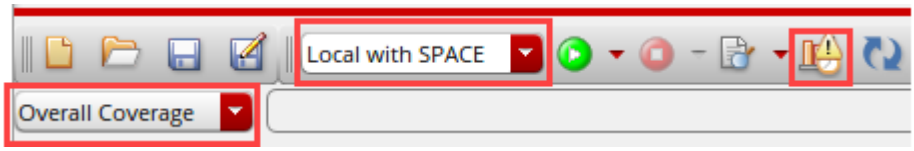
Viewing Sliced Results

After simulation, the results are sliced according to the assigned verification spaces and assigned to the corresponding requirement.

To see the slicing of results data according to the verification spaces:

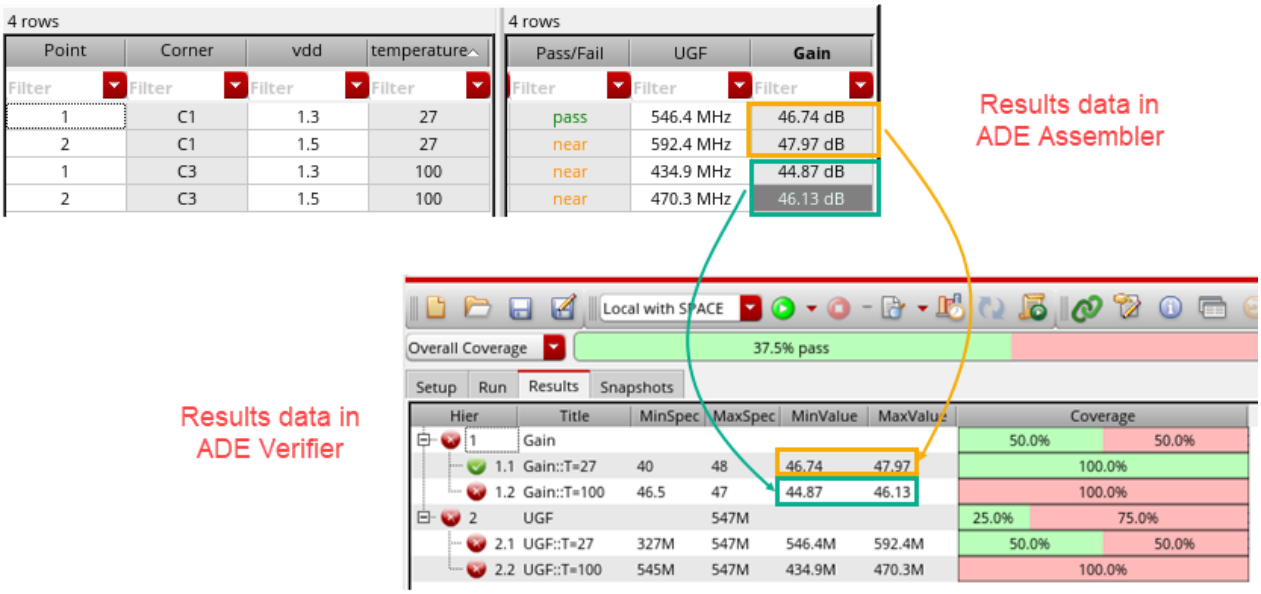
© 2023
All Rights Reserved.

1. On the main toolbar, click the *Overall Progress* list option and switch to the *Overall Coverage* option.



2. Click the Calculate Coverage button, if required.

The Results pane displays the sliced results data.



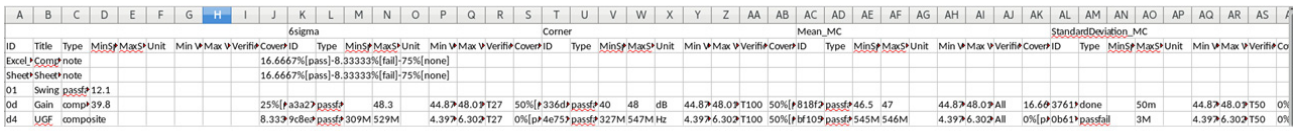
Exporting Composite Requirements

When you export your composite requirements data to an MS Excel file, the data can be exported into two formats, folded and unfolded. The unfolded format has all requirements in a flat structure, where all parent and child requirements are listed individually in a single row.

© 2023
All Rights Reserved.

In the folded format, the information is provided in the composite requirement structure based on the composite header specified in the file.

Folded format



| | | |

3. In the *What to Save* group box, select the properties that you want to export.
4. Select the *Use folded format for composite* option.

   If this option is not selected, the requirements are exported in an unfolded format.
5. Click *OK*.

   The data is exported in folded format.

Similar to the import functionality, you can export your composite requirements in the unfolded format.

© 2023 All Rights Reserved.

4

Implementations and Mappings in Virtuoso ADE Verifier

This topic provides information on how you manage implementations and their mappings with requirements in Virtuoso ADE Verifier (Verifier).

Implementations are `maestro` cellviews and their tests and outputs that you map to their corresponding requirements. Verifier lets you add implementation cellviews, which you can rearrange, and through which you can search and filter. You can also open the implementation cellviews in their native application, such as ADE Assembler.

You can run implementation cellviews from Verifier to capture results, or load existing run results. Verifier uses these results, along with other settings, to determine the verification status. For information on running implementations, see Implementation Runs in ADE Verifier.

Related Topics

Details of an Implementation

Adding Implementations

Implementation Run Modes

Mappings between Implementations and Requirements

Typical Value Setup

Project Directory Setup

Custom Implementations

© 2023 All Rights Reserved.

Details of an Implementation

An implementation represents the simulation setup, testbench, or the measured output of a verification requirement. Implementation data is stored in cellviews of the type `maestro`. You verification project can have a list of unique implementation cellview–history entries. You can change the order of implementations cellviews in this list.

To determine the status of a requirement, Verifier uses the information of the implementations mapped to that requirement with other relevant settings. For example, you can set a requirement to pass verification if the mapped implementation is simulated successfully. In this case, Verifier checks the simulation status of the mapped implementation to determine the verification status of the requirement.

The following table describes the details of an implementation.

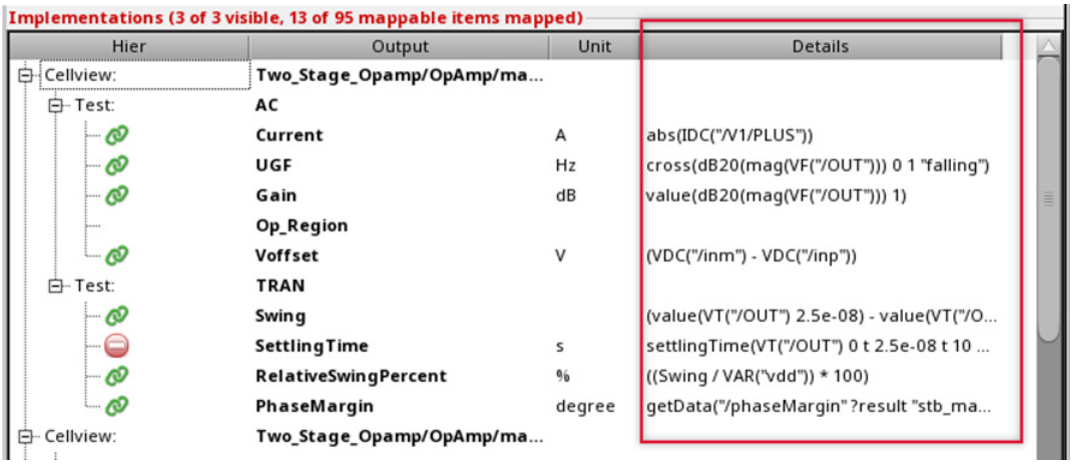
Column	Description
Hierarchy	The expandable and collapsible hierarchy level of implementations.
Run	The option to run the implementation from Verifier. Select this check box if you want to initiate the run from Verifier, or clear to load the test results from the selected history when the implementation cellview is run from outside Verifier.
Output	The name of the simulation output to be used for determining the verification result. Outputs are measurements and assertions that can be evaluated using the simulation results. The *Output* column displays the cellview and test information in addition to the outputs. Therefore, the *Library*, *Cell*, *View*, and *Test* columns are not displayed by default. Ensure that the output name is a non-empty string which is not '.' or '..' and does not contain whitespaces or the characters: '~*/<>'?|\"()=;&$'."
Library	The library to which the implementation cellview belongs.
Cell	The cell that contains implementation data.
View	The view name of the implementation cellview.

© 2023 All Rights Reserved.

Column	Description
History	The option to select the cellview history. When you initiate the simulation of a local implementation from Verifier, a session is loaded with the implementation cellview where the specified history is set as the active history within that session. By default, a new history is created outside the cellview when the implementation is run. ■ The cellview–history entry must be unique in the implementations list. ■ Verifier displays the tests and outputs of the selected history, which you can map to requirements. ■ If the implementation has been run outside the current Verifier session, you need to manually refresh the *History* list by using *Tools -> Update*. ■ If the history does not exists for an implementation, an error message is displayed and the `Active` history is used. When you change the history of an implementation, Verifier saves the *Run* check box settings and the mapping information for the selected history if the *RunPlan* mode is same between the two histories. A warning message is also displayed to warn about mapping loss as the consequence of the selected history changes (the *RunPlan* mode, *Test*, and *Output* changes). Ensure that the history name is a non-empty string which is not '.' or '..' and does not contain whitespaces or the characters: '~*/<>'?|\"()=;&$'."
RunPlan	The run plan associated with a cellview.
Test	The name of the implementation test. Ensure that the test name is a non-empty string which is not '.' or '..' and does not contain whitespaces or the characters: '~*/<>'?|\"()=;&$'."

© 2023 All Rights Reserved.

Column	Description
Details	The measurement of output from the implementation cellview that is loaded from ADE Assembler. The values of these implementation output details are the same as those in the Assembler cellview.



Specification	The specification value for implementation output even if it is Spec Check Fail. The Spec check fail icon is shown besides the specification value. The value is shown in red color and the tool tip displays an error message. Unmapped Implementation: Displays the specification value for implementation output even if it does not map to any requirement. `Spec check fail` icon is shown besides the specification value. The value is shown in red color and tool tip displays an error message.



Units	The unit used for the measurement output.
Priority	The priority used for the simulation run.
EstRunTime	The estimated run time for the simulation run.

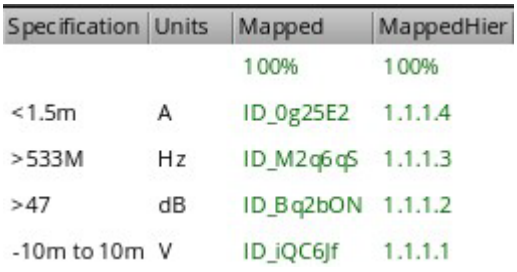
© 2023
All Rights Reserved.

Column	Description
Alias	The short reference used to specify a selected library, cell, view, test, or output.

© 2023 All Rights Reserved.

Column	Description
Mapped	The mapping status of the implementation. ■ This column is hidden by default, because for mapped implementations, the UUID is lengthy. The *MappedHier* column shows the readable hierarchical numbering for requirement instead of the requirement ID. ■ The requirements hierarchy indicates the mapping status through icons. The title of the requirements area and implementations area indicate the overall mapping percentage. The following are the possible mapping status of implementations: ■ *Requirement ID*: Indicates that the implementation is mapped to the requirement ID specified in green.  ■ *No*: Indicates that the implementation is not mapped to any requirement.  ■ *Percentage*: Indicates the percentage of mapped implementations in the cellview and its sub-categories. The percentage is displayed as a note when a cellview or a test is not mapped.  **Note:** Verifier ignores the hidden implementations in the setup when calculating the percentage.

© 2023
All Rights Reserved.

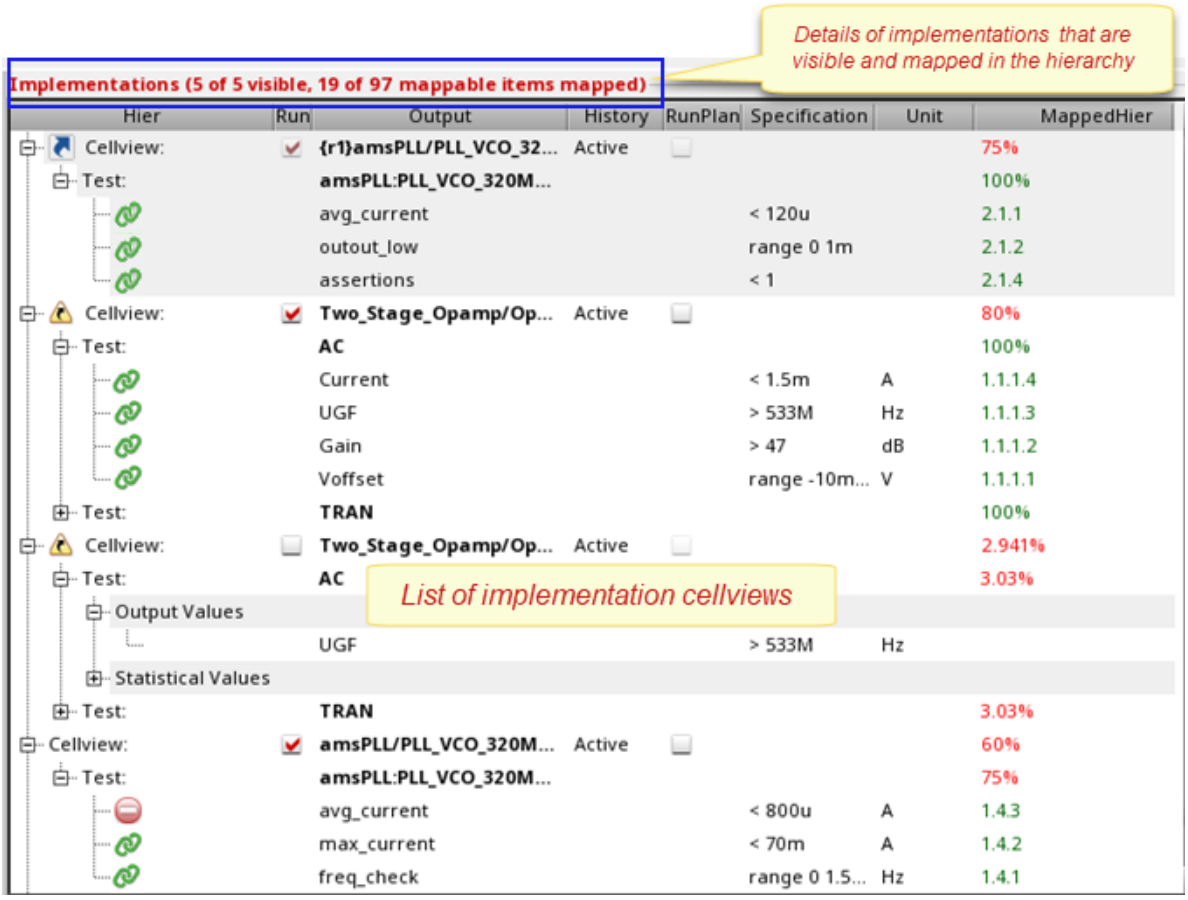
Column	Description
MappedHier	The hierarchical numbering of mapped requirements. Hierarchical number: Indicates that the implementation is mapped to the hierarchical number of the requirement, specified in green.  For more details, see the environment variable includeOnlyMappableOutputsInCount.

Specification	Units	Mapped	MappedHier
		100%	100%
<1.5m	A	ID_0g25E2	1.1.1.4
>533M	Hz	ID_M2q6qS	1.1.1.3
>47	dB	ID_Bq2bON	1.1.1.2
-10m to 10m	V	ID_iQC6Jf	1.1.1.1

Row	Description
Filter Row	This option is available when you right-click anywhere on the column headings and select *Filter Row*. This row contains individual filters for every column in the implementation hierarchy. You can enter the required criteria in a column filter to display only the implementations matching the criteria. You can also refine the filtered implementation by specifying criteria in other columns. This row is hidden by default.

© 2023
All Rights Reserved.

The following figure shows a list of implementations. For more details, see Implementations.



Hier	Run	Output	History	RunPlan	Specification	Unit	MappedHier
Cellview:	✓	{r1}amsPLL/PLL_VCO_32...	Active				75%
Test:		amsPLL:PLL_VCO_320M...					100%
		avg_current			< 120u		2.1.1
		outout_low			range 0 1m		2.1.2
		assertions			< 1		2.1.4
Cellview:	✓	Two_Stage_Opamp/Op...	Active				80%
Test:		AC					100%
		Current			< 1.5m	A	1.1.1.4
		UGF			> 533M	Hz	1.1.1.3
		Gain			> 47	dB	1.1.1.2
		Voffset			range -10m...	V	1.1.1.1
Test:		TRAN					100%
Cellview:		Two_Stage_Opamp/Op...	Active				2.941%
Test:		AC					3.03%
Output Values							
		UGF			> 533M	Hz	
Statistical Values							
Test:		TRAN					3.03%
Cellview:	✓	amsPLL/PLL_VCO_320M...	Active				60%
Test:		amsPLL:PLL_VCO_320M...					75%
		avg_current			< 800u	A	1.4.3
		max_current			< 70m	A	1.4.2
		freq_check			range 0 1.5...	Hz	1.4.1

You can show or hide the columns of the implementations list in the *Setup* tab. For this, right-click anywhere on the column heading in the implementations area and select the columns you want to show. Verifier hides the columns that are not selected.

To sort columns, you can use the drag-and-drop method and change the display of columns. Additionally, you can specify the default column order and hidden columns by using the environment variables impColumnOrder and impHiddenColumns.

Disabled Runs, Reliability Analyses, Tests, or Outputs for Implementation Cellviews

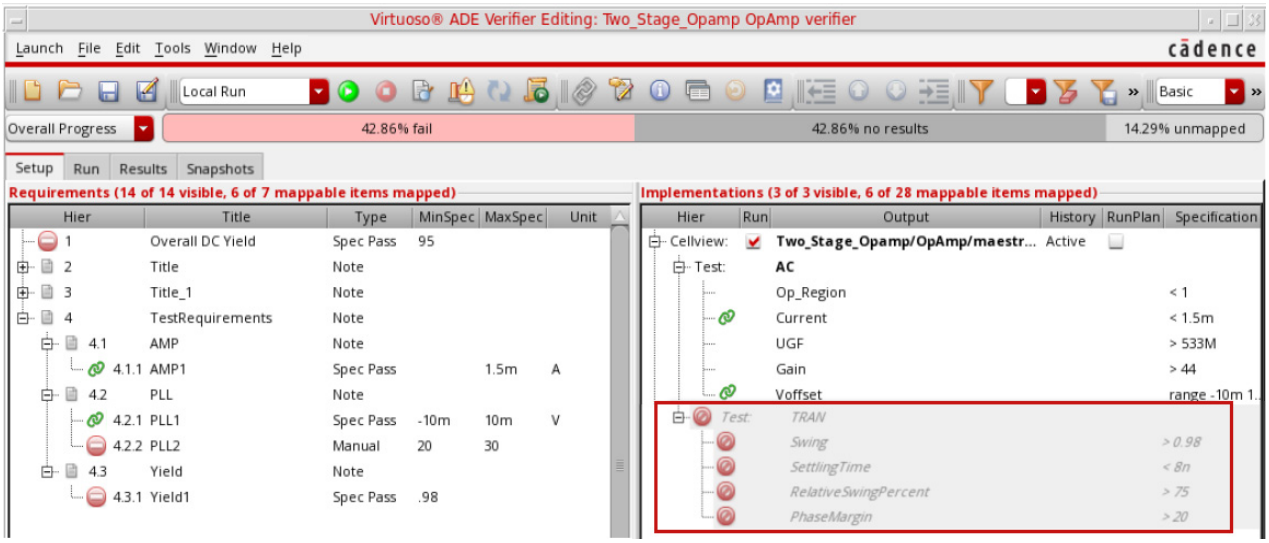
You can include a run, reliability analysis, test, or output that is disabled in ADE Assembler when you add or update an implementation in your Verifier cellview. To do this, select the Include disabled run, reliability, test, or output for implementation setting in the Preferences form.

© 2023 All Rights Reserved.

Tip

Ensure that the shortcut menu item *Show – Disabled* is selected.

The disabled test, run, or output shows with a gray background in Verifier. When selected, this setting shows the disabled tests, runs, or outputs in the implementation.



This setting is applicable only for newly added implementations or the next time you load an existing implementation. For example:

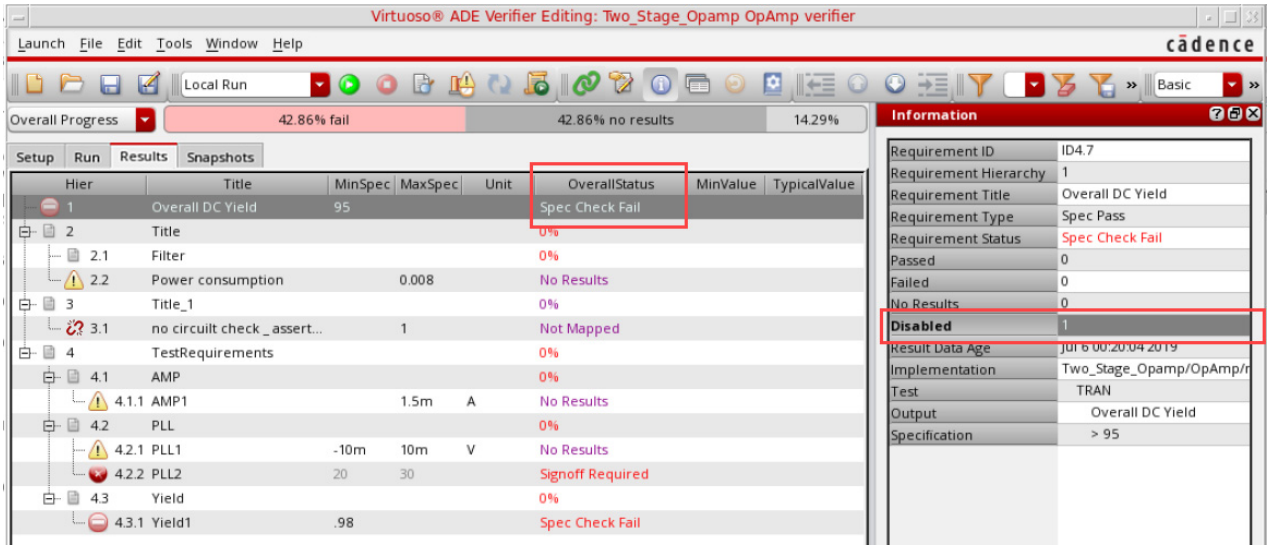
- When you toggle the history
- When you update or modify an implementation

When you move the mouse pointer on a disabled run or test in the *Implementations* pane, the tooltip text shows the following:

- For a run disabled in Assembler but enabled in Verifier: *Disabled Run:* `<run name>` *(Locally enabled)*.
- For a test disabled in Assembler but enabled in Verifier: *Disabled Test:* `<test name>`.
- For a run enabled in Assembler but disabled in Verifier: `<run name>` *(Locally disabled)*.
- For a test enabled in Assembler but disabled in Verifier: `<test name>` *(Locally disabled)*.

© 2023 All Rights Reserved.

The requirements mapped to a disabled test, run, or output shows *Spec Check Fail* in the *OverallStatus* column of the Results tab and *Disabled* in the *Status* column of the Run tab. The Information assistant shows a count of disabled mappings.



Modification of the Enabled or Disabled State

You can edit the enabled or disabled state of implementation runs or tests in Verifier. To do this, use Allow editing the enabled or disabled state for implementation run or test in the Preferences form. When you select one of these, the check box in the *Run* column of the *Implementations* pane becomes editable. You can select these check boxes to enable the run or test locally and run simulations on these local sets in Verifier. Selecting these check boxes removes the gray background from these runs or tests and adds a pale blue background to the *Run* column. The pale blue background behind a test name indicates that the run or test state is different in Verifier from the `maestro` cellview.

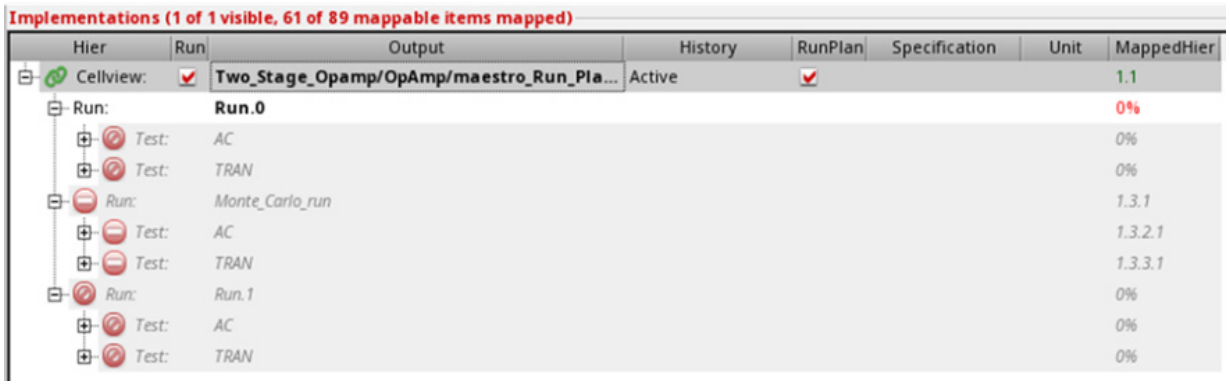
Note: Verifier does not support manually editing the state of an output. To enable or disable outputs, make the changes in ADE Assembler.

Example:

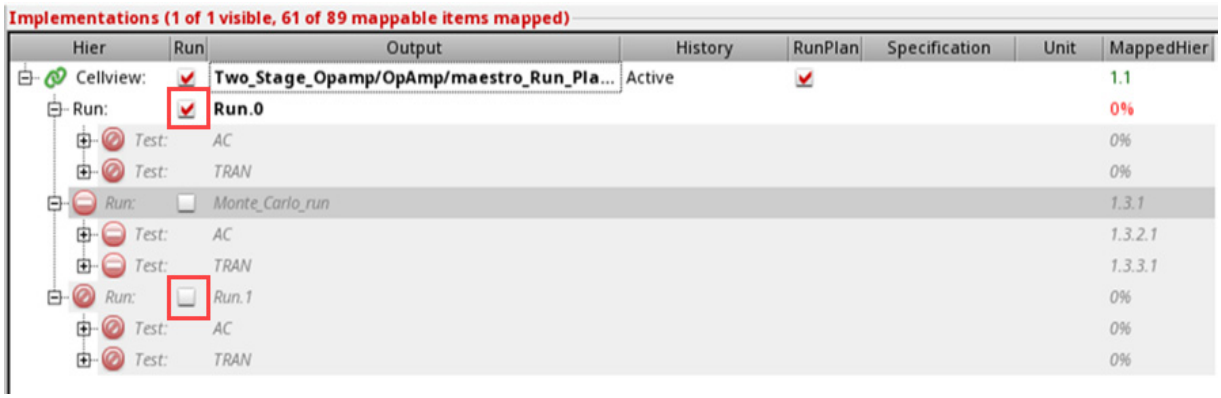
Assume that you have an ADE Assembler cellview `Two_Stage_Opamp/OpAmp/maestro_Run_Plan/Active` with two run plans `Run.0` and `Run.1` that contain two tests `AC` and `TRAN` each. Assume that you have disabled `Run.1` and the test `TRAN` in `Run.0`.You want to run simulations on these run plans or tests but you need to make modifications to the enabled or disabled status of these runs or tests. Instead of opening Assembler, making the changes, and then returning to the Verifier cellview, you can make these modifications directly in Verifier. You add this cellview as an implementation and select the preference option

© 2023 All Rights Reserved.

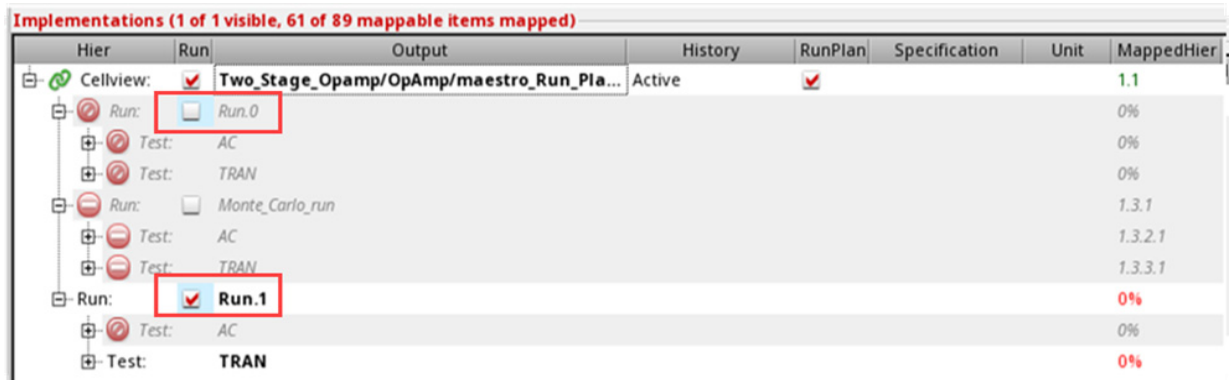
Include disabled run, reliability, test, or output for implementation. The following illustration shows an example of the Verifier cellview.



Now, you select the preference Allow editing the enabled or disabled state for implementation run or test and check boxes in the *Run* column become editable, as shown below.



Deselect the run plan `Run.0` and select `Run.1`. The following illustration shows the pale blue background added to highlight the changes.



© 2023
All Rights Reserved.

You can proceed to run simulations on these variants of the tests and run plans defined in Assembler and view the results in Verifier.

Additional Information

- You can revert the local changes by selecting the implementation and choosing the shortcut menu command *Restore Enabled or Disabled State*.
- When you deselect the *Run* check box and switch to a different history for an implementation, the states are reverted to the original settings from the implementation cellview. After this, the local changes of the states are cleared.
- Choosing the shortcut menu command *Update Implementation* in the *Implementations* pane does not change the locally disabled state for existing runs or tests.
- It is recommended that you also select Include disabled run, reliability, test, or output for implementation when using this preference.
- Deselecting the Allow editing the enabled or disabled state for implementation run or test preference disables the check boxes in the *Run* column and restores the enabled or disabled state from the implementation.

Adding Implementations

In a bottom-up flow, you add implementations and then create requirements from implementations. For details, see Creating Requirements from Implementations.

This section describes the following topics:

- Adding an Implementation Manually
- Implementations with Wave Specifications
- Implementations with ADE Assembler Run Plans
- Creating Outputs in Implementations Using Requirements
- Editing Outputs in Implementations
- Adding Multiple Implementations
- Importing Implementations from Another ADE Verifier Cellview

It is possible that an implementation cellview you added was renamed, moved, or deleted. In this case, Verifier prompts you to specify the new location and name, remove the reference,

© 2023 All Rights Reserved.

or cancel the prompt. When you specify the new location or name, the mapping information is retained.

Adding an Implementation Manually

You can add an implementation cellview to the implementations list manually. The entry for the cellview history must be unique in the list.

To add an implementation manually:

1. Click the *Setup* tab to display the implementations area.
2. Right-click in the empty area or an implementation cellview and choose *Add Implementation*.

   The Add Implementation Form form displays.
3. Select the library, cell, and view of the implementation.
4. Click *OK*.

   If the selected cellview has a single `Active` history, it is used as the default history for the implementation. If it has multiple histories, the Implementation History Item form displays. Select the history that you want to use and click *OK*.

The implementation cellview appears as the last entry in the implementations list.

When you run the cellview from Verifier, a session is loaded with the implementation cellview where the specified history is set as the active history within that session. If you do not want to initiate the simulation from Verifier, you load the results from the selected history.

Note: You can change the history to be used for an implementation cellview. For this, show the *History* column in the implementations pane and select the history you want to use for that implementation cellview from the *History* drop-down list.

Adding Multiple Implementations

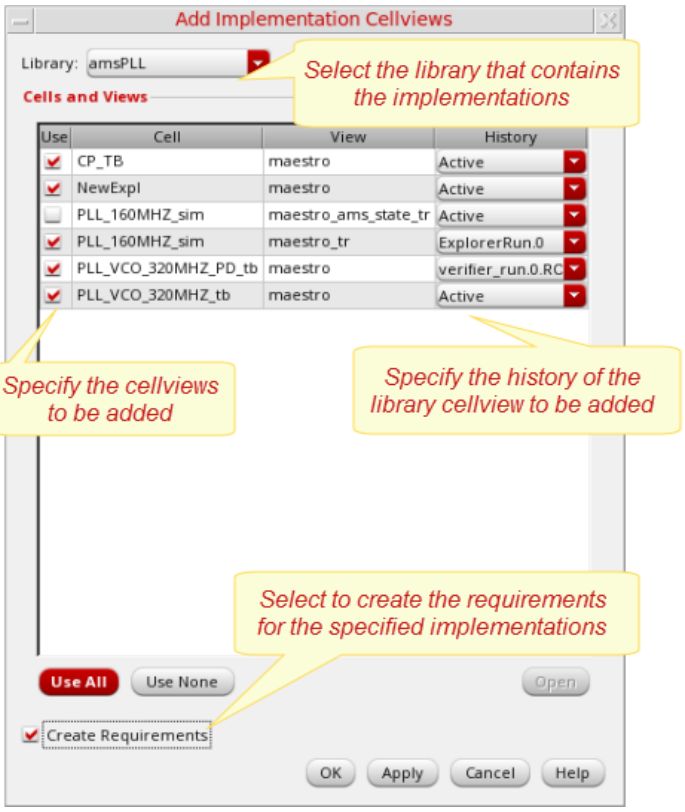
Verifier lets you add multiple implementation cellviews using the Edit Typical Value Setup Form form. You can also create their corresponding requirements.

To include multiple implementations:

1. To start importing implementations in the currently opened Verifier session, choose *Tools – Add Implementation Cellviews*.

© 2023 All Rights Reserved.

The Edit Typical Value Setup Form form appears.



2. Specify the necessary information to add multiple implementations:
   a. Select the library that contains the implementation cellviews.
   b. Select the cellviews you want to add as implementation cellviews.
   c. Select *Create Requirements* to create the requirements corresponding to the implementations. The implementations are also mapped with the corresponding requirements.
3. Click *OK*.

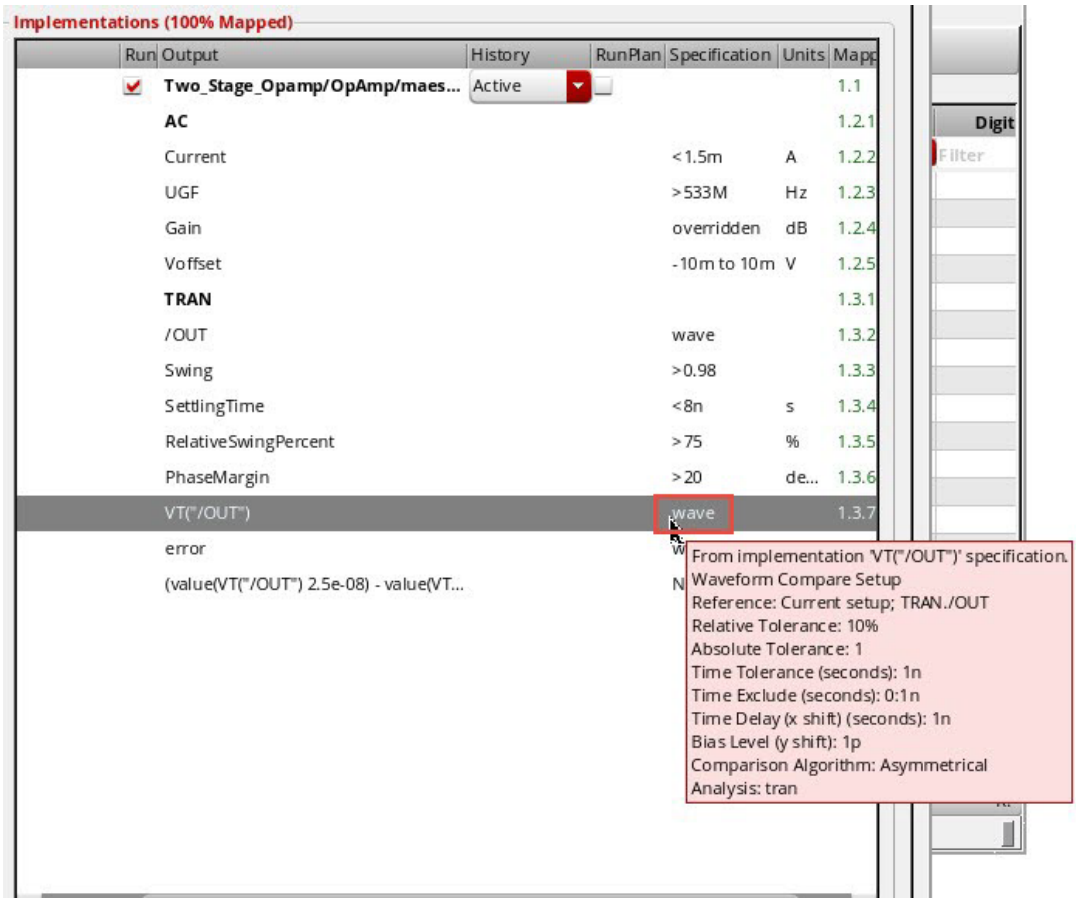
Based on the specified information, Verifier adds the implementations, and optionally creates and maps requirements.

Implementations with Wave Specifications

Verifier lets you add wave specifications defined for signals and wave expressions defined in maestro cellviews. These specifications appear as `wave` in the implementation tree for the

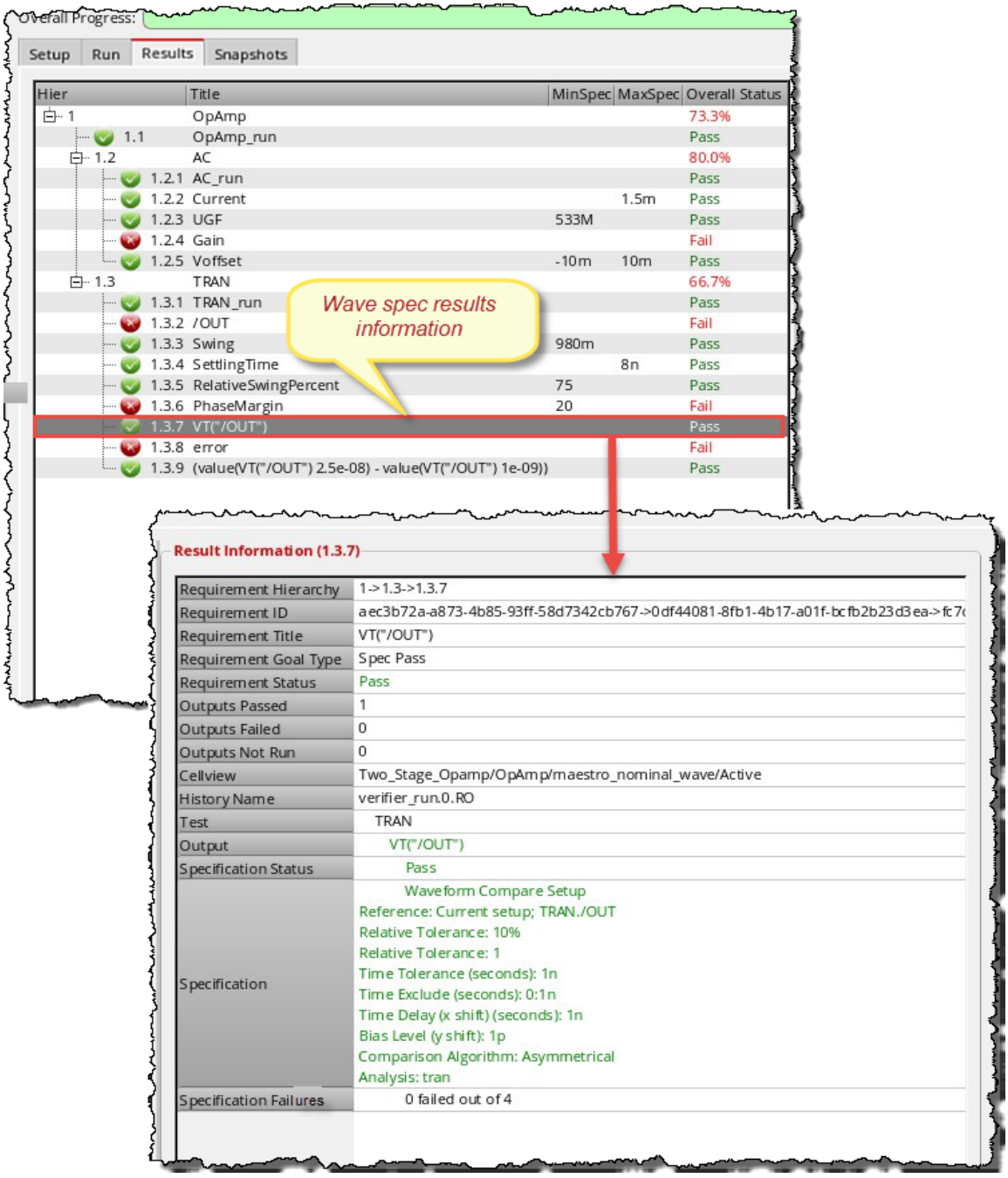
© 2023 All Rights Reserved.

outputs and can be mapped to a requirement of type *Spec Pass* and *Ran OK*. However, the spec check fails if you map the output with `wave` spec to a requirement of the *Spec Pass* containing the *MinSpec* and/or *MaxSpec* values. For details, refer to Comparing Waveforms Using Wave Specifications.



© 2023
All Rights Reserved.

On the *Results* tab, you can view the wave specification in the results tree and also view the details of Waveform Compare Setup in the *Information* assistant.

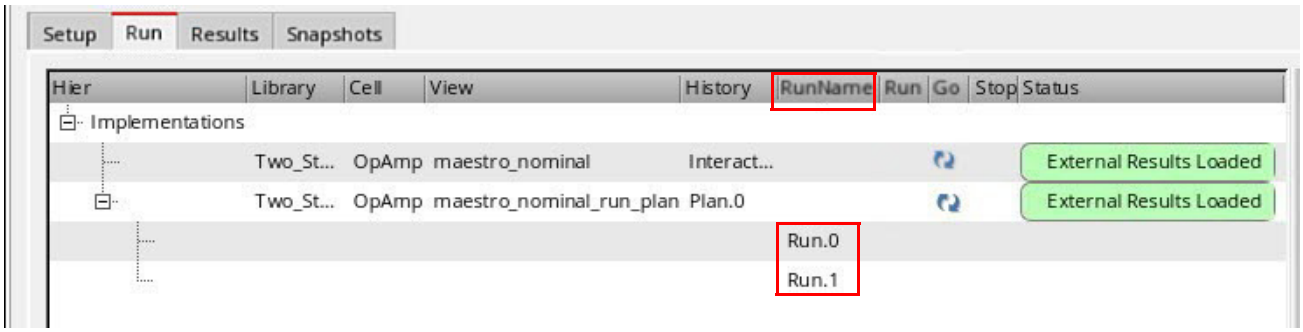


Requirements that contain `wave` in the *Specification* column of the *Implementations* pane are skipped during evaluation of metric prefixes. See Evaluation of Metric Prefix.

© 2023 All Rights Reserved.

Implementations with ADE Assembler Run Plans

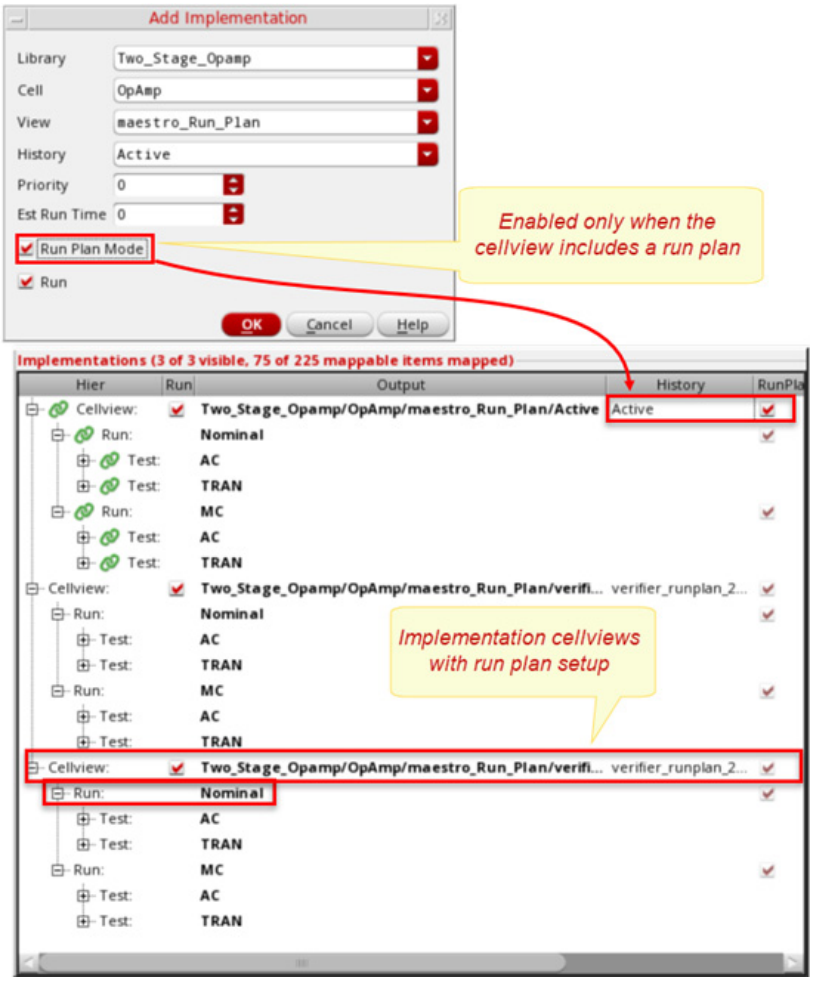
Run plans in ADE Assembler provide the capability to create multiple variations of a setup within a single session, where each run has its own setup details that override the settings in the active history setup. You can also run an implementation cellview that includes Assembler run plans similar to other Verifier cellviews. On the *Run* tab, the *RunName* column shows the run name for the implementation cellviews with run setup. For details, see Implementations with ADE Assembler Run Plans and Working with Run Plan.



You can add ADE Assembler cellviews with run plan setup in Verifier. The *Run Plan Mode* check box is available in the *Add Implementation* form. This check box is disabled by default. However, when you add the cellviews with run plan setup, the *Run Plan Mode* check box is enabled.

© 2023
All Rights Reserved.

When you add cellviews with run plan setup, the check box in the *RunPlan* column of the cellview in the implementation setup gets selected. *Run* name also gets listed in the *Hierarchy* tree.



The *RunPlan* column shows whether a run plan is to be used for an implementation. If you choose *Active* in the *History* column, you can enable or disable the run plan mode and run the implementation accordingly. For other setups of history, the check box is selected automatically.

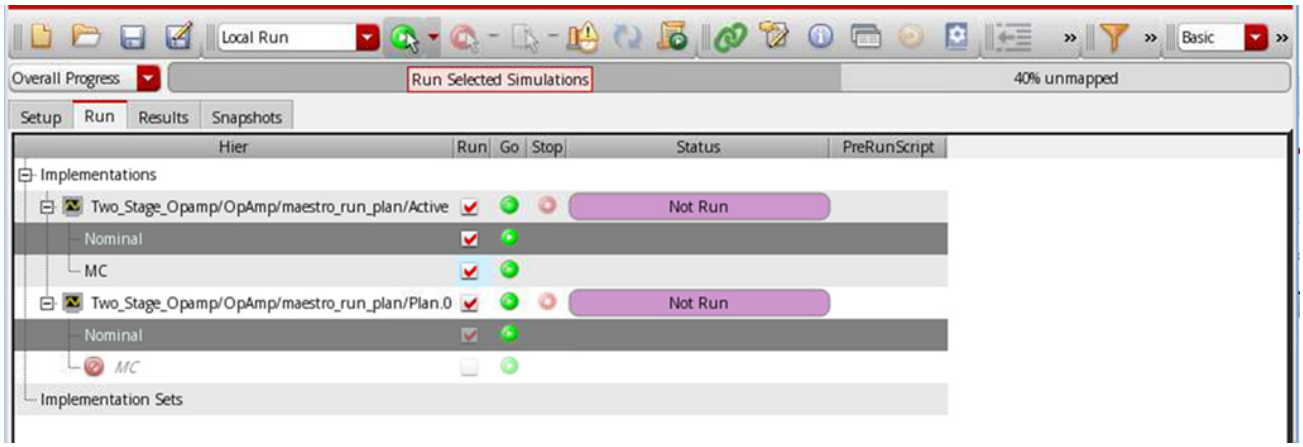
The *Run* tab also shows the *Run* button for each *RunPlan* run. However, the *Run* button is disabled when the run is disabled locally within Verifier or in ADE Assembler. Additionally, the *Run* button is not displayed for those implementations for which the *Run* check box is disabled.

When the preference Allow editing the enabled or disabled state for implementation run or test is enabled, you can simulate a *RunPlan* run even when it is disabled within

© 2023 All Rights Reserved.

Verifier. You can select multiple disabled runs for simulation. The following illustration shows a simulation running for an implementation and a *RunPlan* run. The *Run* check box is deselected for both these runs.

You can simulate a *RunPlan* run only when it is enabled within Verifier or ADE Assembler. You can select multiple runs for simulation. The following figure shows a simulation running for an implementation and a *RunPlan* run. The *Run* check box is selected for both these runs.



Additional Information

- If there is an implementation cellview added with *Active* setup of history, it cannot be added again as an implementation cellview with run plan setup. Use the cellview that has already been added in the implementation tree, select another cellview to be added, or select another setup of history in the current cellview.
- All the runs including the ones skipped for simulations because of failed conditions are shown when you add the *RunPlan* history as an implementation. In addition, these runs are shown even if the *Run* check box is deselected.

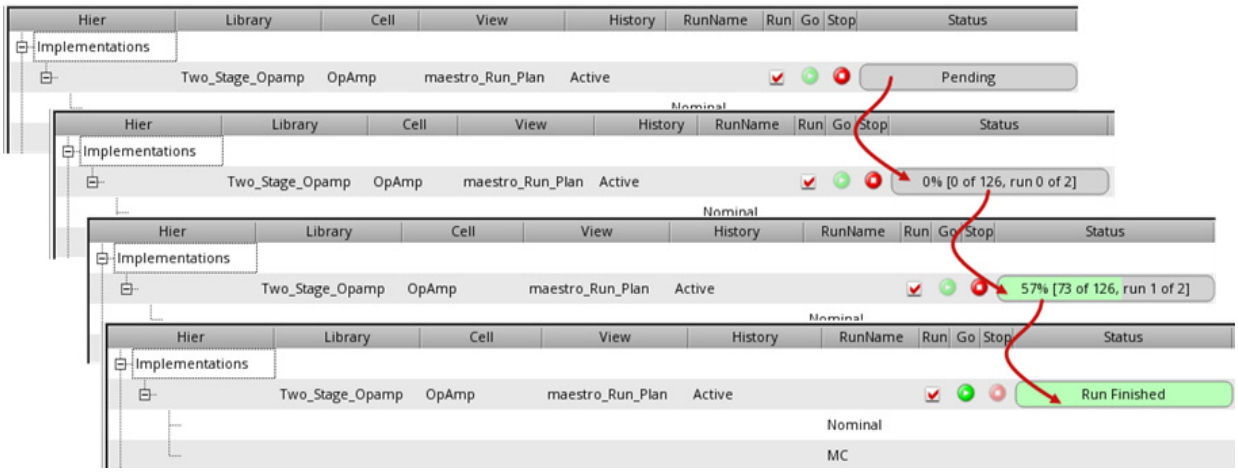
Caution

When RunPlan history is added as an implementation, the statistical outputs are not shown for the runs that have been skipped due to failed conditions even though the Run mode is Monte Carlo.

- Implementations with run plans containing run conditions might show different status during the run. Initially, only the first simulation is considered to be running. Depending on the run conditions, other simulations are sequentially started. Verifier shows the simulations that are running or queued for a run at all times. Before a run starts, you see the status as `Pending`. When the run starts and the first simulation point is complete, the Status column shows a `0%` progress. Completion of the first simulation initiates the

© 2023 All Rights Reserved.

run conditions and schedules the next consecutive simulations. As the run progresses, you see a change in the run completion percentage and the number of runs completed. After the runs are complete, you see a `100%` run status in green.



© 2023
All Rights Reserved.

Customizing Simulations with ADE Assembler Cellviews

Verifier lets you open implementations, modify these implementations, and then run simulations with the modified implementations. This requires reusing the implementation session, which is the ADE Assembler cellview. Verifier remembers the cellview that has been opened and modified and reuses the session information to run simulations.

Important

This feature ignores any pre-run scripts because they might conflict with the modification in the opened ADE Assembler cellview.

To run simulations with modified implementations:

1. Set the loadToActiveWhileViewAndReuseSession environment variable to `t` in the CIW.
2. Right-click and select the *View Implementation Results* command in the *Implementations* pane.

   The ADE Assembler cellview is opened in read-only mode and the last history of the simulation run from Verifier is loaded in the Results tab in ADE Assembler. The history can be any history except `Active`. When the history name specified in Verifier matches the history loaded in ADE Assembler, the session information can be reused to run simulations.

   The ADE Assembler cellview stays open.
3. Make changes in the opened ADE Assembler cellview.

   Modifying and reusing sessions to run simulations is only available with the *Local Run* mode in Verifier. This feature is not available with the other run modes
4. Click the *Run* button on the main toolbar in ADE Verifier.

The simulations for these opened implementations are run with the modifications, but all remaining implementations are run with the default implementation settings.

Creating Outputs in Implementations Using Requirements

In a top-down flow, adding implementations in large setups and specifying mappings takes significant time. To save this mapping time, you can add outputs into implementation cellviews by dragging requirements and dropping them on a test in an implementation. The newly

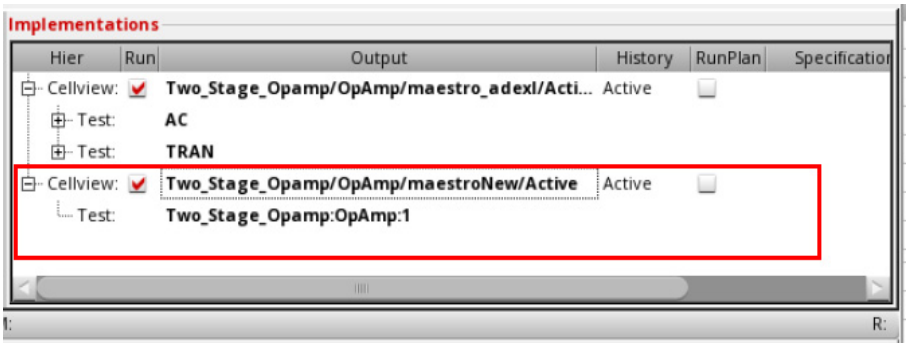
© 2023 All Rights Reserved.

added outputs are mapped to corresponding requirements automatically and added to the `maestro` cellviews in ADE Assembler.

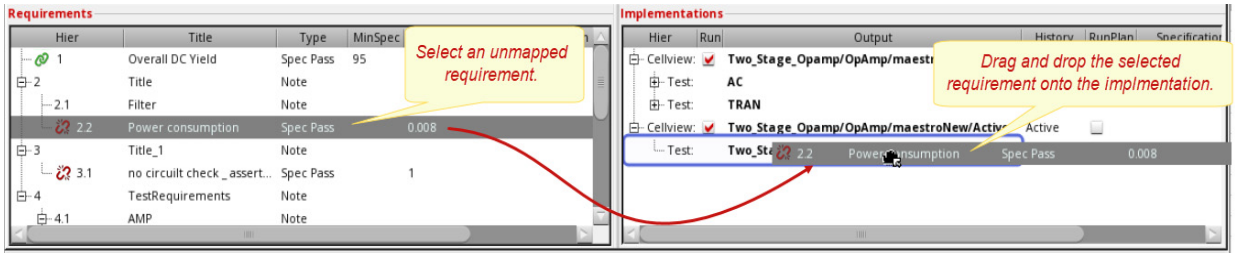
- You can drag only the requirement of type 'Spec Pass' to a test.
- Ensure that the target is a test in the specified implementation.
- Validate that the implementation is using an 'Active' history.
- You cannot create outputs for a history except when 'Active' and when Run Plan is enabled.

To create outputs for implementations:

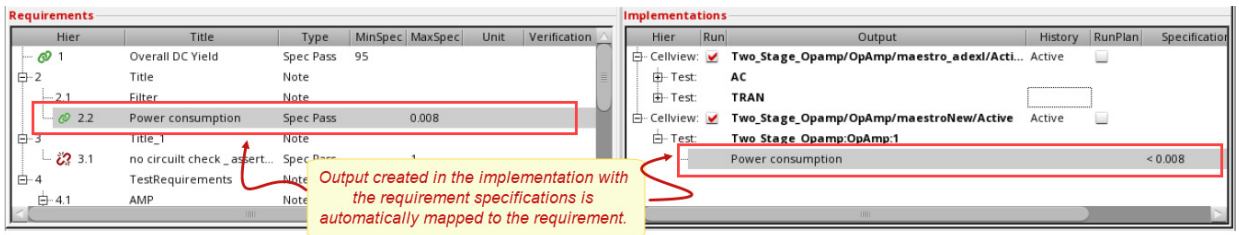
1. Add an implementation in the Verifier cellview. The implementation can be completely empty or can contain some testbench information.



2. Drag and drop requirements into the implementation.

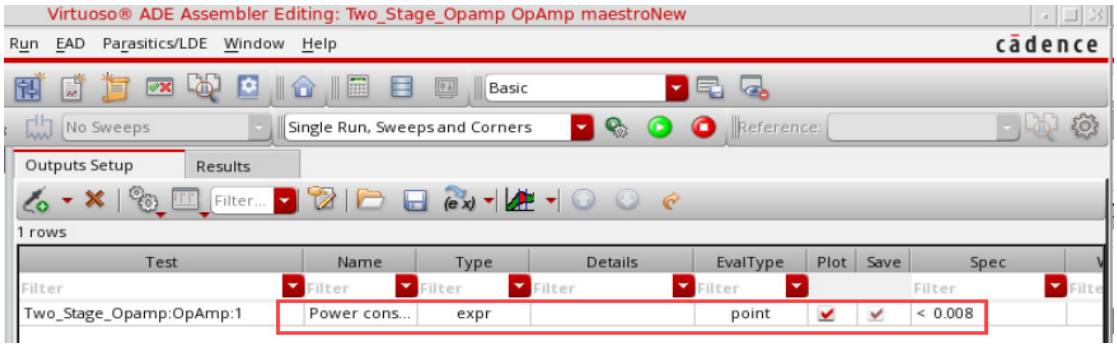


Output expressions are created for those implementations and mapped to the requirements automatically.



© 2023
All Rights Reserved.

The `maestro` cellview in ADE Assembler is updated automatically.



Editing Outputs in Implementations

To edit outputs:

1. Select the implementation that has the output that you want to modify in the *Implementations* pane.
2. Do one of the following:
   - Double-click the value in the *Details*, *Specification*, or *Unit* column to specify the new value directly in the *Implementations* pane.
   - Right-click and select *Modify*. This opens the Implementation Editor in *Modify* mode. Specify the new value in the *Details*, *Specification*, or *Unit* field.
3. Click another field or press `Enter` to save the details.

The implementation is updated with the new value.

Important

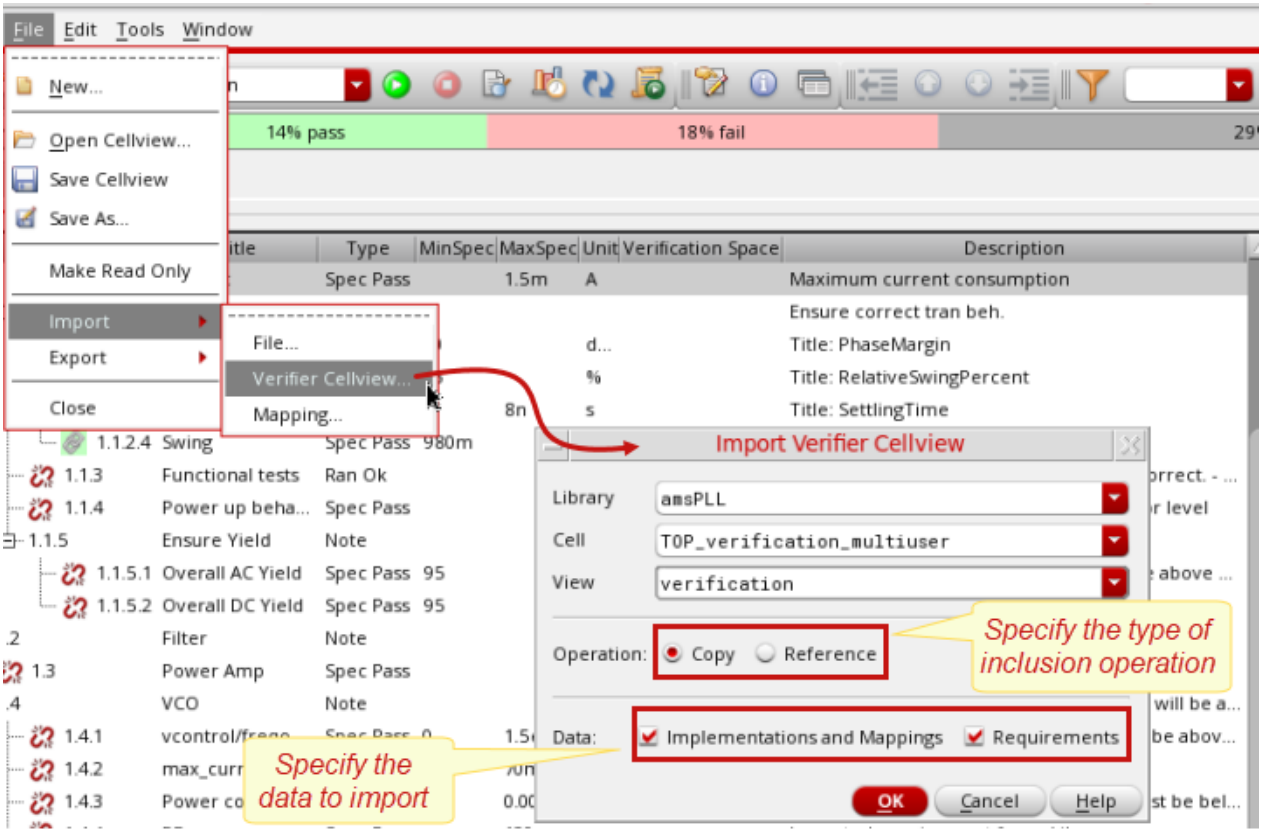
You can edit outputs only for implementations with the *Active* history setup. Ensure that you edit output values carefully, because modifications in an implementation also update the corresponding implementation cellview. Every change opens an ADE Assembler session in the background and closes it after the implementation cellview is updated. Opening and updating the implementation cellview may take time, and may impact the performance of this operation.

© 2023
All Rights Reserved.

Importing Implementations from Another ADE Verifier Cellview

Verifier lets you import implementations from another cellview of the type `verifier`. You can copy or reference the implementations stored in another cellview. To change a referenced implementation, you must modify its source.

The following figure shows how you import implementations from another cellview.



To include the implementations from another cellview:

1. Choose *File – Import – Verifier Cellview*.

   The Import Verifier Cellview Form form displays.

2. Choose the *Library*, *Cell* and *View* to specify the source cellview.

© 2023
All Rights Reserved.

3. Do one of the following:

   - Click *Copy* to copy the implementations from the source cellview. Use this option if you want to modify the imported list.

     - Select *Implementations and Mappings*.
     - Select *Requirements* if you want to import the requirements from the source cellview.

       The implementations–requirements mappings in the source will be maintained.

   - Click *Reference* to include the implementations by reference. Use this option if you want to maintain the implementations list in the source. The implementations–requirements mappings in the source will be maintained.

     Note: The implementations and requirements imported using the *Reference* option will be in read-only mode.

4. Click *OK*.

The implementations-requirements are included in the current Verifier session. The referenced implementation cellviews and tests appear as *Read-Only* in the *Implementations* and *Requirements* panes, and the Requirements Editor assistant.

Additional Information

- A Verifier session can have only unique implementation cellview–history entries.
- Verifier includes the requirement–implementation mappings defined in the source cellview if the corresponding requirement IDs exist in the current cellview.
- You cannot change the referenced implementations list. To change the list, modify the source cellview. For example, you cannot overwrite the implementation specification with the specification of its mapped requirement.
- If a referenced implementation cellview was renamed, moved, or deleted, Verifier lets you choose Edit Cellview or Update Cellview. When you edit or update the cellview, the mapping information is retained.

© 2023 All Rights Reserved.

Implementation Run Modes

Verifier supports the *Single Run, Sweeps and Corners* and the *Monte Carlo Sampling* run modes.

When an implementation cellview is set to use the *Single Run, Sweeps and Corners* run mode, you can map its outputs to requirements. For verification, Verifier compares the defined output specification values with the corresponding simulation results.

When an implementation cellview is set to use the *Monte Carlo Sampling* run mode, Verifier displays the following sections:

- *Output Values*: This section displays the outputs, with values, of the tests in the implementation cellview.
- *Statistical Values*: This section displays the yield, mean, standard deviation, and CPK of each output measurement of the tests in the implementation cellview. It also shows the overall yield of each test.

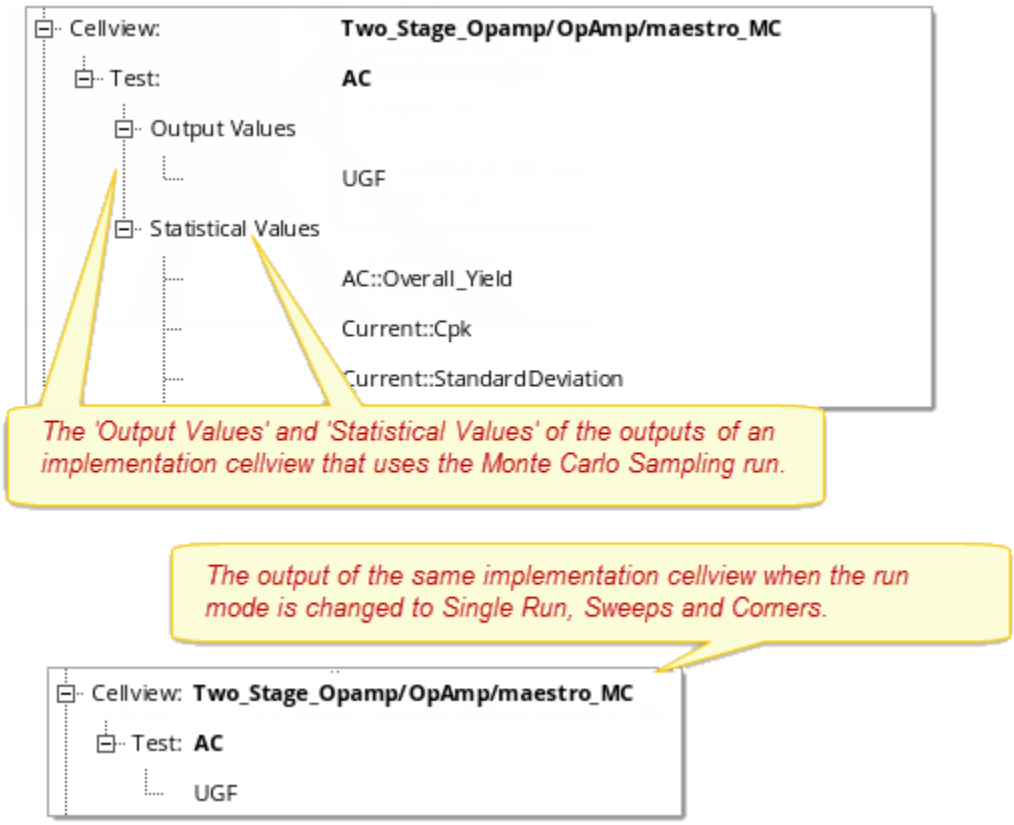
You can map Monte Carlo outputs and statistical values with requirements. Ensure that the requirements that are mapped to Monte Carlo statistical values have specifications.

The specification values used for these statistical parameters are different from the output specification values in ADE Assembler or ADE Explorer. For example, consider that you have a specification value of `DCGain` as `>10` in ADE Assembler. Your Monte Carlo simulation can provide you a yield measurement of `98%` for `DCGain`. The specification value specified in Verifier for such a statistical parameter references the statistical values. This means that a specification value of minimum `0.97` for the requirement mapped to `DCGain::Yield` will ensure that the requirement passes.

You can start the *Single Run, Sweeps and Corners* or the *Monte Carlo Sampling* run of an implementation cellview from Verifier. It is also possible to run the implementation cellview

© 2023 All Rights Reserved.

outside Verifier and load the results in Verifier. For Monte Carlo run mode, Verifier can retrieve the statistical values from the relevant history of the cellview.



Mappings between Implementations and Requirements

You can map implementation cellviews, tests, and outputs to their corresponding requirements of the type *Spec Pass*, *Ran OK*, and *Manual*. If the implementation cellview uses the *Monte Carlo Sampling* run mode, you can map that cellview, its tests, output values, and statistical values with requirements. Ensure that the requirements that you want to map to Monte Carlo statistical values have specifications. For details, see Implementation Run Modes.

It is possible to map multiple implementations with multiple requirements, unless the *Map only one requirement to one implementation* preference option is enabled. If you want to map multiple requirements with an implementation, ensure that the requirements have the same specification.

© 2023 All Rights Reserved.

If the verification project is set for a multi-reference environment, you can use Verifier as a requirements manager for defining the mappings for requirements. In the master cellview, you can also set mappings for local requirements.

Verifier uses the mapping, along with simulation results and preference settings, to determine the verification status of requirements. You can aim to achieve 100 percent mapping if you want to ensure that each test and each output in each implementation cellview is mapped to a requirement. It is not necessary to achieve 100 percent mapping for all the implementations. The correctness of the overall verification status depends on the completeness of the verification plan and mappings.

The overall mapping percentage displays in the Overall Progress bar. The *Mapped* column in the implementations pane also displays the percentage of the mapped implementations in the unmapped cellviews and tests.

Related Topics

Mapping Implementations to Requirements Using the Shortcut Menu

Mapping Implementations to Requirements Using Bindkeys

Mapping Implementations to Requirements Using the Mapping Mode

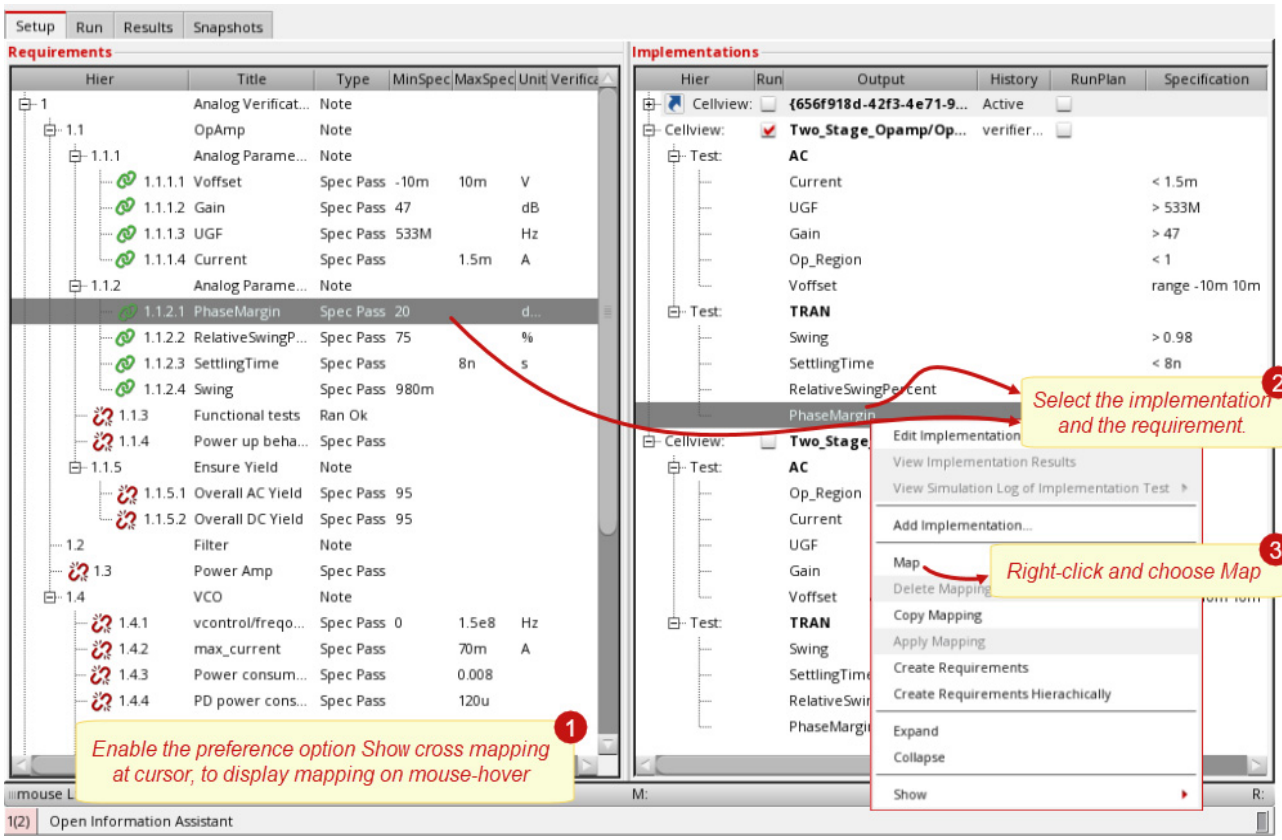
Exporting Mappings

Importing Mappings

© 2023 All Rights Reserved.

Mapping Implementations to Requirements Using the Shortcut Menu

The following figure illustrates mapping implementations to their requirements the using shortcut menu option *Map*.



To map implementations to requirements:

1. Select the requirements and implementations.

   You can use the `Ctrl` key to select multiple entries located in different places in the list, or the `Shift` key to select consecutive entries.

2. Right-click and choose *Map*.

Verifier maps the selected implementations with the selected requirements.

The mapped entries get highlighted in light gray. Click an entry to select its mapped entries.

© 2023
All Rights Reserved.

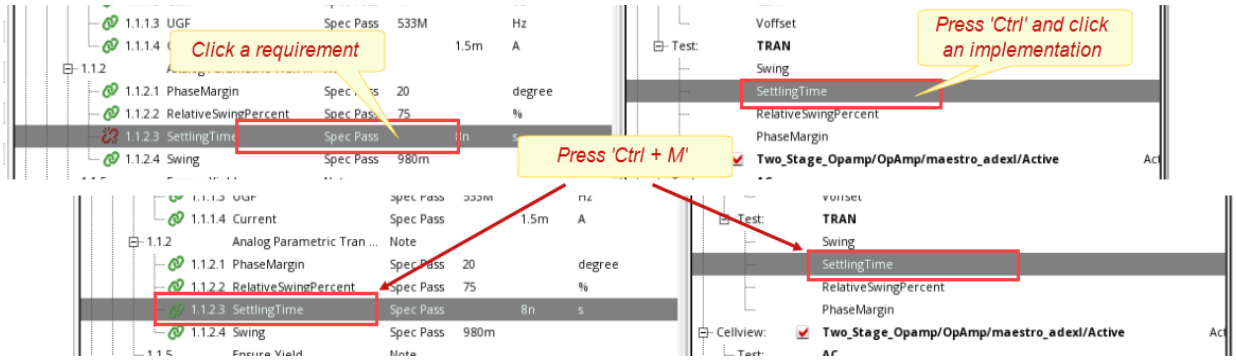
Note: Enable the preference *Show cross mapping at cursor* to show the highlighted cross mapping when the cursor or mouse moves over the mapped requirements or implementations in the Setup tab.

Mapping Implementations to Requirements Using Bindkeys

You can map implementation cellviews to requirements using the bindkey `Ctrl + M`.

To map implementations and requirements using the bindkey:

1. Click on a requirement in the *Requirements* pane.
2. Press the `Ctrl` key and click on an implementation in the *Implementations* pane.
3. Press `Ctrl + M`.
   The requirement and implementation are mapped as illustrated below.



Mapping Implementations to Requirements Using the Mapping Mode

Verifier lets you expedite the mapping of implementation cellviews to requirements using an interactive Mapping Mode.

To use the interactive *Mapping Mode*:

1. Do one of the following:
   - ❑ Press 'M' or 'm' to enter the mapping mode.

© 2023 All Rights Reserved.

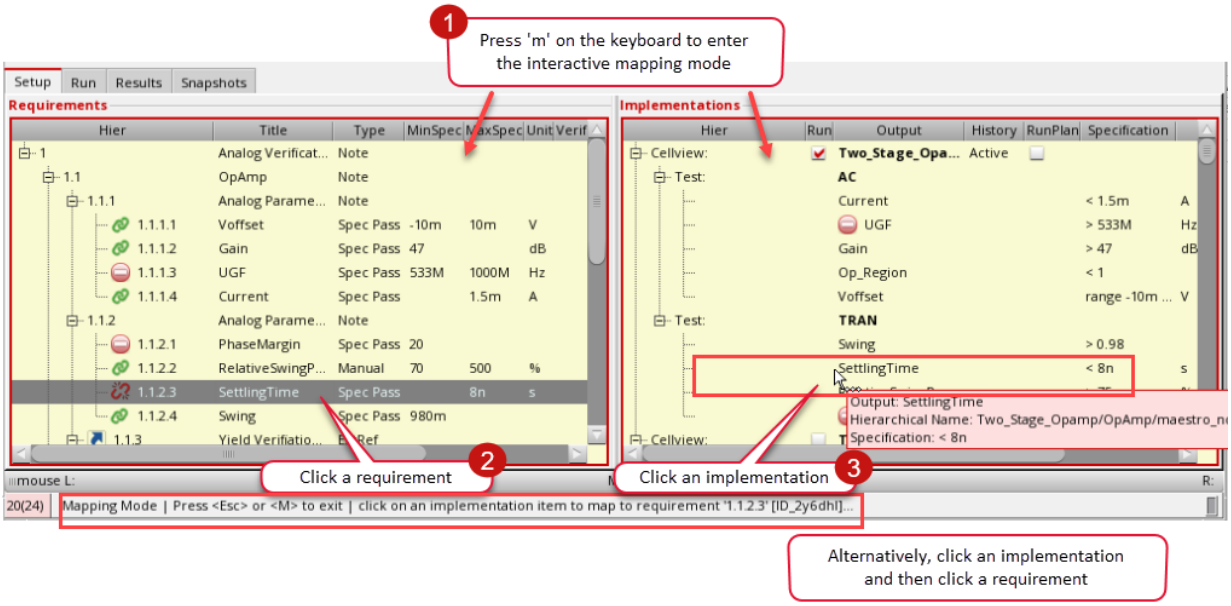
- ❑ Choose *Edit – Enter Mapping Mode*.
- ❑ Click *Enter Mapping Mode*( ) on the toolbar.

A yellow background outlined in red on the *Requirements* and *Implementations* panes indicates that the *Mapping Mode* is enabled.

2. Do one of the following:

- ❑ Click a requirement and then click an implementation.
- ❑ Click an implementation and then click a requirement.

The requirement and implementation are mapped.



3. To exit the Mapping Mode:

- ❑ Press the 'M', 'm', or 'Esc' keys.
- ❑ Choose *Tools – Exit Mapping Mode*.
- ❑ Click *Exit Mapping Mode*( ) on the toolbar
- ❑ Right-click and select *Exit Mapping Mode*.

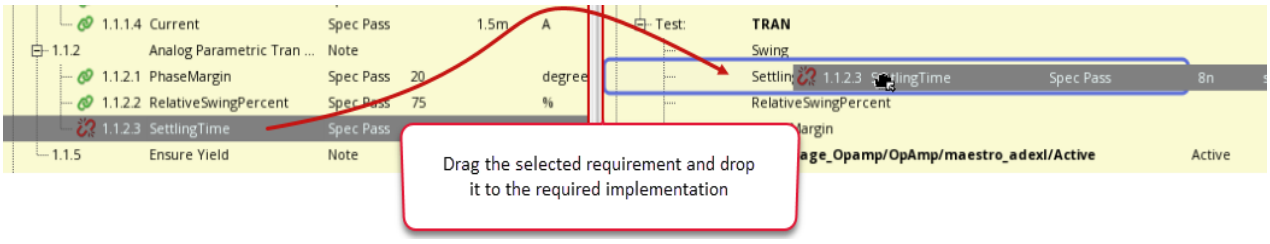
4. To delete an existing mapping, right-click the mapped requirement or implementation and select *Delete Mapping*.

© 2023
All Rights Reserved.

Tip

It is recommended that you exit the *Mapping Mode* when mapping of requirements is complete. The shortcut menu options accessible in the *Requirements* and *Implementations* pane are not available when the *Mapping Mode* is on.

You can also use the Mapping Mode to drag and drop requirements on to implementations to create mappings. To use the drag-and-drop method in the Mapping Mode, select a requirement and drop it onto a required implementation. This automatically maps the requirements and the implementation. For more information, see Reassignment of Requirements.



Note: Drag-and-drop method of mapping requirements and implementations is only available in Mapping Mode.

Managing Mappings

To copy the implementation–requirement mappings:

1. Select the requirements or implementations.
2. Right-click and choose *Copy Mapping*.

Verifier copies the mappings of the selected requirement or object in the implementation.

To apply the copied implementation–requirement mappings:

1. Select the requirements or implementations.
2. Right-click and choose *Apply Mapping*.

Verifier applies the copied mappings to the selected requirement or object in the implementation.

To remove the implementation–requirement mappings:

1. Select the requirements or implementations.

© 2023 All Rights Reserved.

2. Right-click and choose *Delete Mapping*.

Verifier removes the mappings from all the selected requirements and implementations.

Exporting Mappings

Mappings between requirements and implementations in a cellview can be saved and exported as a CSV or Microsoft Excel file to be used in other cellviews, if required.

To export the implementation–requirement mappings in the Verifier setup:

1. Choose *File – Export – Mapping*.

   The *Export Mapping* form is displayed.

2. Choose the required file format.
   The available formats are *CSV* and *Excel*.

3. Click the browse button and specify a new or existing file name in the *Specify a File* form.

4. Click *Save*.

5. Click OK on the *Export Mapping* form.

The implementation-requirement mappings are saved in the specified file format.

Importing Mappings

Exported mappings between requirements and implementations from a cellview can be used in a Verifier setup by importing the mappings using a CSV or Microsoft Excel file, if required.

To import the implementation–requirement mappings in the Verifier setup:

1. Choose *File – Import – Mapping*.

   The *Import Mapping* form is displayed.

2. Choose the required file format.
   The available formats are *CSV* and *Excel*.

3. Click the browse button and select an existing file in the *Specify a File* form.

4. Click *Open*.

5. Click OK on the *Import Mapping* form.

© 2023 All Rights Reserved.

The implementation-requirement mappings are imported in the current setup.

These mappings are in a multi-column format in the file, where the first column is for requirements, and the remaining are for implementation-related details. Here every row consists of a single requirement and mapped implementation details. A requirement that is mapped to multiple implementations has the same requirement name in the first column and the various implementation details in the subsequent columns.

The following list of columns are available in the CSV or Microsoft Excel:

```
Requirement,Library,Cell,View,History,Run,Test,StatisticalTest,Output,Statistical
Output,Hierarchy,Title,Type
```

Here, the `Title` and `Type` columns are ignored because they contain the requirement title and type that are not required for mapping.

Additionally, you can use wildcards to edit the mappings by manually modifying one or more of the following columns in the CSV or Microsoft Excel:

```
Library,Cell,View,History,Run,Test,StatisticalTest,Output,StatisticalOutput
```

For example, consider the following values in the columns for requirement ID1:

```
ID1,Two_Stage_Opamp,OpAmp,maestro_MC,*,,AC,,,,1.1,Title,Ran Ok
```

Here, the value in the History columns is set to *. This will map requirement `ID1` to all histories with the matching implementation details.

Opening Implementations

Verifier lets you open an implementation cellview in edit mode. You can also view the simulation results of an implementation in read-only mode. When Verifier opens an implementation cellview, it loads the history you have selected for that implementation.

To open an implementation in edit mode:

➡ Right-click the implementation and choose *Edit Implementation Cellview*.

The implementation cellview opens its application.

To view simulation results of an implementation cellview whose results are loaded:

➡ Right-click the implementation and choose *View Implementation Results*.

To view the simulation log of an implementation cellview whose results are loaded:

➡ Right-click the implementation and choose *View Simulation Log of Implementation Test,* and select the test.

© 2023 All Rights Reserved.

The implementation cellview opens in read-only mode and the results display in the relevant application.

If the implementation cellview uses the integrated history management, you receive a popup asking you to convert the cellview to use the separate history management scheme.

Updates in Implementations

Verifier periodically checks for saved changes in the implementation cellviews and updates the setup accordingly.

Updates in the implementation cellview are indicated by the  icon. To indicate that the setup needs to be updated, this icon is displayed before the cellview node in the implementation tree. Update the setup by clicking the implementation, and choosing the shortcut menu option *Update Implementations* in the Setup tab. When you update implementations, Verifier does a quick check for results. Implementations that have the *Run* check box enabled are always marked as out of date because the setup has changed. For implementations that have the *Run* check box disabled, Verifier does a quick check with the history name and time stamp to ensure that latest simulation results are used in Verifier. Verifier does a full check with *Report Identical History* when you enable the Report identical histories before run preference.

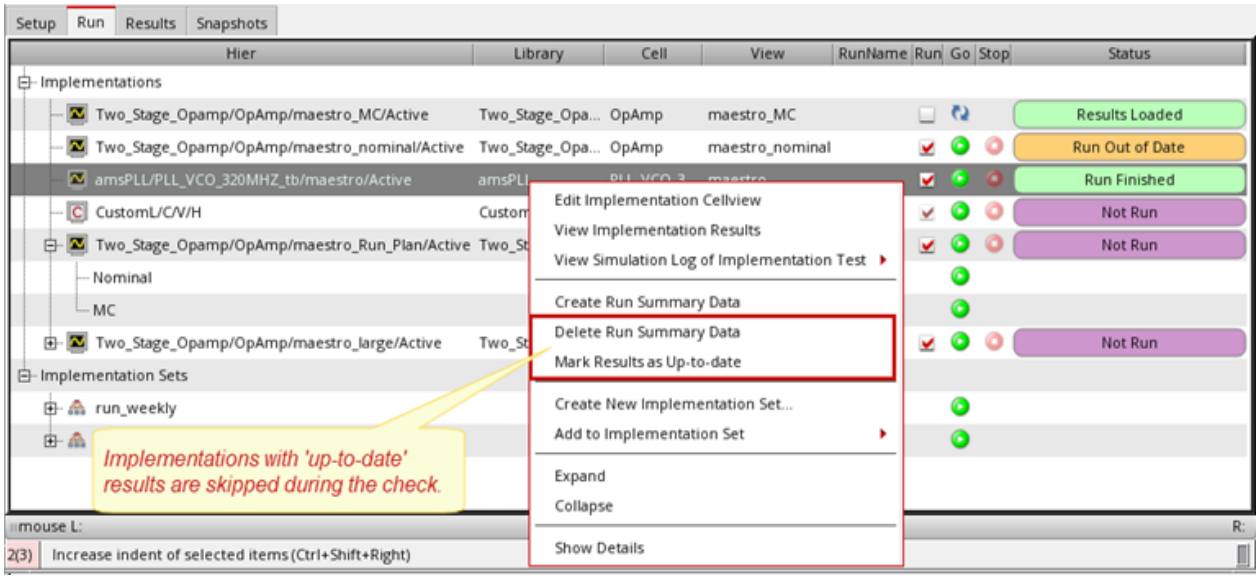
To manually check for updates in the implementation cellview, you can choose the *Check for Changes in Implementations and Simulation Results* option from the *Tools* menu. When you check for changes in implementations and simulation results and the results show an 'out of date' status, you can select *Delete Run Summary Data* from the shortcut menu on the *Run* Tab to ensure that the outdated results are not used.

The *Delete Run Summary Data* command deletes the run summary data file and the datasheet recorded inside it. You can use the environment variable autoCreateDataSheet to create datasheets when you run simulations or load results in Verifier. During report creation, Verifier creates a link to existing datasheets or triggers report generation for any missing datasheets.

To view external simulation results for an implementation without deselecting the *Run* check box, select *Create Run Summary Data* from the shortcut menu on the *Run* tab. This command loads the external simulation results from the latest history of implementation cellview.

Alternatively, you can select *Mark Results as Up-to-date* from the shortcut menu to ignore the check and use the results as valid.

© 2023 All Rights Reserved.



Restarting Implementation Jobs

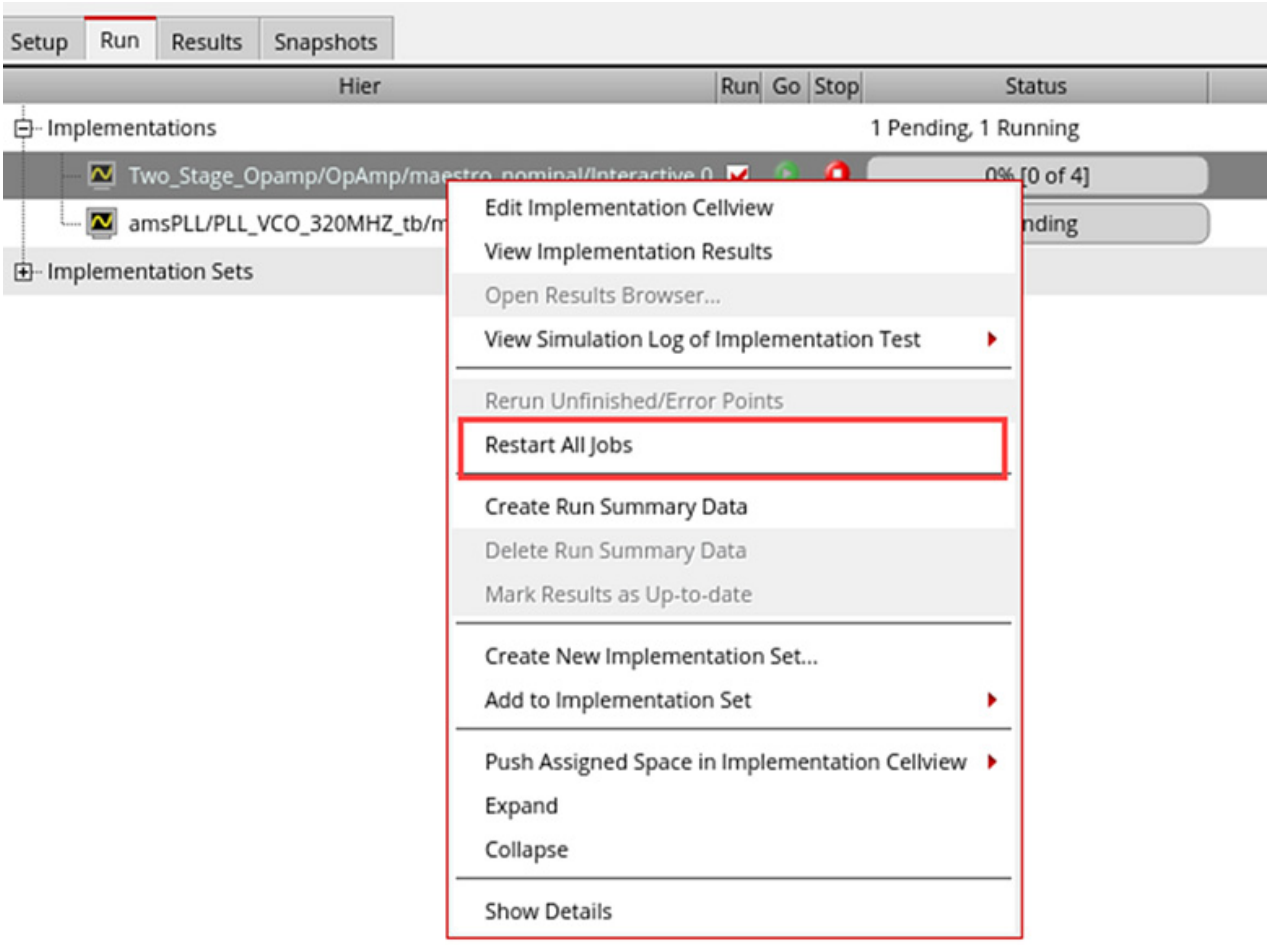
For large designs, it is possible that simulation jobs can be unresponsive in any state for a long time. You might also have some jobs that have been in the `Running` state for some time but are not completing within the expected duration, This might require that these jobs be restarted.

To restart simulation jobs:

1. On the *Run* page, select the implementation that is in the `Running` state.

© 2023
All Rights Reserved.

2. Right-click and select *Restart All Jobs* from the shortcut menu.



The *Restart All Jobs* command has a two-fold action. First, it stops all netlisting and simulation jobs for the selected implementation by triggering the *Stop All Jobs* command in ADE Assembler. Then, it restarts the simulation for the selected implementation, and the run continues with new jobs.

Related Topic

`verifRestart` (SKILL function)

Modifying Implementations

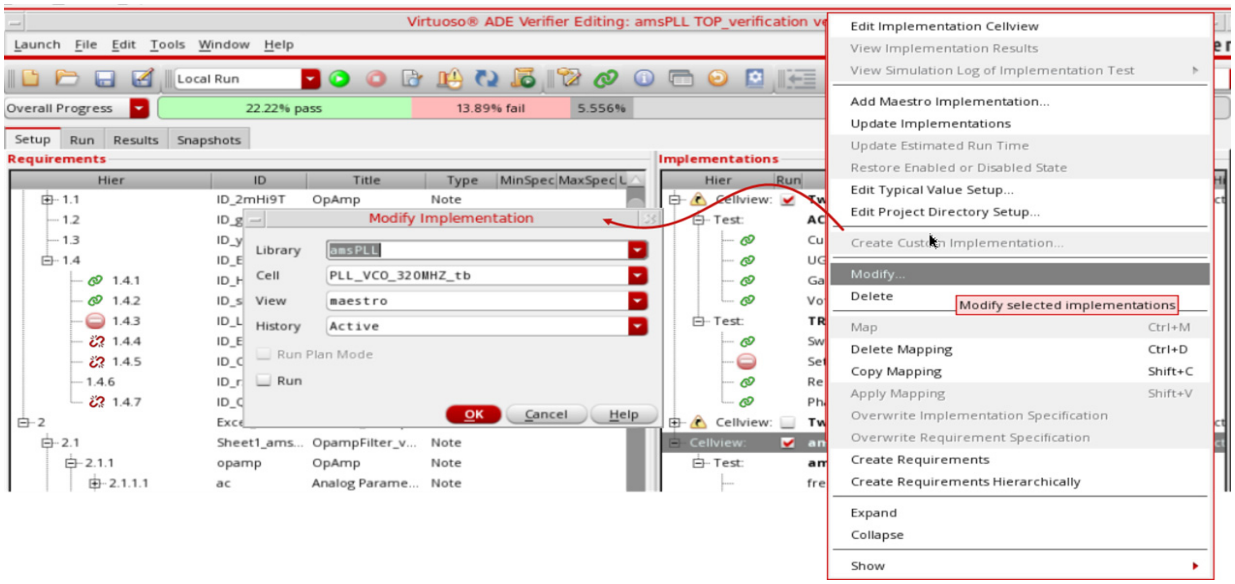
In Verifier, you can make changes to the library, cell, view, history of an implementation by using the *Modify Implementations* form. You can also replace an implementation with

© 2023
All Rights Reserved.

another implementation. The mapping is retained if the new implementation has the same setup as the previous one.

To modify a single implementation:

1. Right-click the implementation in the *Setup* tab.
2. Choose *Modify* as shown in the following illustration.



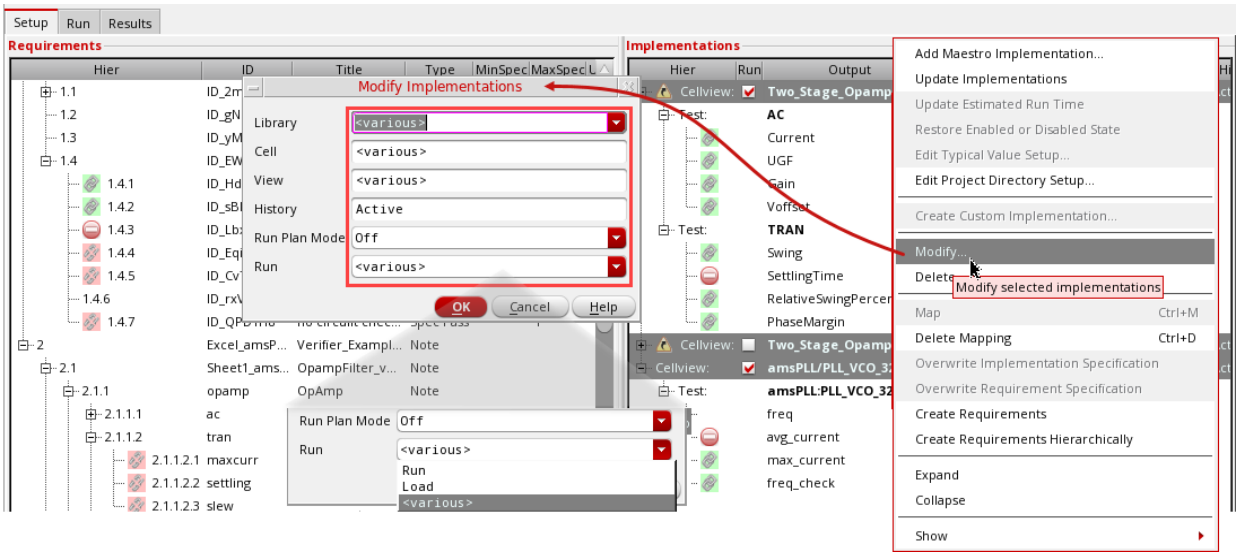
3. Specify the changes to the library, cell, view, and history.
4. Select *Run* to enable a simulation run for the selected implementation.
5. Click *OK* to modify the implementation.

To modify multiple implementations:

1. Select the required implementations while keeping the Ctrl key pressed.

© 2023
All Rights Reserved.

2. Right-click and choose *Modify* as illustrated below.



The fields in the form display *<various>* as shown below for all field properties that have different values.

3. Specify the changes to the library, cell, view, and history.
The selected implementations are modified with the values that you specify in the form. The values of those fields that display *<various>* are not changed.

4. Select *Run* to enable a simulation run for the selected implementations.

5. Click *OK* to modify the implementations.

Overriding Specifications

You can override the specification set in an implementation with the specification set in the requirement mapped to that implementation. Similarly you can set the requirement specification with the implementation specification. This functionality overrides the unit and the minimum and maximum specification values. It is possible to override the specifications of multiple requirements and implementations.

To override the implementation specification with the mapped requirement specification:

1. Right-click the implementation in the *Setup* tab.

   You can select multiple implementations and then right-click to display the popup menu.

2. Choose *Overwrite Implementation Specification*.

© 2023
All Rights Reserved.

Verifier overrides the implementation specification.

When overriding implementation specifications, these points must be considered:

- The implementation cellview must be editable.
- The specified implementation history must be 'Active' because only the specifications in active setup of implementation cellview can be overwritten.
- The selection must contain the mapped implementation outputs with specifications which are not *wave* or *overridden*.
- The requirement which is mapped to the selected output must be of the type *Spec Pass* and have either or both of the minimum and maximum limit specified.
- Selecting an implementation highlights the related requirement only. All the other implementations mapped to the same requirement are not highlighted. However, if you select a requirement, all the related implementations are highlighted. Press the Ctrl key to ensure that the selected implementations remain highlighted when you right-click and choose *Override Implementation Specification*. This is only applicable for implementations with the same specifications.

To override the requirement specification with the mapped implementation specification:

1. Right-click the implementation in the *Setup* tab.

   You can select multiple implementations and then right-click to display the popup menu.

2. Choose *Overwrite Requirement Specification.*

   Verifier overrides the requirement specification.

When overriding requirement specifications, these points must be considered:

- The current Verifier session must be editable.
- The selection must contain the mapped implementation outputs with specifications which are not 'wave' or 'overridden'.
- The requirement which is mapped to the selected output must be editable and of the type *Spec Pass.* This requirement should not be referenced or read-only.
- The specification and unit values of the outputs are also updated automatically in ADE Assembler, if the implementation is open in ADE Assembler for editing.

© 2023 All Rights Reserved.

Deleting Implementations

You can delete the implementation cellviews that you do not need for your verification project. When you remove an implementation cellview, where the cellview, tests, and outputs are mapped to requirements, the mappings are removed. You cannot hide implementation cellviews in Verifier.

To delete an implementation cellview:

1. Do one of the following:
   - Right-click the implementation cellview and choose *Delete Implementations*.
   - Select the implementation cellview and press `Delete`.

   You can select multiple implementation cellviews for deletion.
2. Click *Yes*, when prompted, to confirm the deletion.

Verifier deletes the selected implementation cellviews and associated mappings.

Filters in the Implementation List

It is possible that you have a large implementation list, where locating specific implementations is time-consuming. You can add a filter row in the implementations list to display only specific types of implementations. The filter row is applied to both the requirements hierarchy and the implementations list.

You can also locate implementations using the filter functionality of Verifier.

Related Topics

Filtering Implementations Based on Text Criteria

Filtering Implementations Based on Status

Filter Row

Filtering Implementations Based on Text Criteria

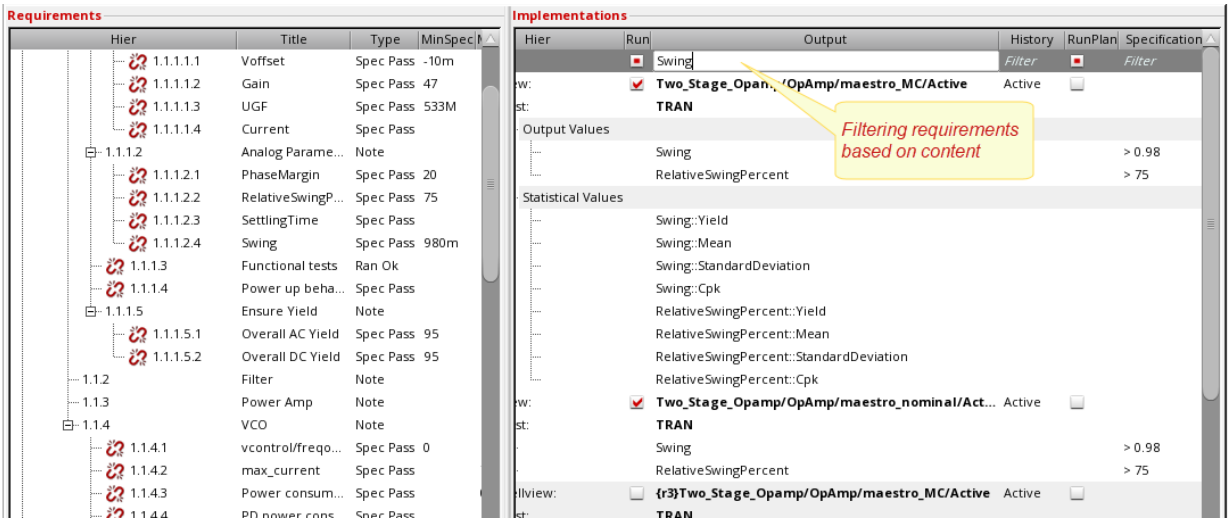
To search implementations with specific text in the hierarchy:

1. Do one of the following:

© 2023 All Rights Reserved.

- ❑ Click the *Show Filter Row* icon on the toolbar.
- ❑ Right-click anywhere on the column heading row and choose *Filter row*.

The Filter row is displayed below the column headings in the *Setup*, *Run* and *Results* tab.



2. Type the filter string or number in the required hierarchy column.

As you type, the filter functionality searches the selected field of the implementations and displays the results in the implementations pane.

To use the other features of the filter functionality:

➡ Enter a name for the filter and click *Save Filter* to store the search string for later use in the session.

➡ Click *Clear the contents of the filter* to remove all filters and show all requirements or implementations.

➡ Click *Delete Filter* to delete a saved filter.

Related Topics

Filter Row

Filters in the Implementation List

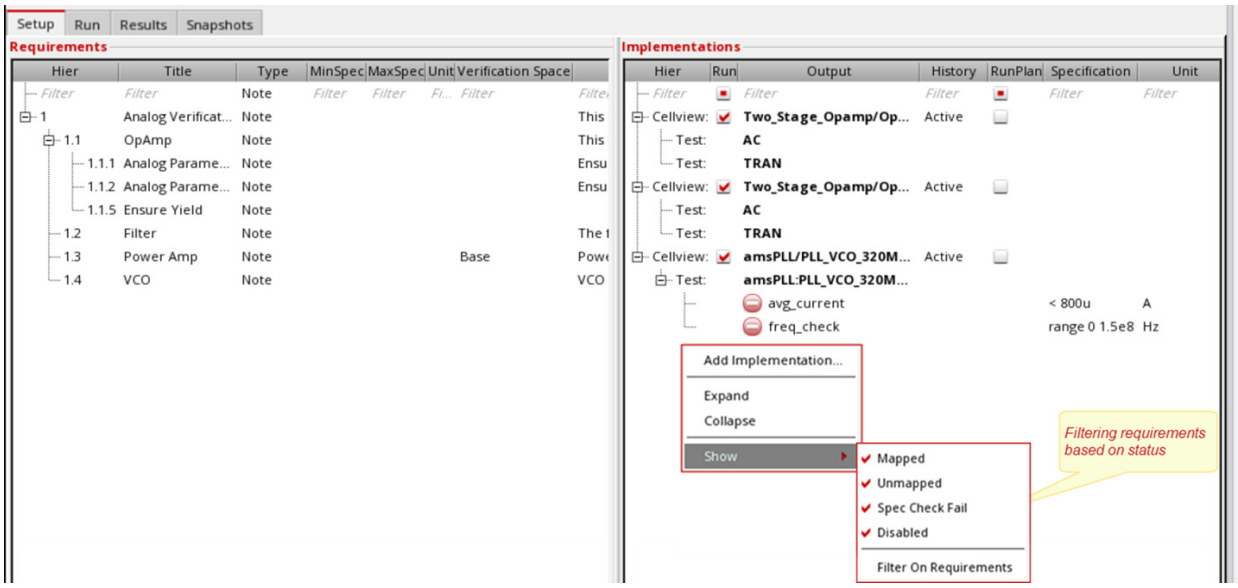
Filtering Implementations Based on Status

© 2023 All Rights Reserved.

Filtering Implementations Based on Status

To filter the implementations based on their status in the hierarchy:

1. Right-click the implementations area and choose *Show*.



2. Set the appropriate filter.

   The available filter options are:

   - *Mapped* – Filters and shows only those implementations that are mapped to a requirement.
   - *Unmapped* – Filters and shows only those implementations that are not mapped to any requirement.
   - *Spec Check Fail* – Filters and shows only those implementations that are mapped to a requirement but have failed the specifications check during verification.
   - *Disabled* – Filters and shows only those implementations that are disabled.
   - *Filter On Requirements* – Filters and shows only those outputs that are mapped to the visible requirements in the *Requirements* pane.

   Ensure that the implementation type that you want to hide is not selected.

Verifier filters the implementations list according to the specified criteria.

© 2023
All Rights Reserved.

Related Topics

Filter Row

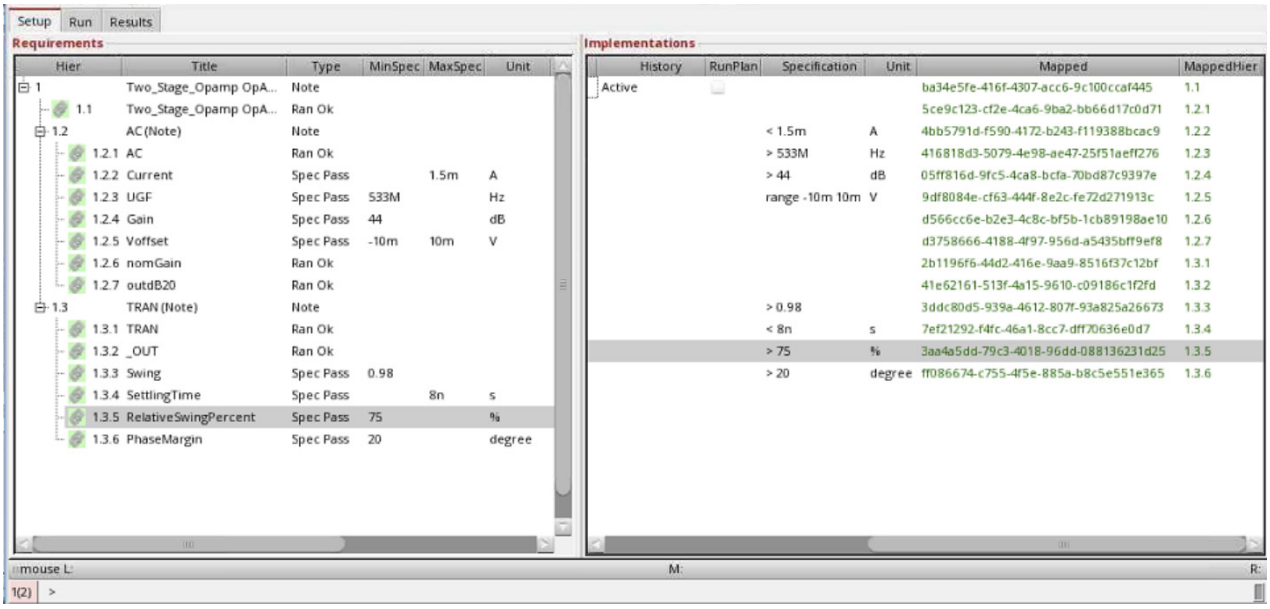
Filtering Implementations Based on Text Criteria

Filters in the Implementation List

Mapped and MappedHier Properties

In the Implementations pane, you can view the *Mapped* and *MappedHier* columns. These columns display the requirement IDs and the position of the requirement within the hierarchy. However, the data in the column changes in different scenarios.

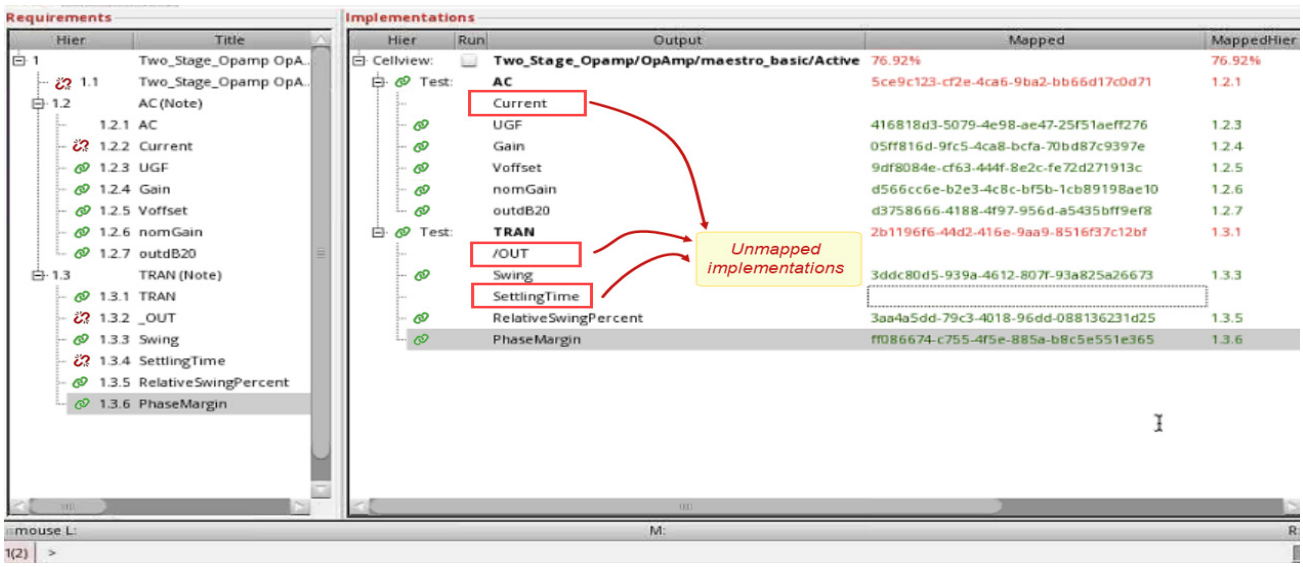
Example: When all implementations are mapped to requirements:



Here, the *Mapped* column shows the IDs of the requirement which is mapped to a specific implementation. The *MappedHier* column shows the hierarchical position of the mapped requirement in numeric format, such as 1.1, 1.2, 1.2.1.

© 2023 All Rights Reserved.

Example: When the implementation cellview and some outputs are not mapped to requirements:



Here, both *Mapped* and *MappedHier* columns in the *Implementations* pane of the cellview show the mapping status in percentages.

For unmapped implementation cellviews, runs or tests, the following calculation expression is used to compute the mapping percentage:

$$\frac{\text{Total number of mapped child items of an implementation or run or test}}{\text{Total number of mappable child items of an implementation or run or test}} = \textit{Mapped \%}$$

Here, the mappable child item can be a implementation cellview, run, test, output or statistical item.

In the above illustration, the calculation expression is:

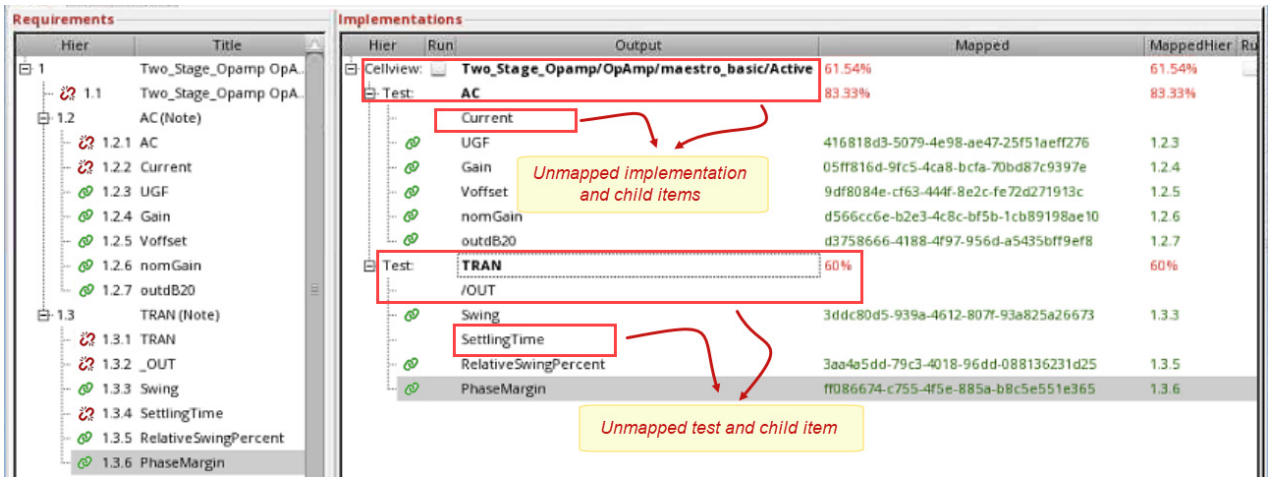
`10 / 13 = 76.92%.`

Total number of mapped child items = `10`.

Total number of mappable child items = `13`.

© 2023 All Rights Reserved.

Example: When the implementation and tests are not mapped:



Here, both *Mapped* and *MappedHier* columns in the *Implementations* pane of the cellview show the mapping status in percentages.

For unmapped implementation cellviews, runs or tests, the following calculation expression is used to compute the mapping percentage:

$$\frac{\text{Total number of mapped child items of an implementation or run or test}}{\text{Total number of mappable child items of an implementation or run or test}} = \textit{Mapped \%}$$

Here, the mappable child item can be a implementation cellview, run, test, output or statistical item.

In the above illustration, the calculation expression for the implementation is:

`8 / 13 = 61.54%.`

Total number of mapped outputs of the implementation = `8`

Total number of mappable tests in the implementation = `13`

The calculation expression for test `AC` is:

`5/6 = 83.33%`

The calculation expression for test `TRAN` is:

© 2023
All Rights Reserved.

```
3/5 = 60%
```

Note: For implementations with the Monte Carlo run mode, the 'Output Values' and 'Statistical Values' are not mappable and not counted for the mapping calculation. Only the statistical outputs are counted.

Typical Value Setup

The typical value feature in Verifier is mainly used for reporting and does not include verification. This feature is used to define a combined set of sweep or corner variables, parameters, or model files from all the implementations added in Verifier. From this combined set, you can identify a set of parameter values that match a typical point of variables or parameters defined in an implementation. This set of parameter values is assumed to be 'typical' in the current Verifier session.

To define the typical value for all implementations in Verifier, click *Tools – Typical Value Setup*. To setup the value based on the parameters, variables, or model files from the selected implementation, right-click an implementation and select the option *Edit Typical Value Setup*.

Consider these important points:

- Ensure that you specify a common definition for typical across all the implementations in a project.
- *Model Files* is a file name that includes the absolute or relative path. The Corners Setup in ADE Assembler and the Typical Value Setup in Verifier display only the name of the model file as shown below:

```
        <model>
               <name>gpdk045.scs</name>
...
               <modelFile>./gpdk045.scs</modelFile>
               <modelSection>fs</modelSection>
        </model>
```

  However, for comparison of the typical values, Verifier uses `modelFile` and not the `name`.

- In the current Verifier session, there should only be one common definition for typical across all the implementations.
- In this form, *Model Files* is not editable and would only be set in the Edit Typical Value Setup form when the selected corner has model files.
- The typical definition is a global definition per Verifier session. In most cases there would even be only one global definition of 'typical' for a given project/group.

© 2023 All Rights Reserved.

- In order to reuse the typical setup between multiple Verifier sessions within the same project, the settings can be exported and imported.
- In this form, the item with an empty value is ignored during comparison with the real parameters used for each data point.
- If the setup is modified, the implementations on the *Run* tab are marked as `out-of-date`. You need to re-run or re-load the implementations to evaluate the typical value with new settings.
- For a normal value coming out of a Monte Carlo run, `various` is returned as a typical value because of multiple matched points.

  The typical value is shown in the *Typical Value* column of the *Results* tab and the *Information* assistant. Also, it could be exported to an CSV or Excel file or other reports.

Related Topics

Setting Up the Typical Value for Implementations

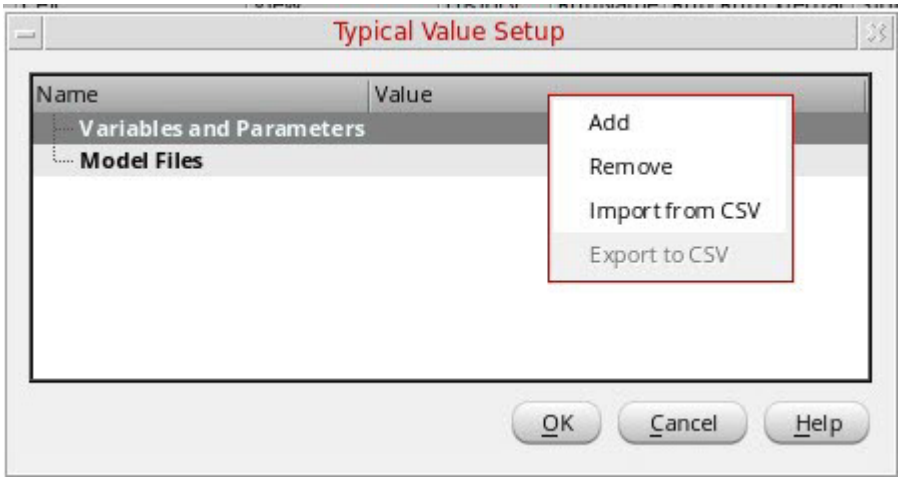
Edit Typical Value Setup Form

Setting Up the Typical Value for Implementations

To import the typical data from a CSV file or to add it one by one:

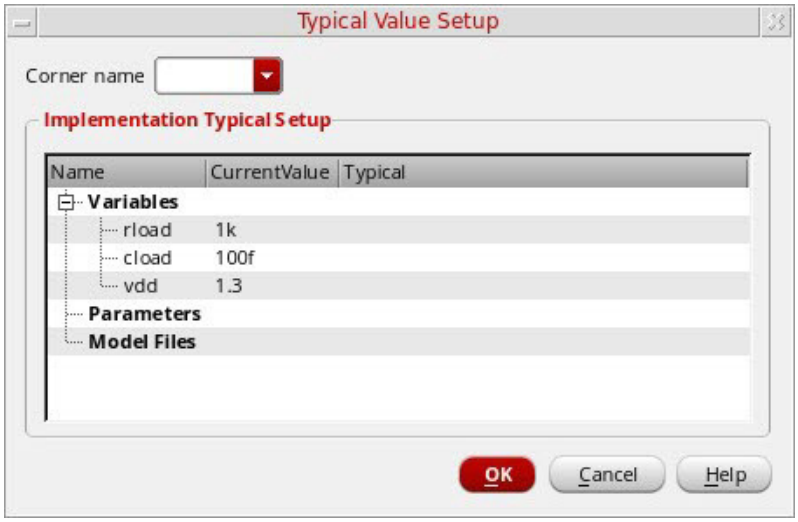
➡ Click *Tools – Typical Value Setup*.

The Typical Value Setup form appears.



© 2023
All Rights Reserved.

The column headers in the CSV file must contain 'Name,Value' where the name is a unique single-line text and the value is a number. You can modify these values in the form.



Verifier automatically removes any white spaces when you do the following:

- Specify a typical value that has white spaces at the beginning, center, or at the end
- Import the typical values from a comma-separated (CSV) file;
- Load the typical values containing white spaces from an existing Verifier cellview from an old version

You can export the typical values with the other results by choosing *File – Export*.

To set up the typical value for all the implementations directly:

1. Click *Tools – Typical Value Setup*.
2. Add the variable or parameter in the form by using one of the following methods.
   a. Right-click and select *Add.*

      Specify a unique name in the Add dialog box and a numeric value in the *CurrentValue* column by double-clicking it.

   b. Right-click in the empty area, variable, or parameter item and select *Import from CSV*.

      Ensure that the column headers of the specified CSV contains the 'Name,Value'. The name must be unique one-liner text without spaces. The value must be blank or a number.

© 2023 All Rights Reserved.

3. Click OK to save the settings.

For details, see Edit Typical Value Setup Form.

To modify or set up the typical value based on the selected implementation cellview:

1. Right-click and select *Edit Typical Value Setup* in the Implementations table.

   By default, all global variables and parameters are loaded and shown on the form.

   Current Value indicates the real value for the variable or parameter in the implementation cellview.

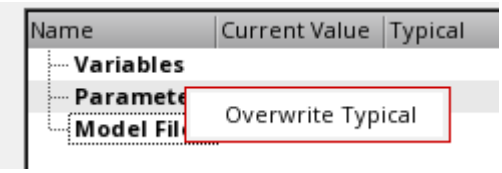
   Typical indicates the defined value for this variable or parameter that is used to look for the required data point.

2. Select a corner name from the corresponding drop-down list box. The variables, parameters, or model files for the specified corner are loaded from the typical and corner setup. The *Current Value* is overwritten by the corner value for the given variable or parameter.

3. Select one of the following:

   a. Double-click the *Typical* column to specify the value.

   b. Right-click and select *Overwrite Typical* to overwrite the model file path and the section.



   You can overwrite the typical values using a model file without section details.

4. Click OK to save the settings.

For details, see Edit Typical Value Setup Form.

To identify the typical value for the implementation output, you need to run implementation from Verifier, or load the implementation simulation results in Verifier. A blank value indicates that there is no data point match with the typical value setup. `various` indicates that there are multiple matched data points. You need to re-run or re-load the implementation when you have modified the typical value setup.

To report the typical value, do one of the following:

© 2023 All Rights Reserved.

- Click *File – Export*.

  The typical value would be exported with requirement properties and other result information.

- Click *Tools – Publish HTML Report*.

  The typical value would be shown in the HTML report.

Related Topics

Typical Value Setup

Edit Typical Value Setup Form

Project Directory Setup

By default, Verifier uses the default project directory to access read-only history data and run simulations in read-only mode. You can create additional project directories for implementations for which you have the required read-access permissions. These additional project directories allow you to access read-only histories owned by other users from the ADE Assembler setup.

While performing read-only operations, such as fetching implementation details or results from an implementation history, Verifier searches for the information in the default project directory as well as other specified directories.

To view or load read-only histories that belong to other users, you must add the project directories of other users to the implementation.

Consider a scenario when user1 creates a `maestro` cellview `impl1` in ADE Assembler and runs read-only simulations. User2 creates a new Verifier cellview and adds `impl1` in it. In this case, Verifier shows all histories for which the user has write permission and hides read-only histories from user1.

User2 then adds the project directory used by user1, for example `/disk/projdirs/user1/simulation/` to `impl1`. In addition to all the writable histories of `impl1` and read-only histories of user2, Verifier also shows all read-only histories from user1.

When user2 runs a simulation on `impl1`, the results are saved into `/disk/projdirs/user2/simulation`. The *Results* tab in Verifier shows the simulation results.

When user1 opens the same Verifier view, the latest history created by user2 is not visible because the default project directory is the same as the additional project directory `/disk/`

© 2023 All Rights Reserved.

`projdirs/user1/simulation`. When user1 adds `/disk/projdirs/user2/` to the list of additional project directories, both user1 and user2 can view all histories in the Verifier cellview.

Related Topics

Opening the Project Directory Setup

Adding and Removing a Project Directory

Modifying and Saving a Project Directory

Histories from the Project Directory Setup

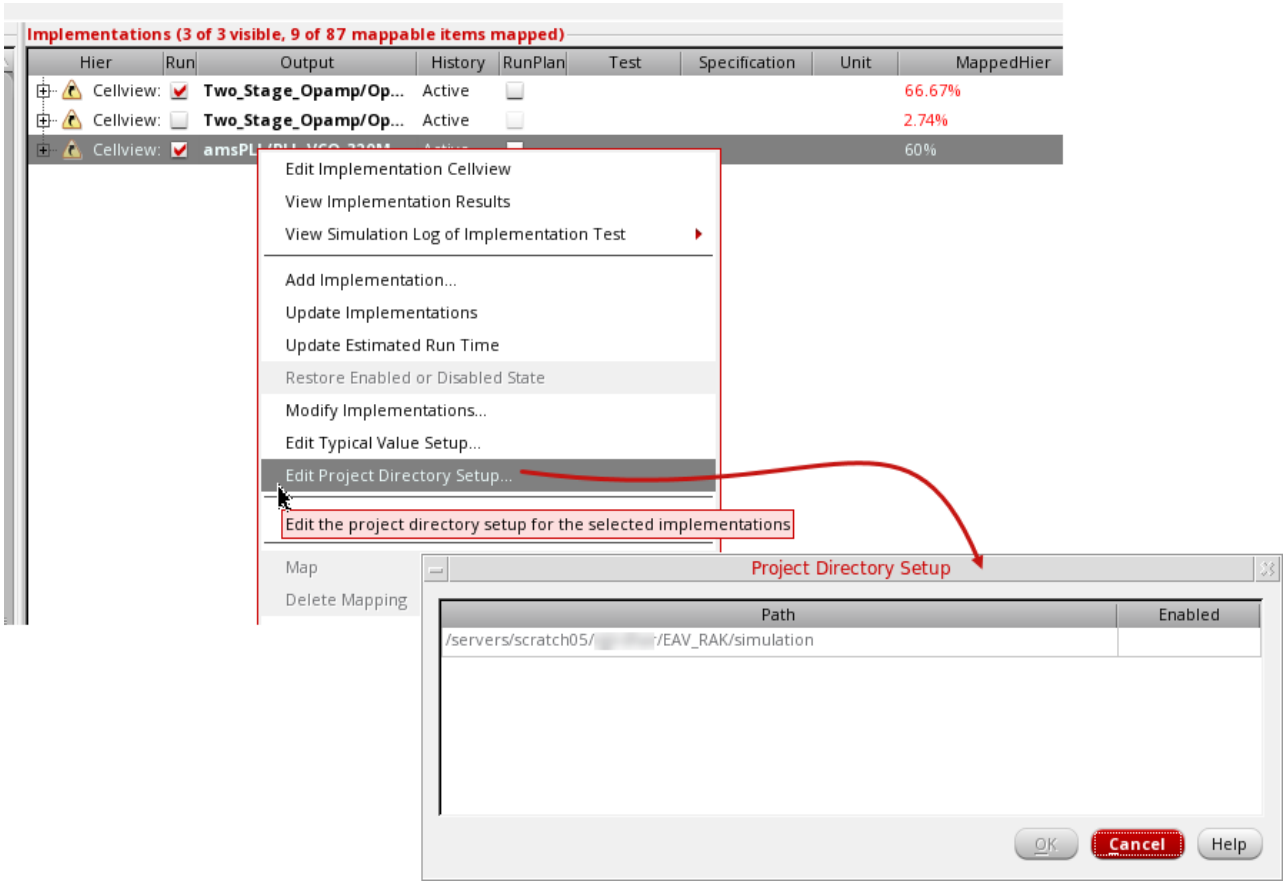
Opening the Project Directory Setup

To access the project directory setup individually for a single implementation:

1. Select an implementation in the *Implementations* pane of the *Setup* tab.
2. Choose the shortcut menu command *Edit Project Directory Setup*.

© 2023 All Rights Reserved.

The *Project Directory Setup* form appears.



The Project Directory Setup form shows the following information:

- *Path*: Indicates the project directory path of the specified implementation.
- *Enabled*: Indicates that the read-only histories from the specified project directory are available in Verifier.

By default, the first row shows the default project directory that you define by using the environment variable `asimenv.startup projectDir`. The default project directory is always enabled. To run simulations, Verifier always uses the default project directory.

The subsequent rows in the project directory setup table show the additional project directories where the project directory information is fetched from the selected implementation. Alternatively, you can use the additionalProjectDirectory environment variable to retrieve this information.

© 2023
All Rights Reserved.

Related Topics

Project Directory Setup

Adding and Removing a Project Directory

Modifying and Saving a Project Directory

Histories from the Project Directory Setup

Adding and Removing a Project Directory

To specify these directories, you can use the absolute or relative path or a path that contains UNIX environment variables $ (for PATH) and ~ (for HOME). Escape characters do not need to be prefixed to special characters. These additional project directories are saved with the implementation data in Verifier.

In case of naming conflicts, the default project directory gets preference. The additional project directories are added in a sequential order.

To add a project directory:

1. Right-click the table in the *Project Directory Setup* form and choose *Add*.

   The Add Project Directory form appears.



2. Specify a path to an existing directory in the *Project directory path* field by using one of these ways:

   - Enter an absolute or relative path.
   - Click *Browse* to select a directory.

3. Click *OK* to add the project directory.

   The *OK* button is enabled only when the specified project directory path is valid.

   The Add Project Directory form closes and the specified project directory is added.

© 2023 All Rights Reserved.

The Project Directory Setup form displays the path of an invalid project directory in red text and in a disabled state. When you move the mouse pointer on the *Path* column, a tooltip shows the reason for the invalid status. It might be useful to retain an invalid project directory path, because it might be valid for a different user's working environment.

To remove a project directory:

1. Select a project directory on the Project Directory Setup form. You can make multiple selections.
2. Right-click and choose *Remove*.

   The selected project directory is removed.

Related Topics

Opening the Project Directory Setup

Project Directory Setup

Modifying and Saving a Project Directory

Histories from the Project Directory Setup

Modifying and Saving a Project Directory

To modify the *Path* to a project directory on the Project Directory Setup form:

➡ Double-click the specified project directory path. The path is available for editing.

To modify the enabled status of the project directory on the Project Directory Setup form:

➡ Click the *Enabled* check box.

If you specify an invalid path, the project directory path is shown in red. If you modify an existing invalid path and make it valid, the text indicating the project directory path reverts to the default color.

To save a project directory:

1. Make the required modifications.
2. Click *OK* to save the project directory setup for the selected implementations.

   Alternatively, click *Save* on the Verifier toolbar.

© 2023 All Rights Reserved.

Related Topics

Opening the Project Directory Setup

Adding and Removing a Project Directory

Project Directory Setup

Histories from the Project Directory Setup

Histories from the Project Directory Setup

When you select an implementation in the *Implementations* pane of the *Setup* tab, the *History* column shows a tooltip for the implementation that displays all additional project directories. This tooltip shows histories using the following order of locations:

- ADE Assembler cellview
- Default project directory in ADE Assembler defined by using the environment variable `asimenv.startup projectDir`
- Additional project directories for the implementation in Verifier using the Project Directory Setup form

For every implementation, Verifier shows the read-only histories from the default project directory and the additional project directories, and the writable histories from the ADE Assembler cellview. The history list shows only the first history if duplicate histories (with identical names) are available. Each history has a tooltip showing the path from which it is loaded. A reference icon against the history name indicates that it is from another project directory.

When you select a history from an additional project directory in Verifier, you can update the implementation with the specified history or load the external simulation results from the specified history. In contrast, ADE Assembler supports reading a history only from the current project directory. Therefore, when you choose the shortcut menu commands *View Implementation Results* and *View Simulation Log of Implementation Test*, a separate instance of ADE Assembler is opened in a child Virtuoso session if the loaded history is read-only and not from the current project directory. To open these child processes, Verifier uses `virtuoso` or any other executable name specified using the Executable or script for batch runs setting in the Preferences form. All such child Virtuoso sessions are closed when the Verifier session is closed.

Note: You cannot run a simulation for an implementation with a history specified from an additional project directory. To run simulations, Verifier always uses the default project

© 2023 All Rights Reserved.

directory that you define by using the environment variable *asimenv.startup projectDir*.

Important

You can open a read-only Verifier cellview in multiple Virtuoso sessions at the same time. However, you can run a simulation in only one of the sessions at a time so that the run summary data of the simulation is not overwritten. To run a simulation in multiple sessions, specify a different project directory in each session.

To know how Verifier stores verification results of cellviews that use project directories, see Location of Run Summary Data.

Related Topics

Opening the Project Directory Setup

Adding and Removing a Project Directory

Modifying and Saving a Project Directory

Project Directory Setup

Custom Implementations

Custom implementations are special implementations that use a structure similar to maestro implementations but use a customized mechanism to retrieve output values. Additionally, with custom implementations, you can perform calculations on output values, statistical values from Monte Carlo simulations, or results of multiple requirements. You can also calculate the sum of currents of multiple design blocks from multiple implementations.

A custom implementation is highly convenient and useful because of the following reasons:

- It uses the same structure as a maestro implementation, which supports all SKILL functions that are available to maestro implementations.
- It contains SKILL expressions for the outputs, which makes the expressions flexible and manageable.
- It can have minimum and maximum specifications and a unit that allows it to be mapped to requirements.
- It can be simulated to evaluate output expressions. This is required to save simulation results and displaying these results in the *Results* tab.

© 2023 All Rights Reserved.

- It can have a Verification Space assigned to it. You can also calculate coverage for a requirement that is mapped to a custom implementation.

Related Topics

Creating a Custom Implementation

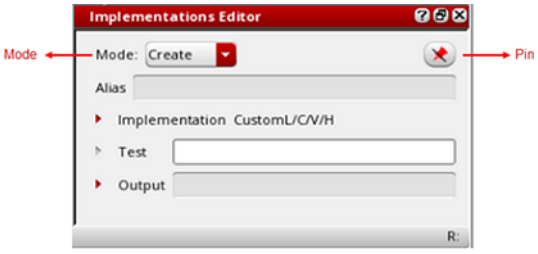
Edit Custom Implementations

Creating a Custom Implementation

To create a custom implementation:

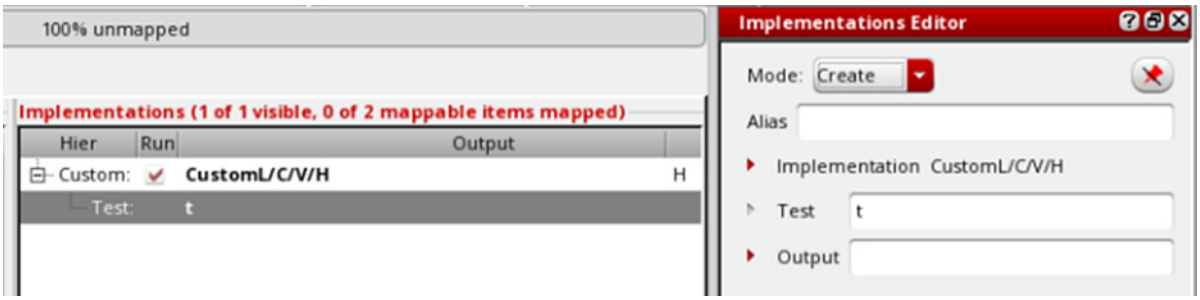
1. Right-click anywhere in the *Implementations* pane and choose *Create Custom Implementation*.

   The Implementations Editor Assistant assistant is displayed in the *Create* mode.



2. In the *Test* field, specify a test name. The test name lets you group multiple output expressions together and define multiple tests in a custom implementation.

   The custom test is automatically created and displayed in the *Implementations* pane.



3. In the *Output* field, specify an output name.
4. In the *Details* field, specify an expression.

© 2023
All Rights Reserved.

The custom output is created and displayed with the specified expression in the *Implementations* pane.

Tip

You can add more implementations by clicking anywhere below the implementations list in the *Implementations* pane, and then selecting the *Create* mode in the Implementations Editor assistant.

A custom implementation uses the same structure as a maestro implementation, such as `library/cell/view/history/test/output`. The test name in this structure groups the various output expressions.

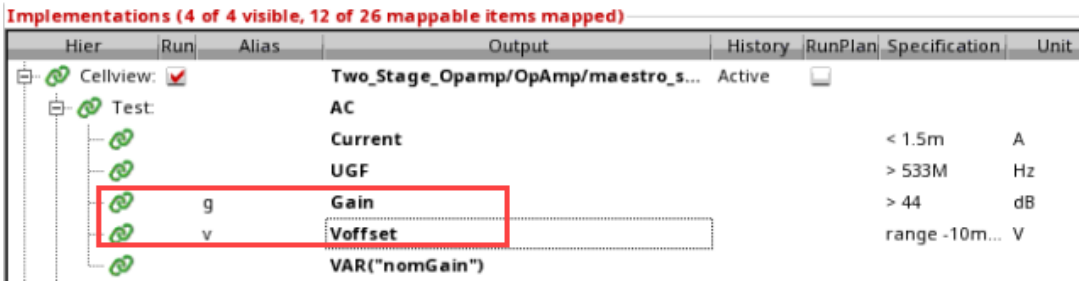
Caution

A custom implementation name is created using the default values `CustomL/C/V/H`***. This does not create an actual implementation. Avoid modifying these default values.***

Example: Creating an implementation for adding two outputs

To know the results after adding two outputs:

1. Specify an alias name to each value that you want to add.

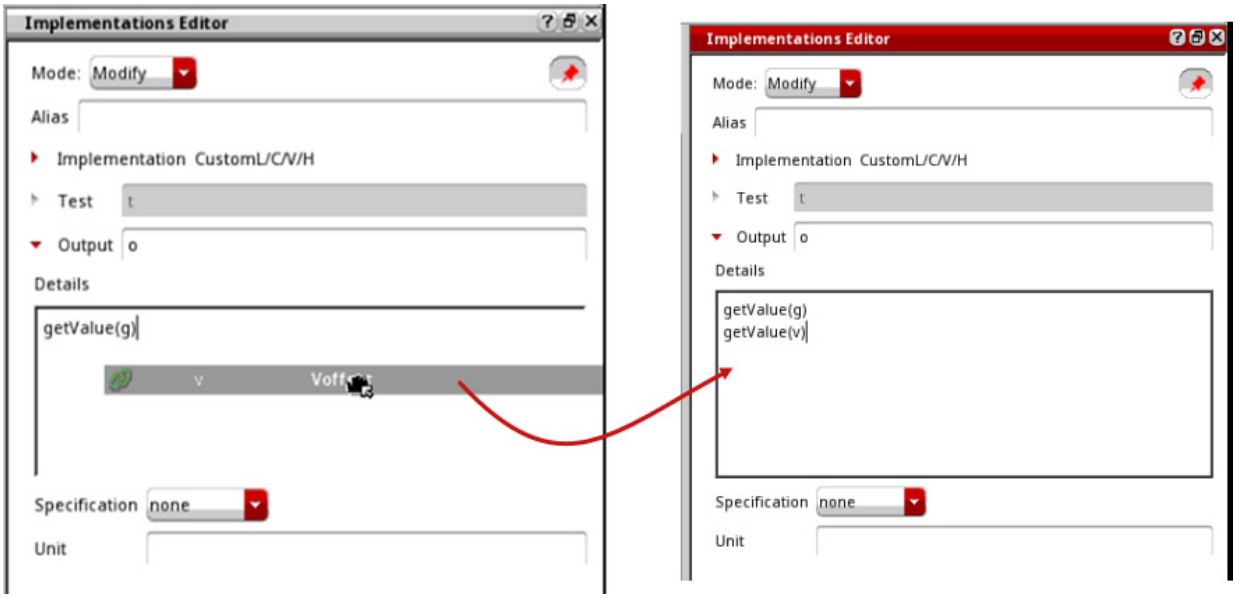


2. Create a custom implementation.
3. Specify the test and output names.
4. Click the *Pin* button to pin the Implementations Editor.
5. Drag and drop the output values to the *Details* field.

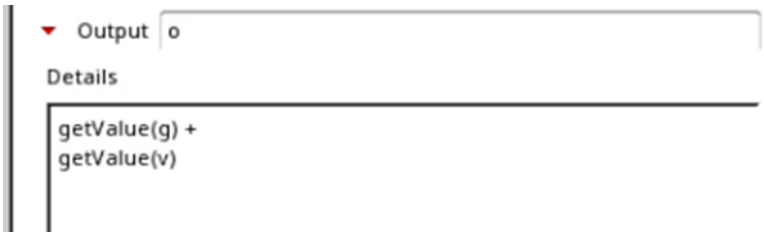
   The output expressions are created automatically.

© 2023
All Rights Reserved.

An output expression is a combination of the output name and a string property which is a mathematical expression and defines the output.



6. Add the '+' operator between the two expressions.



Related Topics

Custom Implementations

Edit Custom Implementations

Edit Custom Implementations

In ADE Verifier, a maestro implementation allows you to view the specifications set in the maestro cellview. On the contrary, if you have a custom implementation in your cellview, you can make changes to the values or edit the output expressions to modify your calculations.

You can edit custom implementations in multiple ways as described in the following topics.

© 2023 All Rights Reserved.

- Editing a Single Custom Implementation
- Editing Multiple Custom Implementations
- Editing Output Expressions in Custom Implementations
- Using Aliases in Custom Implementations

Editing a Single Custom Implementation

To edit a single custom implementation:

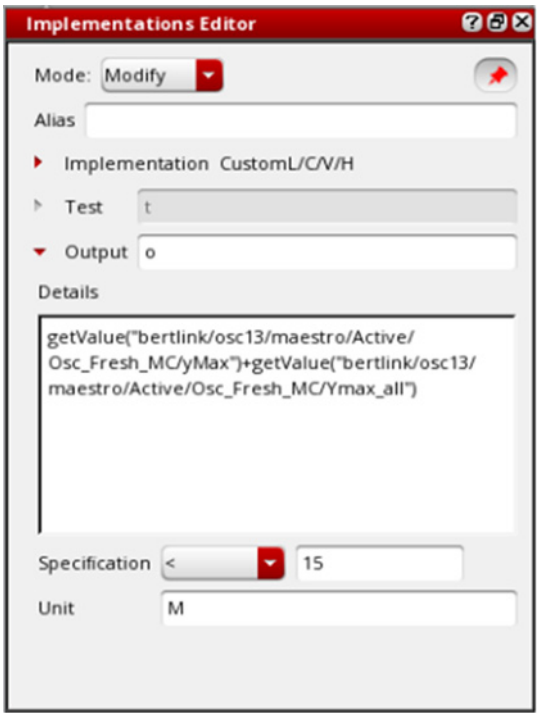
1. In the *Implementations* pane, select the custom implementation.
2. Do one of the following:
   - Double-click the required field to make it editable.



   - Right-click and choose *Modify Custom Implementation*.

© 2023
All Rights Reserved.

The Implementations Editor opens in the *Modify* mode.



3. Specify the modified value.

Related Topics

Edit Custom Implementations

Editing Multiple Custom Implementations

Editing Output Expressions in Custom Implementations

Using Aliases in Custom Implementations

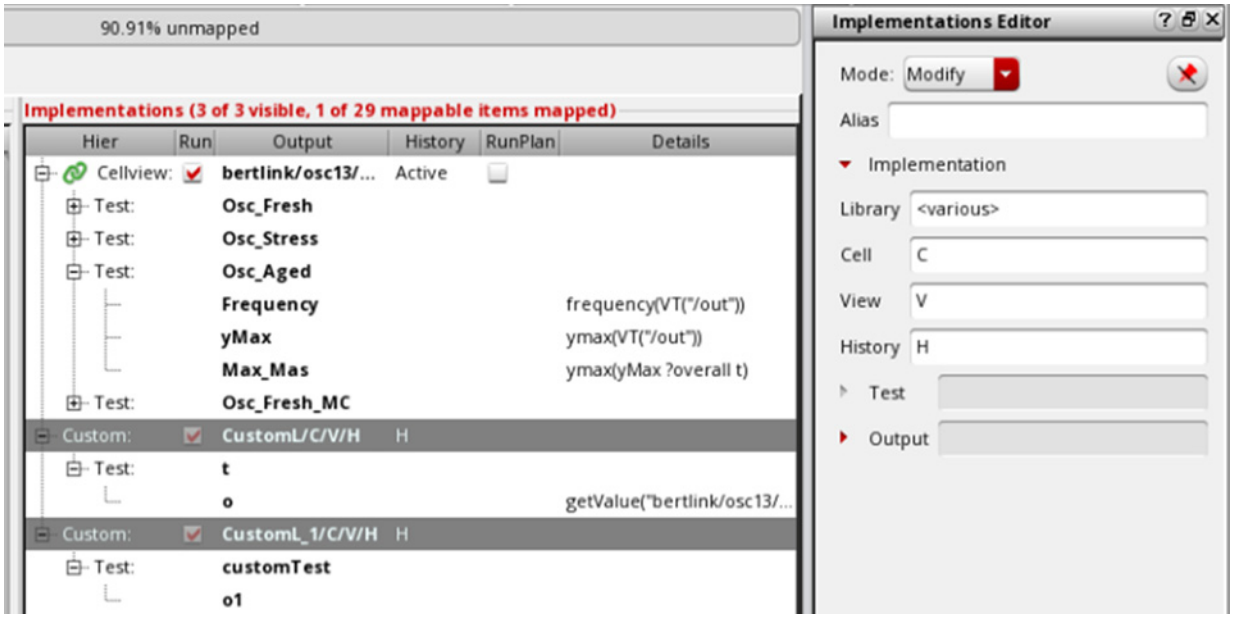
Editing Multiple Custom Implementations

To edit multiple custom implementations:

1. With the `Ctrl` key pressed, select the custom implementations that you want to modify.
2. Right-click and choose *Modify Custom Implementations*.

© 2023 All Rights Reserved.

The Implementations Editor is displayed in the *Modify* mode, and the *Library* field shows *<various>*.



3. Specify the modified value.

Tip

To reevaluate multiple custom implementations with a single click of the *Run* button, add all the corresponding expressions to a new custom implementation or group all the custom implementations into an implementation set.

Related Topics

Edit Custom Implementations

Editing a Single Custom Implementation

Editing Output Expressions in Custom Implementations

Using Aliases in Custom Implementations

© 2023
All Rights Reserved.

Editing Output Expressions in Custom Implementations

To edit output expressions:

1. Select the custom implementations that you want to modify.
2. Right-click and choose *Modify Custom Implementations*.
3. Do one of the following:
   - Use the `getValue` function or specify a SKILL statement block to retrieve the result values.
   - Click the *Pin* button on the top right corner of the Implementations Editor to pin it to the current selection. Then drag and drop any output to the *Details* field to insert `getValue` functions automatically.
     - Pinning ensures that pin the assistant to the current selection, and clicking any other item in the hierarchy will not change the displayed values.
     - To unpin the assistant, click the Pin button again. The Pin button is displayed with a diagonal strike-through in the unpinned state and the original selection is restored.
   - Drag and drop a SKILL script file into this field.
4. Click another field or press the `Ctrl+S` key combination to set the details.

© 2023 All Rights Reserved.

For calculations, the following SKILL functions are available within the Implementations Editor.

Function	Description

`getValue(`*`outputName`*`)`	Returns result values from maestro implementations and is defined only in the local environment of the custom expression. The function fetches the `Min`, `Typical`, or `Max` values according to the `Min`, `Typical`, or `Max` output calculated.
	
	To explicitly get `Min`, `Typical`, or `Max` values, use:
	`getValue(outputName ?key ["min"|"typical"|"max"])`
	
	To retrieve only the typical value of *`outputName`*, use:
	`getValue(outputName ?key "typical")`
	
	To retrieve the `Min`, `Typical`, or `Max` value using the `getKey()` and `getValue()`, use:
	`case(getKey()`
	`("min" getValue(xxx ?key "max"))`
	`("typical" getValue(xxx ?key "typical"))`
	`("max" getValue(xxx ?key "min") + getValue(xxx ?key "max"))`
	`)`

`getSession()`	Returns the current Verifier session number.
`getPath()`	Returns the hierarchical path or parts of the path of the selected custom output.
`getLib()`	Returns the hierarchical library of the selected custom output.
`getCell()`	Returns the hierarchical cell of the selected custom output.
`getView()`	Returns the hierarchical view of the selected custom output.
`getHistory()`	Returns the hierarchical history of the selected custom output.
`getTest()`	Returns the hierarchical test of the selected custom output.
`getOutput()`	Returns the hierarchical output of the selected custom output.
`getKey()`	Returns the Min or Max value being calculated.
`expandAlias(alias)`	Returns the expanded string value of the alias provided.

Important

These functions cannot be used outside the Implementations Editor. This is

© 2023 All Rights Reserved.

because the Implementation Editor provides a special SKILL environment and these functions are only available in this environment.

You can use these functions in custom SKILL functions by creating macros. For example, assume that you create a custom implementation with an expression that specifies the constant values of `4` and `8` for the output values.

```
case(getKey()
         ("min" 4 )
         ("max" 8)
    )
```

To simplify the process, you then decide to define the following function:

```
procedure(minmax(_min _max)
case(getKey()
("min" _min )
("max" _max)
)
)
```

Using this function in the custom implementation will report an error because the `getKey` function has not been defined in the environment of your custom SKILL function '`minmax`'. When you use a macro as shown below, no errors are reported:

```
defmacro( minmax (_min _max)
    `(case (getKey)
                ("min" ,_min)
                ("max" ,_max)
     )
)
```

Related Topics

Edit Custom Implementations

Editing a Single Custom Implementation

Editing Multiple Custom Implementations

Using Aliases in Custom Implementations

Example: Combining Multiple Pass/Fail Results

Example: Combining Min or Max Values in Custom Implementations

© 2023 All Rights Reserved.

Results of Modifying Expressions in Custom Implementations

When you modify the library, cell, view, or history specified for a maestro implementation, the run summary data for the implementation is deleted. Similarly, when you modify the library, cell, view, or history for a custom implementation, the corresponding run summary data is deleted.

Contrarily, the *Run* page displays the simulation status of the custom implementation as *Run Out of Date* when you modify the following in a custom implementation,

- Test: Includes modifying the test or its name.
- Output: Includes modifying the output name, its details, specification, or unit.

Related Topics

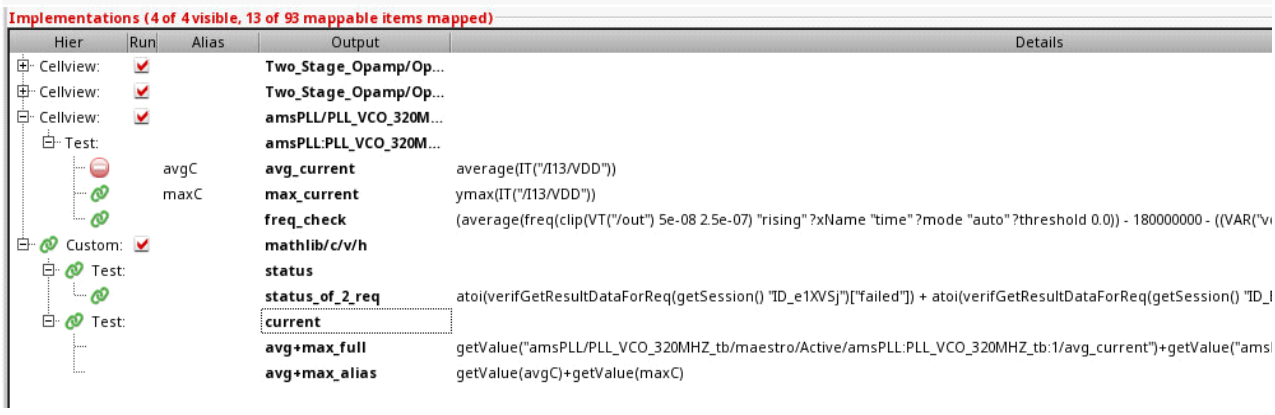
Editing Outputs in Implementations

Editing Output Expressions in Custom Implementations

Simulation Status of Implementations

Using Aliases in Custom Implementations

In ADE Verifier, the hierarchy lists the complete structural path of an implementation to uniquely identify an implementation. This causes the implementation title to become lengthy and difficult to read. By using aliases, you can reference specific values, such as an implementation, test, or output. An alias is a short and readable reference to a specific value and must be unique within the Verifier cellview.



© 2023
All Rights Reserved.

An alias is a SKILL variable defined in the local environment of the expression to reference an output. The variable value is a string corresponding to the referenced output in the following format:

```
getValue(myOutput) = getValue("myLib/myCell/myView/myHistory/myTest/myOutput")
```

To combine alias and strings, use the `strcat` function:

```
getValue(strcat(TestAlias "/out1"))
```

To get comprehensive details in error messages, use the `expandAlias` function:

```
getValue(expandAlias("myOutput"))
```

To refer to other references, use these commands sequentially:

```
getValue(r1->r2->myOutput)
getValue(expandAlias("{r1/r2}myOutput"))
```

To specify an alias:

1. Right-click a custom implementation or a test or an output from a custom implementation, and choose *Modify Custom Implementation*.

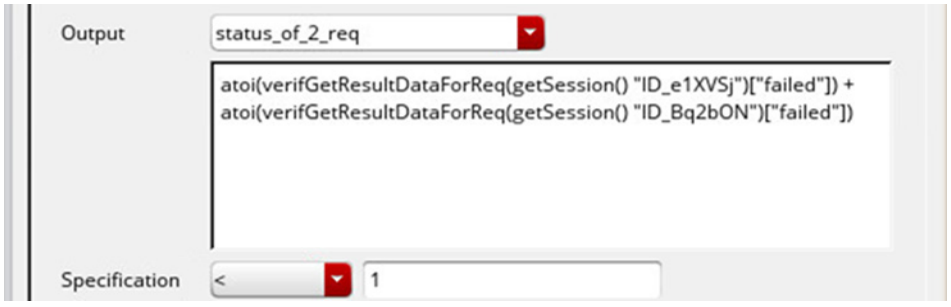
   The Implementations Editor assistant is displayed.

2. In the *Alias* field, specify an alias name. The selected item can be referenced using the specified alias in `getValue` expressions.

   Alternatively, select the *Alias* column to show in the Implementations pane and specify the alias value inline.

Example: Combining Multiple Pass/Fail Results

Assume that you want an output to pass only if multiple requirements have passed. Using the `getSession` function, you can combine the status of failed requirements by adding their failed points. If any point fails, its output fails as well.

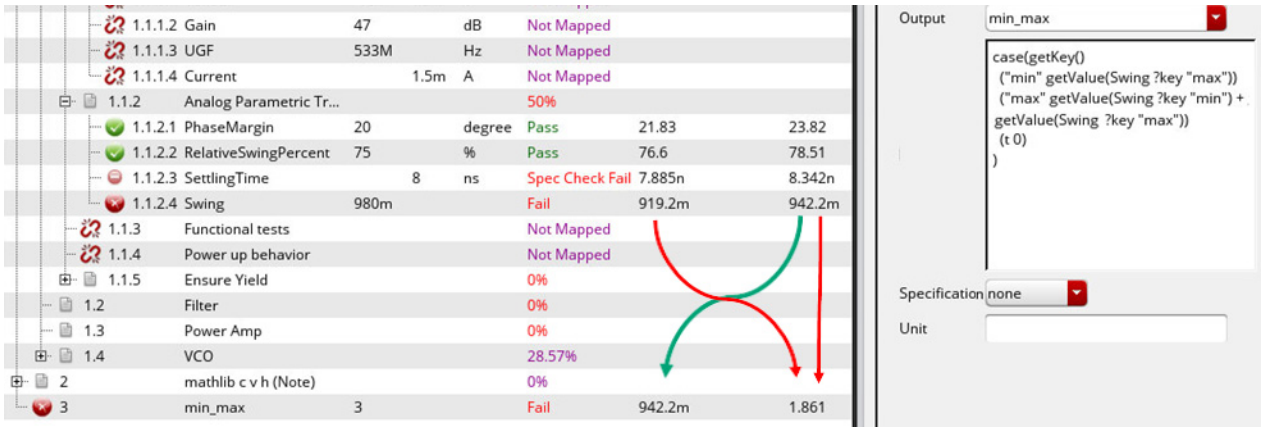


© 2023
All Rights Reserved.

Example: Combining Min or Max Values in Custom Implementations

To combine *Min* and *Max* values in an output expression, use:

```
case(getKey()
  ("min" getValue(xxx ?key "max") )
  ("max" getValue(xxx ?key "min")+ getValue(xxx ?key "max")))
```



Related Topics

Edit Custom Implementations

Editing a Single Custom Implementation

Editing Multiple Custom Implementations

Using Aliases in Custom Implementations

© 2023
All Rights Reserved.

© 2023 All Rights Reserved.

5

Implementation Runs in ADE Verifier

After you have set up your design verification project with mapped requirements and implementations, you capture the simulation results of the implementation cellviews. Verifier uses these results to determine the verification status of the entire project, and for each requirement in the project. To capture the results, you can run the implementation cellviews from Verifier or load the existing run results in Verifier. You can also review the results.

You can set various aspects of the implementation runs that you initiate from Verifier. The main graphical user interface lets you enable or disable runs, whereas the run preferences let you control the run settings. You can also manage implementation runs that include ADE Assembler run plans.

You can organize implementation cellviews into implementation sets. An implementation cellview can belong to multiple sets. You can then run all the cellviews in a set.

Verifier lets you generate batch scripts to run implementation cellviews of your project from the command line interface. You can schedule these scripts to run as required. For example, you can set a batch script to run all the cellviews of an implementation set, say `run_weekly`. You can create a cron job to run this script once a week, or as required.

Related Topics

Implementation Simulation Flow

Setting Pre-run Scripts

Implementation Sets

Simulation of Implementations

Simulation of Implementations Externally

Environment Setup to Run Simulations in ADE Verifier

Simulation Run Results

© 2023 All Rights Reserved.

Batch Scripts in ADE Verifier

Scheduling Runs

© 2023 All Rights Reserved.

Implementation Simulation Flow

Verifier lets you simulate implementations to capture results, or load existing simulation results. It uses the results to determine the verification status of the requirements mapped to those implementations. The overall verification project status depends on the verification status of all the requirements.

The following is a sample flow for simulating implementations for verifying your design:

1. Specify the implementations that you want to run from Verifier, and the implementations whose results you want to load in Verifier. For details, see Implementation Runs in ADE Verifier.
2. Organize the implementations that you want to run from Verifier in implementation sets. For details, see Implementation Sets.
3. Run the implementations from Verifier, or run the implementations from outside Verifier and load the results in Verifier. For details, see Simulation of Implementations.

   Verifier lets you generate batch scripts, which you can use to schedule the runs. For details, see Batch Scripts in ADE Verifier and Scheduling Runs.
4. Review the results from Verifier, or directly in the main application of the implementations. For details, see Simulation Run Results.

   If required, you can modify the design and rerun the implementations.

When Verifier has the simulation results, you can review the verification reports. For details, see Managing Verification Results

Setting Pre-run Scripts

You can use a pre-run script to update a set of implementations before running the simulation. For example, you can enable or disable tests or corners, use setup states, change variables and parameters, and so on. Pre-run script is the script file that is executed before simulation.

To add a pre-run script, do one of the following:

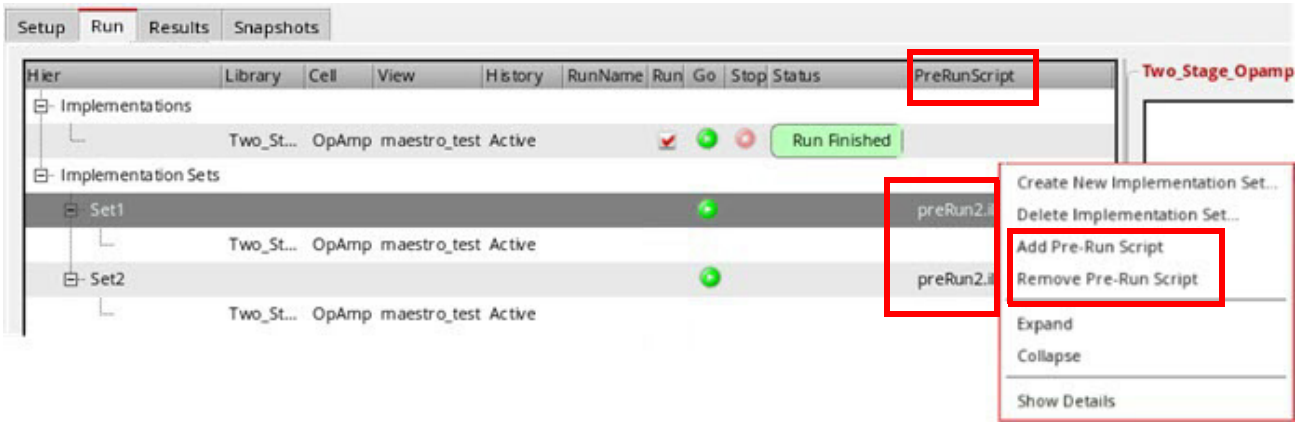
- Right-click an implementation set in the *Run* tab and select *Add Pre-Run Script*.
- Double-click the *PreRunScript* column of an implementation set and specify the script name.

Additionally, you can specify a relative or absolute path to the script file, or modify the specified path. You can also shell environment variables in the file path. For example, you can

© 2023 All Rights Reserved.

specify `${MYSCRIPTS}/prerun.il`. To retrieve the name of a pre-run script file, you can use the SKILL function `verifGetImpSetPreRunScript`. To specify a new pre-run script file, you can use the SKILL function `verifSetImpSetPreRunScript`.

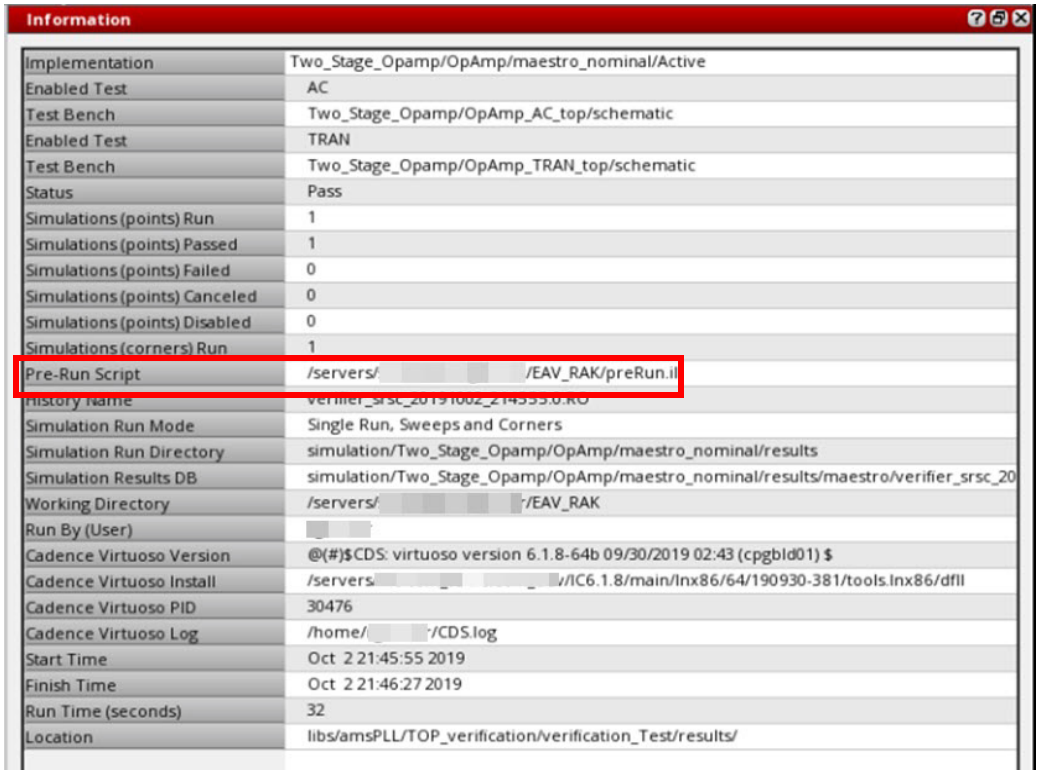
You can use pre-run scripts within implementation sets only. Pre-run scripts are not used if you run a simulation for an implementation, outside of an implementation set. However, when you run the simulation for an implementation set, the pre-run script impacts all the implementations in the set except for the implementations that are in the "Load" mode.



During the simulation the pre-run script in use is shown in the *PreRunScript* column of the *Run* tab for the individual implementations. Once the simulation is complete, the pre-run script information for the implementations is removed from the *PreRunScript* column.

© 2023
All Rights Reserved.

You can observe the *Pre-Run Script* field populated in the *Information* assistant when simulation is complete. The *Information* assistant does not display the *Pre-Run Script* row if the pre-run script is not run because of errors.



Examples

The following example shows the script demonstrating changes in variables and tests.

```
printf("** PreRun Script Example **\n")

printf("** change global variable values **\n")
maeSetVar("vdd" "1.3")

printf("** change global temperature **\n")
maeSetVar("temperature" "25")

printf("** enable/disable tests **\n")
maeSetSetup(?tests '("AC") ?enabled nil)
maeSetSetup(?tests '("TRAN") ?enabled t)

printf("** Enable/Disable some corners **\n")
maeSetSetup(?corners '("myCorner1_0") ?enabled nil)

printf("** Load Setup State **\n")
maeLoadSetupState("test" ?tags list("vars") ?operation `overwrite)

printf("** change some corner values **\n")
enCorners = maeGetSetup(?typeName "corners" ?enabled t)
foreach(cornerName enCorners
```


© 2023 All Rights Reserved.

```
    when( (cornerName != "nominal")
       maeSetVar("vdd" 1.4 ?typeName "corner" ?typeValue list(cornerName))
    )
)

printf("** Change model files **\n")
foreach(testName maeGetSetup(?typeName "tests" ?enabled t)
   maeSetEnvOption(testName ?options '(("modelFiles" (("gpdk045.scs" "tt")))))
   maeSetEnvOption(testName ?options '(("includePath" "./gpdk045_v_3_5/models/
spectre")))
)
```

To know about the `mae*` functions, see Virtuoso ADE SKILL Reference.

Setting Up Run Preferences

To set the run options:

1. Choose *Edit – Preferences*.

   The Preferences form displays.

2. Click the *Run Options* tab.

3. Edit the options as required.

   You can edit the following types of preference options organized in group boxes.

Group Box	Description
Implementations	Set options to show and run mapped implementations, and run implementations in debug mode.
Report Identical History	Set options to do pre-run checks on implementations and results.
Check the following items	Set options for tests, test states, views, model and definition files.
Run Controls	Set the maximum number of jobs that Verifier can run in parallel, and the default job policy that Verifier must use when running implementations.

4. Click *OK*.

Verifier uses your run settings when it runs implementations.

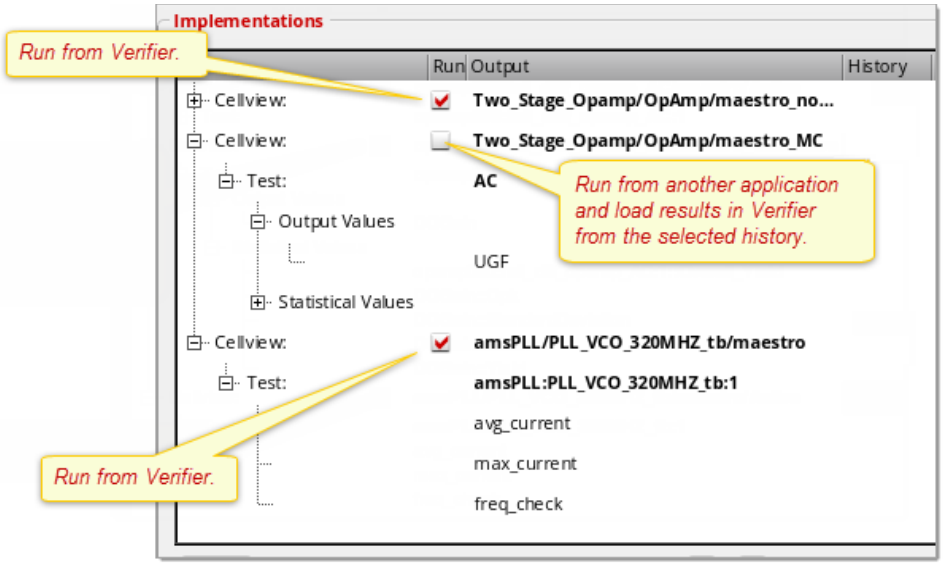
© 2023 All Rights Reserved.

Related Topics

Run

Enabling or Disabling Implementation Runs from Verifier

You can run an implementation cellview from Verifier to capture the results used for verification. Alternatively, you can run the implementation cellview from an external application and load its results in Verifier. In this case, you must select the cellview history where the results are stored and load them in Verifier.



To ensure that you can run an implementation cellview from Verifier:

1. Click the *Setup* tab or the *Run* tab to view the implementations.
2. Ensure that the *Run* check box is selected for the implementation cellview.

   If required, display the *Run* column. For this, right-click anywhere on the header area in the *Implementations* pane and select *Run*.

The option to run the implementation cellview becomes available in the *Run* tab.

To ensure that you can only load the run results of an implementation cellview:

1. Click the *Setup* tab or the *Run* tab to view the implementations.
2. Ensure that the *Run* check box is not selected for the implementation cellview.

© 2023
All Rights Reserved.

3. Select the history from the *History* drop-down list of the implementation cellview where the results are available.

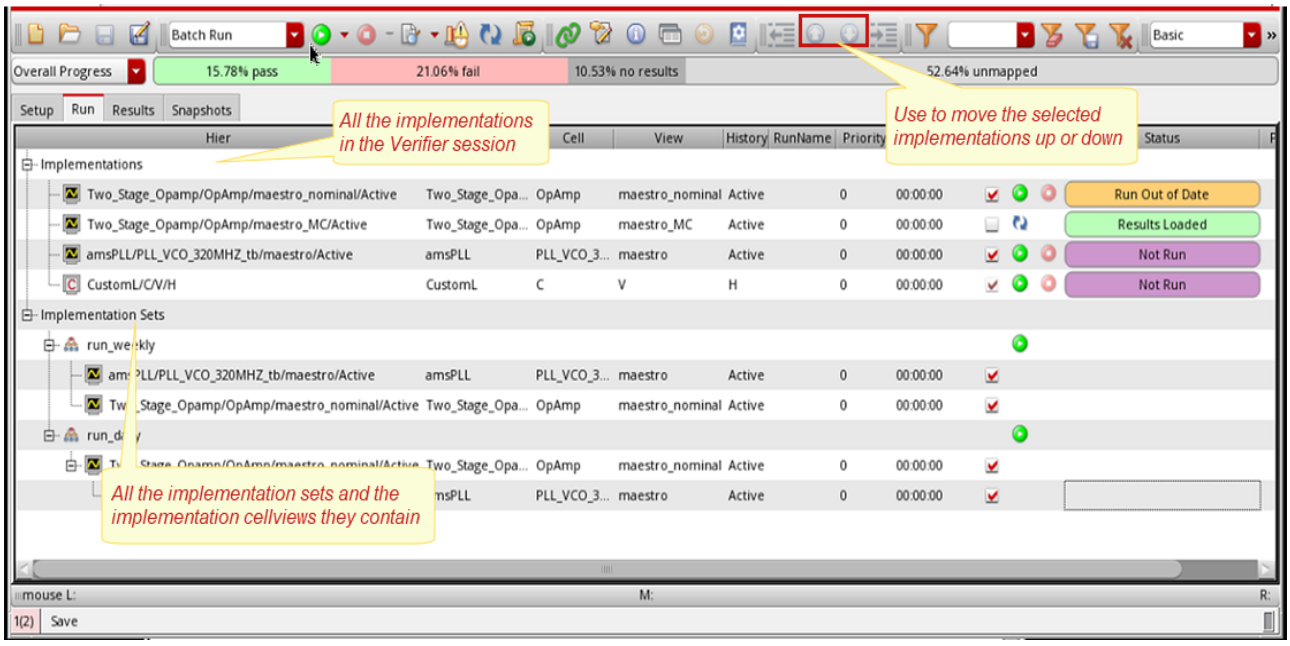
   Verifier displays the tests and outputs of the selected history. If the *History* column is not displayed, click *Show* and select *History*.

The option to load the results becomes available in the *Run* tab.

Implementation Sets

You organize and run the implementation cellviews of your verification project using the features available in the *Run* tab.

As shown in the following figure, all the implementation cellviews appear under the *Implementations* node. You can optionally create implementation sets to group the implementation cellviews according to your needs. For example, you can group them according to their run frequency. You can create a set called `run_daily` to group the implementation cellviews that must be run every day. Similarly, you can create `run_weekly` to group the implementation cellviews that must be run once a week. You can then choose to run all the implementation cellviews that belong to an implementation set. An implementation cellview can be included in multiple sets.



In this figure:

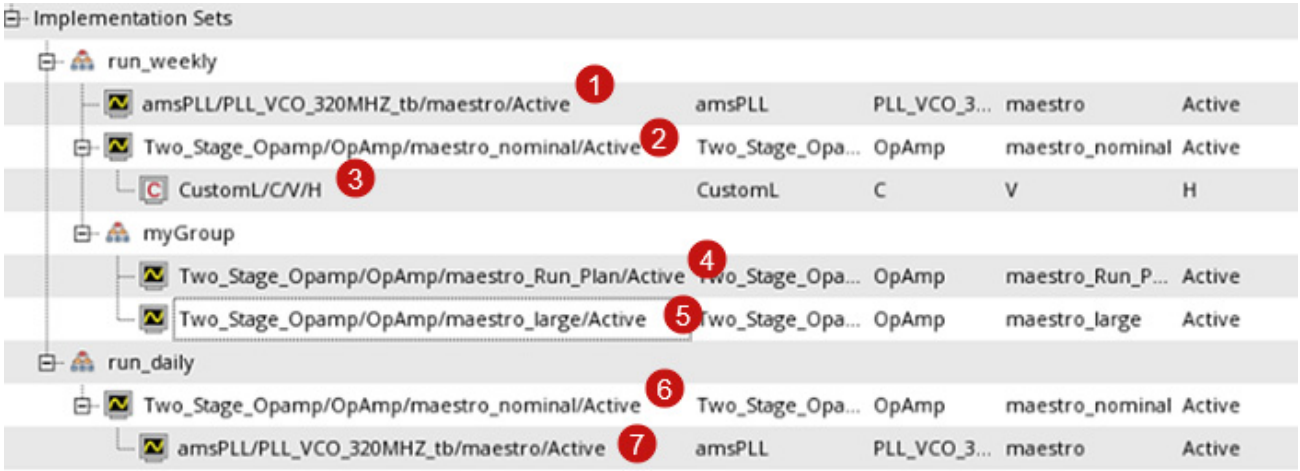
-  indicates a run group or an implementation set

© 2023
All Rights Reserved.

- indicates a custom implementation
- indicates an implementation

The implementation sets follow a hierarchical methodology that helps you to define dependencies for runs. All implementations at a given level of hierarchy are considered equivalent.

For example, consider the following figure:



This figure shows two top-level implementation sets, *run_weekly* and *run_daily*, which can be run independently because there is no dependency between them.

The simplest case of a dependency is visible in *run_daily*, where implementation 7 will only run after implementation 6 has successfully completed.

The set *run_weekly* contains two implementations, implementation 1 and implementation 2, and a child set *myGroup*. These two implementations have to successfully finish before the contents of *myGroup* , implementation 4 and implementation 5, are even considered for running. Specifically, for any child implementation set, simulation of any sibling implementations of its parent set have to finish before the implementations of the child implementation set can be released to run.

Additionally, within *run_weekly*, implementation 3 is a dependent of implementation 2. This means that implementation 3 will be scheduled to run only when the simulation of implementation 2 finishes. The preference Maximum implementations to run in parallel determines the number of implementations that can run at any given time.

When a parent implementation or set fails or is stopped, all its dependencies are stopped. For example, in the given illustration, if implementation 6 fails, then implementation 7 is stopped.

© 2023
All Rights Reserved.

Similarly, if implementation 2 fails or is stopped, then implementations 3, 4, and 5 are stopped.

The *Priority* value also determines the order of simulation runs. However, Verifier honors this value at the level where the implementation exists in the hierarchy. For example, consider that implementation 2 has a higher priority than implementation 1. If you run the implementation set *run_weekly*, then implementation 2 runs first. Similarly, when you run *myGroup*, implementations 4 and 5 will run in the order of their specified *Priority* value. Again, if implementation 4 has the highest priority of all the implementations, it will run only after the simulation of its parent has finished.

The key points to remember about an implementation set hierarchy are as follows:

- An implementation that is a child of an implementation set is also considered to be at the top level in the hierarchy, and is not a dependent.
- An implementation that is a child of another implementation is considered as a dependency of the parent implementation.
- An implementation set that is a child of an implementation set is considered a dependency of the parent set.



Tip

Organizing and grouping implementations or implementation sets logically will avoid any confusion during simulation runs.

To run implementation sets in Batch Run mode and honor the *Maximum implementations to run in parallel* and *Maximum batch runs to start in parallel* settings, enable the distributeImpSet environment variable. To ensure that implementations within implementation sets are distributed according to these settings in batch mode, the implementation must not have any hierarchy or dependencies. In case of hierarchical implementation sets, all implementations of a hierarchical branch must be included in the same batch run, which makes it impossible to distribute these implementations between different runs.

Consider the following points when creating implementation sets:

- An implementation within an implementation set or a group must be unique.
- An implementation set name must be unique in the hierarchy.
- The *Run* button is available only at the top level in an implementation set.
- You can modify the *Run*, *Priority*, and *EstRunTime* values for the implementation anywhere on the *Run* tab.
- You can drag an implementation and drop it into a group or implementation set.

© 2023 All Rights Reserved.

- You cannot add a local and referenced implementation into the same top-level group if they have the same library, cell, view, and history.

Creating and Deleting Implementation Sets

To create an implementation set:

1. Right-click the implementations area and choose *Create New Implementation Set*.

   If you want to create a sub set in an existing implementation set, right-click the existing set and choose *Create New Implementation Set*.

2. Type the name of the implementation set in the displayed form.
3. Click *OK*.

The new set is added to the *Implementation Sets* node, which is added below the *Implementations* node, if it is not already present..

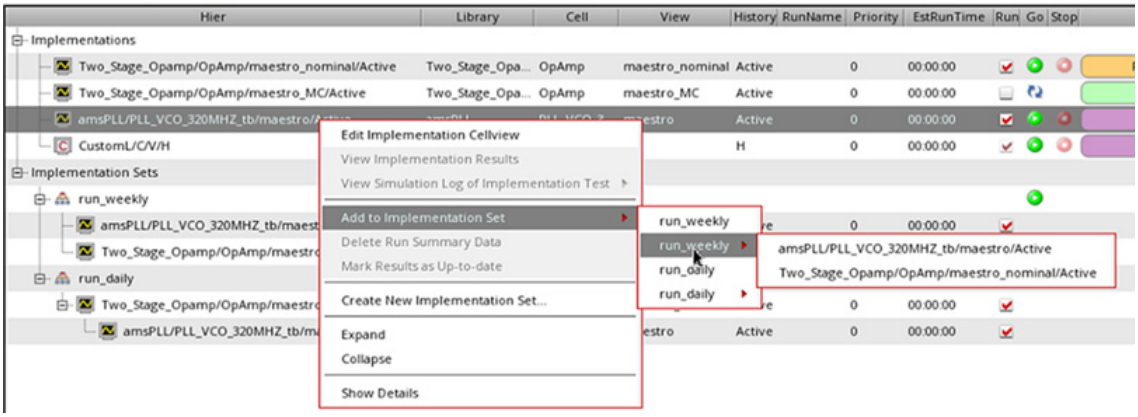
To edit an implementation set name:

➡ Double-click and edit the name directly in the *Implementation Sets* node.

To add an implementation cellview to an implementation set:

1. Right-click the implementation cellview in the *Implementations* node and choose *Add to Implementation Set*. You can select multiple implementation cellviews.

   A hierarchical list of existing implementation sets is displayed. For every set, a pair of sub-menu items are displayed, where the first item allows adding the implementation to the set on the first level. The second item in this sub-menu allows adding the implementation as a dependency of existing implementations in the set.

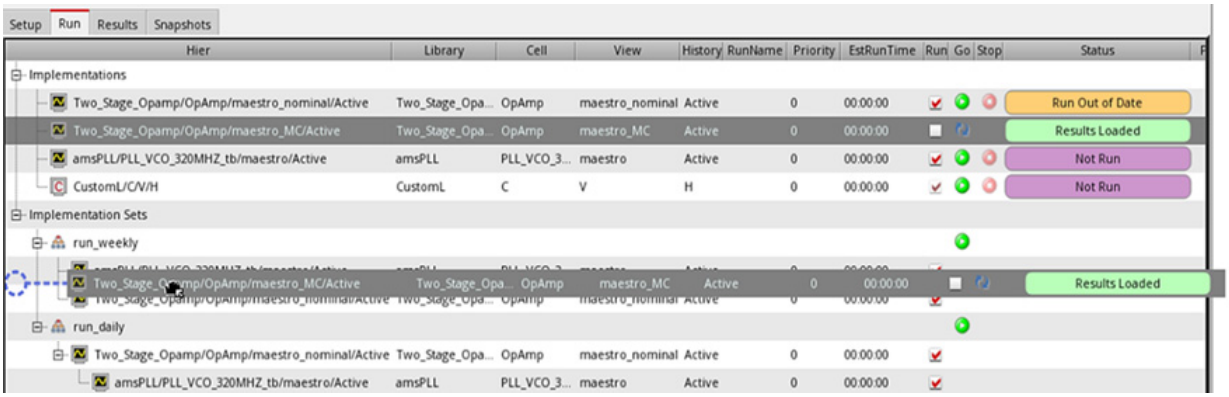


© 2023
All Rights Reserved.

2. Choose the implementation set.

   You can add an implementation to multiple implementation sets.

   Alternatively, you can drag an implementation from its source location and drop it to the required destination.



3. Click *OK*.

Verifier adds the selected implementation cellview to the specified implementation set.

To remove an implementation set, or an implementation cellview from an implementation set:

1. Right-click the implementation cellview in the *Implementation Sets* node and choose *Delete Implementation Set*.

   Verifier displays a prompt to confirm the removal.

2. Click *OK*.

The implementation set or cellview entry is removed from the *Implementation Sets* node.

Rearranging Implementation Sets

To rearrange an implementation set or cellview:

1. Select the implementation set or cellview from the *Implementation Sets* node.
2. Click the *Up*, *Down*, *Left*, or *Right* arrow buttons on the toolbar to change the sequential order.

When you choose to run the cellviews of an implementation set, the order in which they are run depends on their order in the *Implementations* node. Rearranging the cellviews under the *Implementations Sets* node does not reflect on the *Setup* page.

© 2023
All Rights Reserved.

Simulation of Implementations

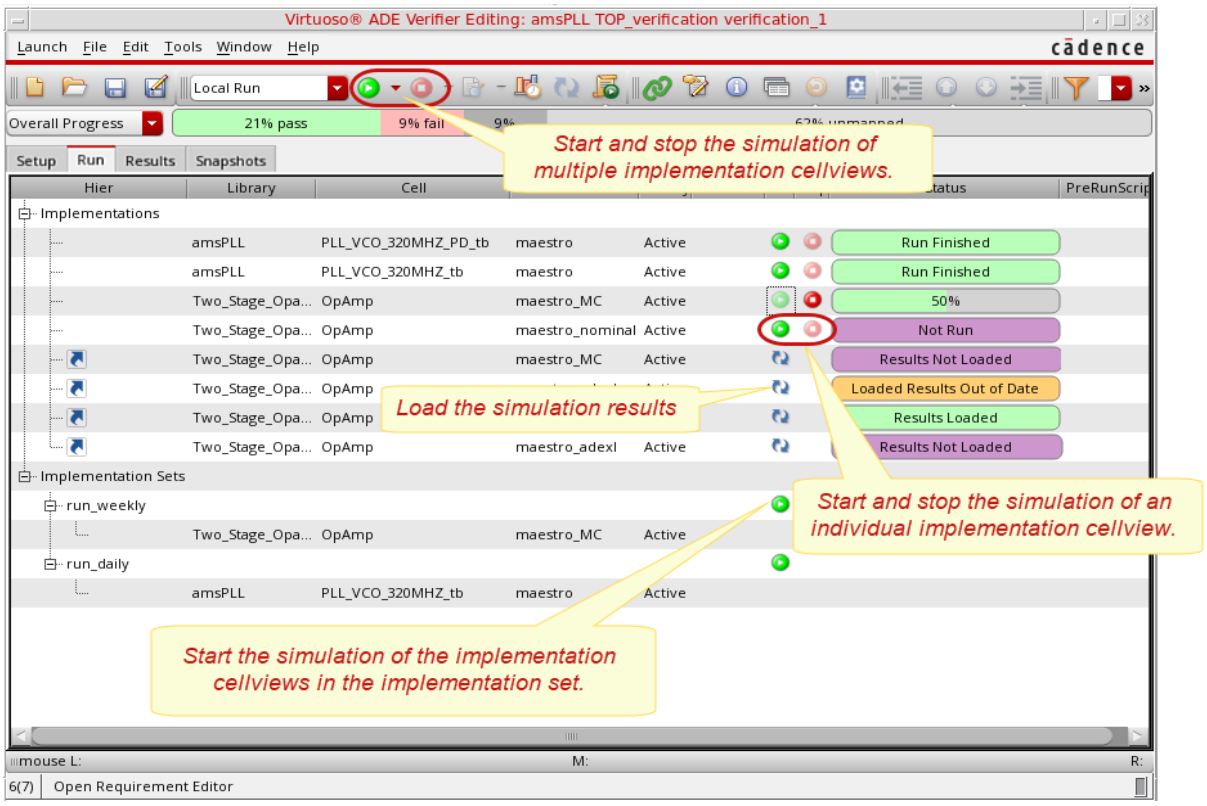
Verifier lets you simulate implementation cellviews individually, serially, or in parallel. You can use implementation sets to organize the implementation cellviews and their runs. You can configure how you want to run implementations.

It is possible to run implementations from another application and load the run results in Verifier. For details, see Implementation Runs in ADE Verifier and Implementation Sets.

For information on using a batch script to simulate implementations, see Batch Scripts in ADE Verifier.

If you want to make changes to your implementation cellview source when its simulation is running, ensure that the option to ignore the design changes made during the runs is enabled. For this, set the environment variable `ignoreDesignChangesDuringRun` to `t` in the `.cdsenv` or `.cdsinit` file.

You can view the current simulation status of implementation cellviews from their *Status* indicators. For details, see Simulation Status of Implementations. It is possible to stop a simulation that is in progress. The following figure shows the run controls of Verifier.



© 2023
All Rights Reserved.

You can show or hide the columns of the implementations list in the *Run* tab. For this, right-click anywhere on the column heading in the implementations area and select the columns you want to show. Verifier hides the columns that are not selected.

To sort columns, you can use the drag-and-drop method and change the display of columns. Additionally, you can specify the default column order and hidden columns by using the environment variables runColumnOrder and runHiddenColumns.

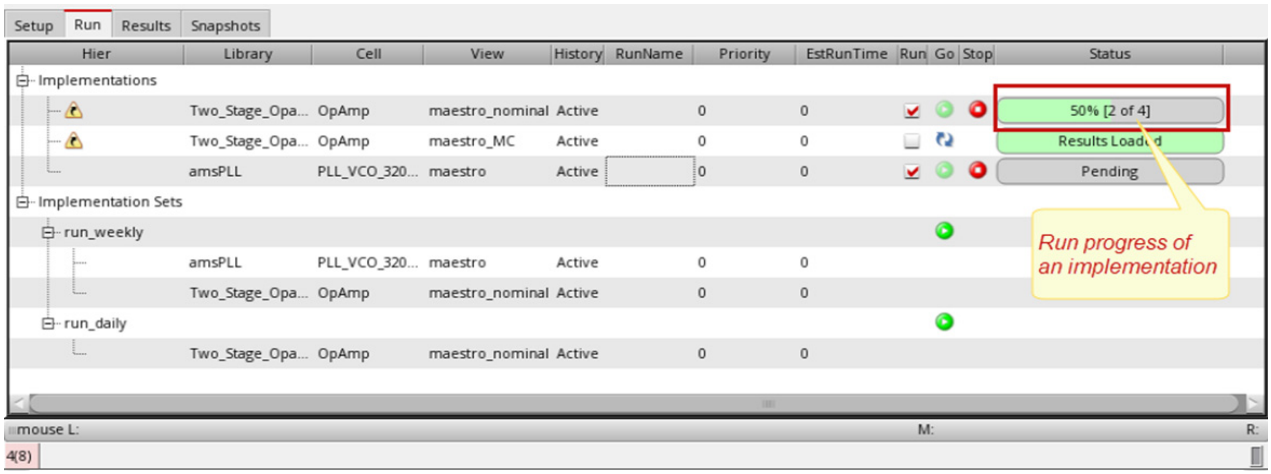
Additional Information

- When you run an implementation cellview, only the enabled tests of that cellview are run. The disabled tests are not run.
- If the current `verifier` cellview is opened in read-only mode and Verifier is set to store the run summary data in a directory, you can run the implementation cellviews. If Verifier is set to store the run summary data in the current cellview, you cannot run the implementation cellviews because Verifier cannot write to the read-only cellview. These settings are available in the Run tab of Change Referenced Cellview Form.
- You can choose the appropriate job policy that must be applied to all implementation runs by setting the following preferences or their corresponding environment variables:

Preferences	Corresponding Environment Variable
Override job control mode	impJobControlMode
Override simulation job policy	impjobpolicy
Override netlisting job policy	impNetlistingJobPolicy

© 2023 All Rights Reserved.

You can also view the status of simulations that are currently running in the *Run* tab. Move the mouse pointer to the *Status* column of a simulation to see the number of simulation points submitted, completed, or failed.



Running and Stopping Simulations

To run all the implementations organized in an implementation set:

➡ Click *Run* on the toolbar.

To run an individual implementation cellview:

➡ Click *Run* corresponding to that cellview.

To run multiple implementation cellviews:

1. Do one of the following:

   - Ensure that you have not selected any cellview or implementation set if you want to run simulations for all the implementation cellviews.
   - Select the relevant implementation sets or implementation cellviews if you want to run simulations for only those cellviews.

   Implementation cellviews can run serially or in parallel, depending on the value set in the *Maximum implementations to run in parallel* preference option.

2. Click *Run* on the toolbar.

Verifier runs the implementation cellviews and lets you monitor their status.

© 2023
All Rights Reserved.

If the implementation cellview uses the integrated history management, you receive a popup asking you to convert the cellview to use the separate history management scheme.

For information on running simulations on referenced cellviews, see Simulation of Implementations Externally.

To stop a run that was started from Verifier, do one of the following:

- Click *Stop*  on the toolbar.
- Click *Stop* corresponding to the implementation cellview you want to stop.

To view the requirements associated with an implementation in the *Run* tab, right-click and select *Show Details*.

 *Tip*

If an implementation cellview does not run properly, try to run it outside of Verifier. Also, check the run preferences, license requirements, and other settings in Verifier. For example, if the simulator path is not set properly, the implementation cellviews cannot run. You can attempt to run the cellview after enabling the preference option *Debug mode, open implementation interface*.

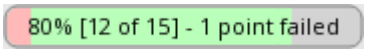
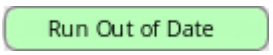
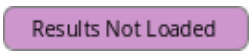
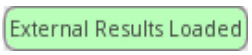
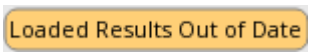
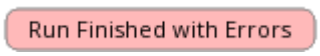
Simulation Status of Implementations

You can monitor the run status of your verification project using the *Status* column on the *Run* tab. This column displays each implementation cellview or custom implementation with its corresponding status indicators.

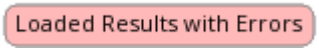
The following table describes these indicators.

Status Indicators	Description
2 Not Run, 4 Pass, 1 Pending	The summary of the simulation status is displayed.
Not Run	The cellview is not simulated.
Pending	The cellview has been selected for simulation and is in queue.
Launching...	The cellview simulation is no longer pending but waiting for the simulation run to start.
Running (50%)	The cellview is being simulated, and the indicated percentage of simulation has been completed.

© 2023 All Rights Reserved.

Status Indicators	Description
	The cellview is being simulated. The indicated percentage of simulation has been completed and the indicated failed points have been detected.
	The cellview simulation has completed successfully, and the results are available for verification.
	(`LSCS only`) The simulation is running, and the status shows the progress of the simulation and netlist generation.
	The cellview simulation is stopping.
	The cellview simulation was stopped manually.
	The pending cellview simulation was stopped manually.
	The cellview simulation was started, but failed because of an error. When the error is encountered, a popup message appears. You can also view the details in the Virtuoso CIW.
	The cellview was simulated previously. However, the available simulation results can be out of date because the cellview has changed since it was last simulated. The color of this indicator changes from green to yellow to indicate potential problems. Consider re-simulating the cellview to get the latest results.
	The cellview simulation results are not loaded in Verifier and Verifier cannot initiate its simulation. You must load the results from a history. The red background color of this indicator indicates issues in loading the results. For example, the color becomes red when the simulation results are not found in the selected history.
	The cellview simulation results are loaded from a history.
	The cellview simulation results were loaded previously. However, the loaded results can be out of date because the cellview has changed since it was last simulated. The color of this indicator changes from green to yellow to indicate potential problems. Consider reloading the latest results in Verifier after running the cellview externally.
	The external simulation results were partially available and were loaded from an incomplete history.
	The simulation run finished with errors.

© 2023
All Rights Reserved.

Status Indicators	Description
	The external simulation results were loaded but contained errors.
	The simulation is running. The indicated percentage of simulation and netlist generation has been completed and the indicated failed points have been detected.

Report Identical Histories

By default, before running a simulation, the setup details are not compared with the loaded history. The design of many verification blocks might not have changed and this can result in running a number of redundant simulations with the same setup. To avoid this, choose Edit - Preferences and enable the check for Report identical histories before run. This check does the following:

- For implementations with 'Active' history, compares the active setup with the setup from the loaded history, and runs simulation only if the setup is unique. If any identical history is found, it is reported.
- For implementations with a named or specified history such as 'Interactive.0', compares the setup of the specified history and the loaded history, and runs simulation only if the setups are different. If any identical history is found, it is reported.

Simulation of Implementations Externally

In the normal run mode, the current Virtuoso session is used to launch ADE Assembler in the background and run simulations. This works well for short simulations. However, for simulations that run for longer durations, you might prefer to launch an independent Virtuoso session. The *Batch Run* feature typically launches an independent Virtuoso session in `nograph` mode that allows running your simulations outside of the current Verifier session. It launches without `nograph` mode if you select the *Debug mode, open implementation interface* preference. The *Batch Run* includes the following process:

1. Creates a small Virtuoso batch script.
2. Starts the batch script in the background. This batch script internally launches Virtuoso, which creates a new Verifier session. The new Verifier session runs the simulations.

© 2023 All Rights Reserved.

3. Triggers automatic loading of results when the batch run is finished. In addition, the status information is continuously sent to the current Verifier session to monitor the progress.

The main advantage of using the *Batch Run* is that the implementation simulation run and the Virtuoso GUI session are two independent Virtuoso processes. It minimizes the impact on the performance of Verifier. The Verifier or Virtuoso process can be closed without the risk of impacting the simulation run. The flow is particularly useful for long simulation runs. Splitting the simulation into multiple independent processes minimizes the risk of unexpected failures impacting the overall results. You can start a batch run by choosing *Batch Run* from the Run modes combo box on the toolbar and clicking the run button  on the toolbar.

Every time you start a batch run, it automatically creates a script in the following format:

`verifRun_<library>_<cell>_<view>_<id>.script`

The corresponding log file is saved in the following format:

`verifRun_<library>_<cell>_<view>_<id>.log.`

Here, `<library>`, `<cell>` and `<view>` indicate the Verifier cellview which initiates the batch run, and `<id>` is an integer value starting from 0, which is reset at the end of the run.

Forexample, if the current Verifier cell name is `TOP_verification`, the library in use is `Two_Stage_OpAmp`, the view is `verification`, and the id is `21`, the script file is named as `verifRun_Two_Stage_OpAmp_TOP_verification_verification_21.script`.



It is strongly recommended that you do not modify the script file because this file is created automatically and uses internal functions.

Running and Stopping Implementations in Batch Mode

To run an individual implementation cellview in batch mode, do the following

1. Select an implementation.
2. Select *Batch Run* in the Run modes combo box on the toolbar.



© 2023
All Rights Reserved.

3. Click the *Run* button () on the toolbar.

To run all the implementations organized in an implementation set in batch mode:

➡ Click the *Run* button () corresponding to that implementation set.

Verifier runs the implementation cellviews. You can monitor their status.

To stop a run that was started from Verifier, do one of the following

➡ Click the *Stop* button () on the toolbar, at the bottom-left side of the screen, or corresponding to the implementation cellview.

➡ Click the *Stop* button () corresponding to the cellview you want to stop.

➡ Right-click the cellview you want to stop running and choose *Stop Simulation*. Unlike the internal simulation, batch mode simulation can be started concurrently. If one simulation is already running in batch mode, you can start simulation for other implementations to be run in batch mode.

➡ Set the stopBatchRunOnServerClose environment variable to `t`. Setting this environment variable closes all batch run simulations if the Virtuoso session that launched the batch run is closed.

Setting Up Pre and Post Simulation Operations in Batch Scripts

You can add pre- and post-simulation operations in your batch script file to automate operations for your batch runs. To do this, you must set the following SKILL variables in the CIW or the `.cdsinit` file before generating the batch script.

- `verifGetPreRunText`: Specifies the text to be inserted before the simulation run command line. The text can be a string, symbol, lambda function, or a function.
- `verifGetPostRunText`: Specifies the text to be inserted after the simulation run command line. The text can be a string, symbol, lambda function, or a function.

Note: If the text is a symbol or function, it must have the three arguments, *`n_session`*, *`n_runId`*, and *`t_mode`*, and return a string, which is inserted into the script without any changes.

Specifying a string

1. In the CIW or the `.cdsinit` file, specify the following:

© 2023 All Rights Reserved.

```
    verifGetPreRunText = "; Before Run"
    verifGetPostRunText = "; After Run".
```

2. On the Verifier toolbar, click *Run*.

The specified text is added to the script file, as shown below:

```
;; Automatically generated - do not edit

let((log_dir)
   log_dir = "/home/user/adexl/designs/verifier/V3_Examples/logs_user/verifier/
verifRun_opamp090_full_diff_opamp_AC_verification1_0"
   info("Log Dir = %L\n" log_dir)
   ;; Setting up a unique jobFileDir so as to not conflict with other virtuoso
processes.
   envSetVal( "adexl.distribute" "jobFileDir" 'string log_dir)
)

; Before Run

imps = '(
         "opamp090/full_diff_opamp_AC/maestro/Active"
        )

_verifRunSimulation("opamp090" "full_diff_opamp_AC" "verification1"
?parentSession 1 ?parentRunId 9 ?parentHostname "vleu-lnx-45-2" ?parentPID 8147
?implementation imps ?runMode "Local Run" ?readOnly nil)

; After Run

foreach(s verifGetAllSessions()
   verifCloseSession(s ?saveIfModified nil)
)

exit()
```

Specifying a lambda function

1. In the CIW or the `.cdsinit` file, specify the following:

```
verifGetPreRunText = "lambda((sess id mode) sprintf(nil "info(\"User %s running
cellview %s id=%L mode=%s\\n\")" getLogin()
buildString(verifGetSessionCellView(sess) "/") id mode))"

verifGetPostRunText = lambda((sess id mode) sprintf(nil "; %L %L %L\n" sess id
mode))
```

2. On the Verifier toolbar, click *Run*.

© 2023 All Rights Reserved.

The specified text is added to the script file, as shown below:

```
;; Automatically generated - do not edit

let((log_dir)
   log_dir = "/home/user1/adexl/designs/verifier/V3_Examples/logs_user1/
verifier/verifRun_opamp090_full_diff_opamp_AC_verification1_0"
   info("Log Dir = %L\n" log_dir)
   ;; Setting up a unique jobFileDir so as to not conflict with other virtuoso
processes.
   envSetVal( "adexl.distribute" "jobFileDir" 'string log_dir)
)

info("User user1 Running cellview opamp090/full_diff_opamp_AC/verification1 id=23
mode=Batch Run\n")

imps = '(
         "opamp090/full_diff_opamp_AC_bad/maestro/Active"
        )

_verifRunSimulation("opamp090" "full_diff_opamp_AC" "verification1"
?parentSession 1 ?parentRunId 23 ?parentHostname "vleu-lnx-45-2" ?parentPID 8147
?implementation imps ?runMode "Local Run" ?readOnly nil)

; 1 23 "Batch Run"
foreach(s verifGetAllSessions()
    verifCloseSession(s ?saveIfModified nil)
)

exit()
```


© 2023 All Rights Reserved.

Distributing Batch Simulation Runs

You can start multiple implementations and run simulations in parallel in the batch mode. Running simulations in batch mode launches an independent Virtuoso session in -`nograph` mode on the same host machine. These multiple runs imply that multiple ADE Assembler sessions are launched in the current session. To avoid overloading the current Virtuoso session, you can use load balancing commands to distribute the batch runs to different hosts. When using distributed procession of simulation runs, Assembler launches multiple Virtuoso ICRPs (IC Remote Process). The batch runs are distributed to these Virtuoso ICRPs on different hosts on a server farm.

Note: An ICRP (IC Remote Process) is essentially a `virtuoso -nograph` process launched by Assembler when you select *Batch Run* and click the *Run* button in Verifier. This process runs a simulation, evaluates the results, and send the results back to Assembler. The job policy setup in Assembler manages the distribution of simulations.

Verifier lets you specify settings to launch the `nograph` session used by the batch run. This `nograph` session runs the implementations in multiple ADE Assembler processes. These ADE Assembler processes can use a job policy to further distribute the ICRPs.

If you have enabled the *Optimize Single Point Run* option in your Job Policy Setup form in ADE Assembler for a single point run, ADE Assembler optimizes the run by doing the following:

- It completes the netlist generation and expression evaluation tasks inside the ADE Assembler process
- It runs the simulation as per the distribution method specified by the job policy.

Therefore, the ADE Verifier process is blocked by these tasks and the Message Processing System messages, such as restart or stop simulations, in a batch run are possibly lost because of the blocking. The Optimize Single Point Run is automatically disabled by setting the environment variable `"adexl.distribute useSameProcess"` to `nil` during simulation runs started from Verifier.

To specify the batch run settings, use the preference option Executable or script for batch runs and do one of the following:

- Specify an external program to use when running Verifier with *Batch Run*. By default, the external program is '`virtuoso`'.
- Use `bsub` to submit the launched Virtuoso processes to the job queue on a farm.

```
bsub -K -P IC:6.1.8:RD:test -R select[OSREL==EE60] -R
rusage[mem=3000] -W 200:0 -sp 50 -o /tmp/bsub.job virtuoso
```


© 2023 All Rights Reserved.

Note: The `-K` option is mandatory to wait for the simulations to finish before the command returns.

- Specify a script that provides additional functionality.

```
#!/bin/csh -f
echo "starting bsub virtuoso $*"
bsub -K -P "IC:6.1.8:RD:test" -R "select[OSREL==EE60]" -R
"rusage[mem=3000]" -W 200:0 -sp 50 -o /tmp/bsub_job_$$ virtuoso $*
set res=$status
echo "done bsub virtuoso $*"
echo "status = $res"
echo "bsub output in /tmp/bsub_job_$$]"
exit $res
```

Note: The `-K` option is mandatory to wait for the simulations to finish before the command returns. In the above script, the -K option of bsub blocks the command from returning before the simulation finishes.

- Specify a variant of the above script in your `.cdsinit` file.

```
envSetVal("verifier.preferences" "binaryName" 'string "bsub -K -
P IC:6.1.8:RD:test -R select[OSREL==EE60] -R rusage[mem=3000] -W
200:0 -sp 50 -o bsub.log virtuoso")
```

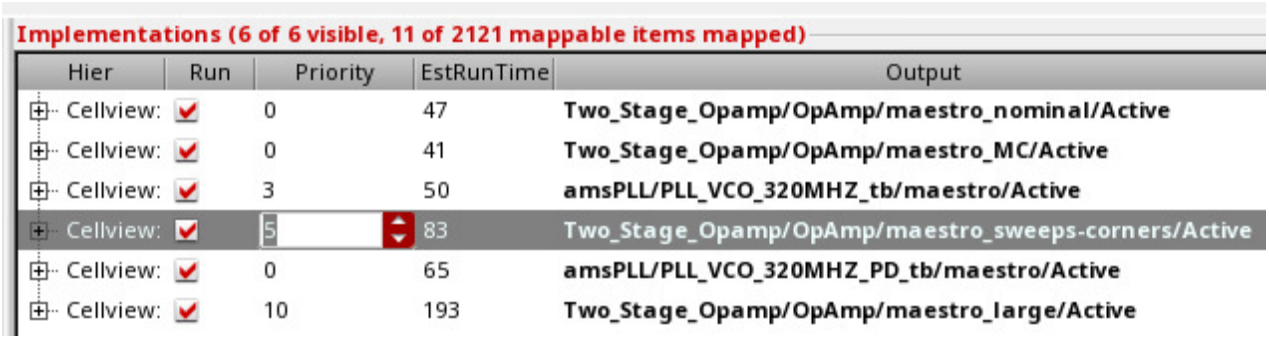
Alternatively, you can use all the above options with the environment variable binaryName.

Simulation Runs based on Priority and Estimated Run Time

You can specify *Priority*, *EstRunTime*, *Maximum implementations to run in parallel*, and *Maximum batch runs to launch in parallel* for an implementation in the *Setup* tab and schedule the simulation of runs based on the specified values. The *Priority* and

© 2023 All Rights Reserved.

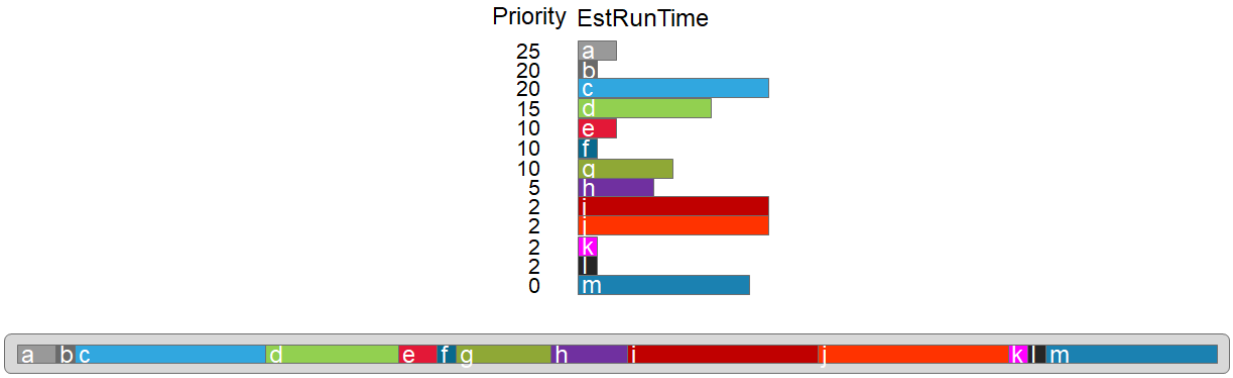
EstRunTime columns are also available on the *Run* tab and are synchronized with the columns on the *Setup* tab.



These specified values influence factors such as the scheduling order and the queue for an implementation.

- The specified priority defines the sequence of running simulations. The higher the numeric value specified, the higher the priority. For implementations with the same priority, the order in the UI decides the priority. The *EstRunTime* is only used when running simulations in batch mode, and it influences the distribution of the runs to batches.

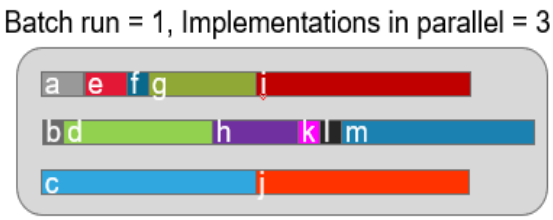
  In this example, without any parallel runs, the order of execution would be:

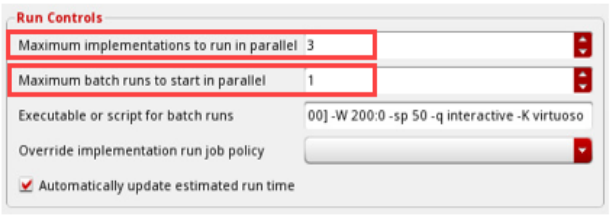


- Each batch run gets one implementation initially based on the specified priority order. Each implementation goes into the batch run with the least estimated run time, until all implementations are assigned to a batch run.

© 2023
All Rights Reserved.

For example, when you set *Maximum implementations to run in parallel* to a number > 1, such as 3, the order of execution is as follows:

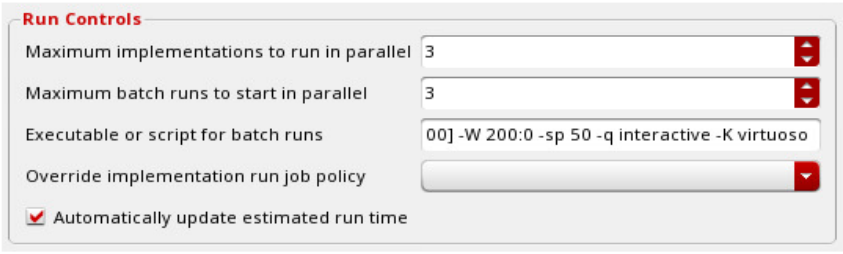




When you set both *Maximum implementations to run in parallel* and *Maximum batch runs to run in parallel* to a number > 1, such as 3, the order of execution is as follows:



Batch runs = 3, Implementations in parallel = 3



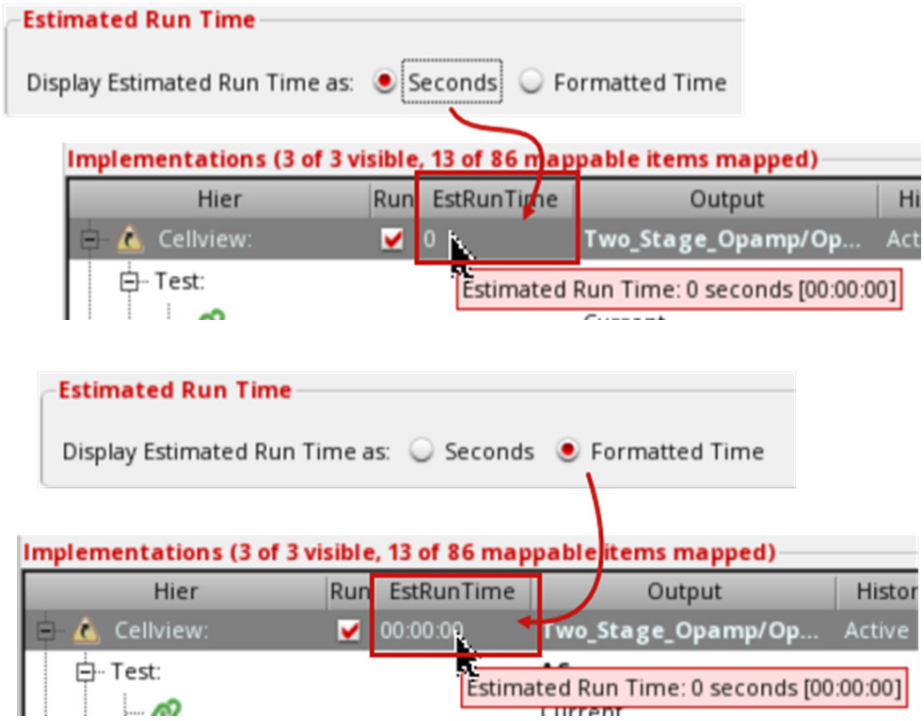
Here, the runs are assigned to the job queue with the least estimated run time.

Important

When the Automatically update estimated run time check box is selected and the Run check box is deselected, EstRunTime is not changed for implementations. The *Automatically update estimated run time* setting enables auto update of the *EstRunTime* numbers after simulation runs are triggered from Verifier.

© 2023 All Rights Reserved.

You can also set the Display estimated run time as: option to *Seconds* or *Formatted Time*.



In the *Setup* or *Run* tab, you can specify the estimated run time as follows:

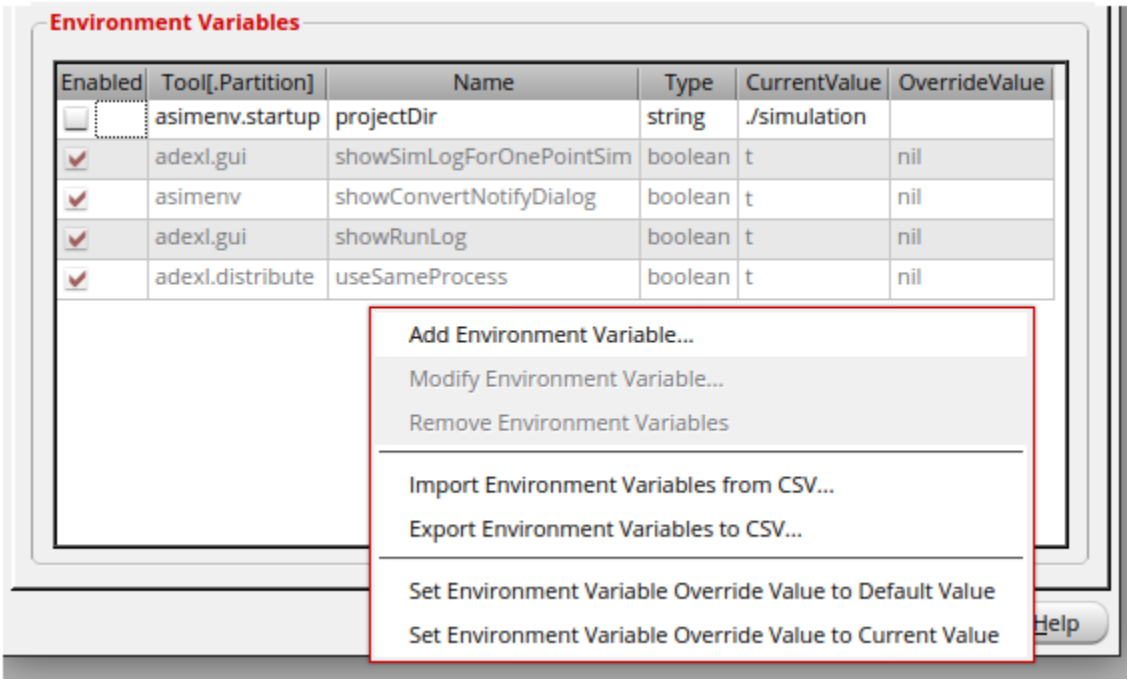
- Number of seconds. For example, 42s.
- Number followed by single character unit for hours, minutes or seconds. For example, 42s, 33M, 3D.
- Number of days (optional) followed by H:M:S. Here, hours(H) and Minutes(M) are optional. For example, '1D 3:42:59', '2D 12:21', '3:56'.

Alternatively, you can specify the estimated run time format using the estRunTimeDisplayFormat environment variable.

© 2023
All Rights Reserved.

Environment Setup to Run Simulations in ADE Verifier

You can access the environment setup for running simulations from Verifier by using the Environment Variables table in the *Run* tab of the Preferences form.



In this table, the following environment variables that appear in a read-only state are always enabled by default because the tool requires these to configure the simulation environment. These environment variables cannot be modified and are always displayed at the bottom of the table.

- `"adexl.gui" "showSimLogForOnePointSim"`
- `"asimenv" "showConvertNotifyDialog"`
- `"adexl.gui" "showRunLog"`
- `"adexl.distribute" "useSameProcess"`

This section describes how you can modify the Verifier environment to run simulations. It covers the following topics:

- Creating a CSV File for Defining Environment Variables
- Adding and Removing Environment Variables
- Modifying Environment Variables

© 2023 All Rights Reserved.

- Creating a CSV File for Defining Environment Variables
- Exporting Environment Variables to a CSV File
- Setting the Environment Variable Override Value

The *Environment Variables* table contains the following columns:

Column	Description
Enabled	If selected, the *OverrideValue* overwrites the *CurrentValue* of the environment variable before running a simulation from ADE Verifier.
Tool[.Partition]	The name of the tool and its optional partition. **Note:** A partition groups the related set of variables associated with a tool.
Name	The name of the environment variable.
Type	The type of the environment variable. Possible values: string, int, boolean, cyclic, float, toggle.
CurrentValue	The currently used value of the environment variable set through the `.cdsenv` or `.cdsinit` file or through the `envSetVal` SKILL function.
OverrideValue	The value of the environment variable used for simulations in ADE Verifier. This value overwrites the *CurrentValue* before running the simulation and restores the *CurrentValue* after the simulation is complete.

The table also offers the following shortcut menu commands:

Command	Description
Add Environment Variable	Adds a new environment variable in the Verifier session. See Adding and Removing Environment Variables.
Modify Environment Variable	Modifies the values for Name, Tool[.Partition], or the OverrideValue of the selected environment variable with values that are not in use. See Modifying Environment Variables.
Remove Environment Variables	Removes the selected environment variables from the list. See Adding and Removing Environment Variables

© 2023 All Rights Reserved.

Command	Description
Import Environment Variables from CSV	Imports environment variables from a CSV file. See Creating a CSV File for Defining Environment Variables.
Export Environment Variables to CSV	Exports environment variables to a CSV file. See Exporting Environment Variables to a CSV File.
Set Environment Variable Override Value to Default Value	Resets the value of the selected environment variable to its default value. See Setting the Environment Variable Override Value.
Set Environment Variable Override Value to Current Value	Resets the value of the selected environment variable to its current value. See Setting the Environment Variable Override Value.

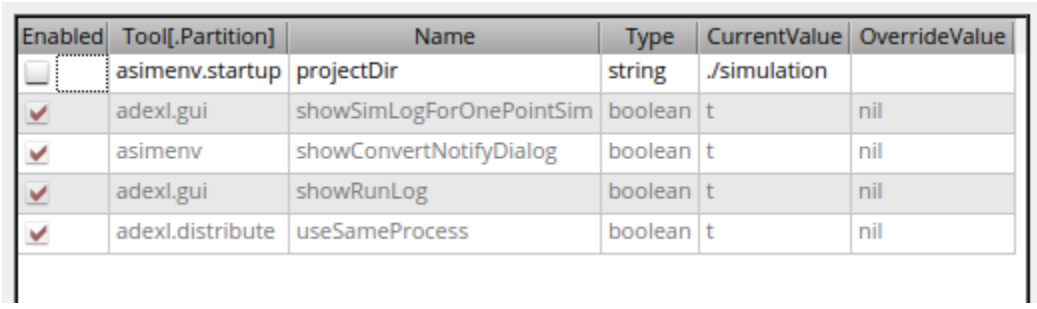
Creating a CSV File for Defining Environment Variables

To create a CSV file that defines environment variables in the current Verifier session:

1. Set the envVarConfigFile environment variable.
2. Create a new Verifier cellview.
3. Choose *Edit – Preferences – Run*.

The table loads and displays the environment variables that you have defined in the CSV file.

If a CSV file is not specified, the table displays the following default rows:



Enabled	Tool[.Partition]	Name	Type	CurrentValue	OverrideValue
☐	asimenv.startup	projectDir	string	./simulation	
☑	adexl.gui	showSimLogForOnePointSim	boolean	t	nil
☑	asimenv	showConvertNotifyDialog	boolean	t	nil
☑	adexl.gui	showRunLog	boolean	t	nil
☑	adexl.distribute	useSameProcess	boolean	t	nil

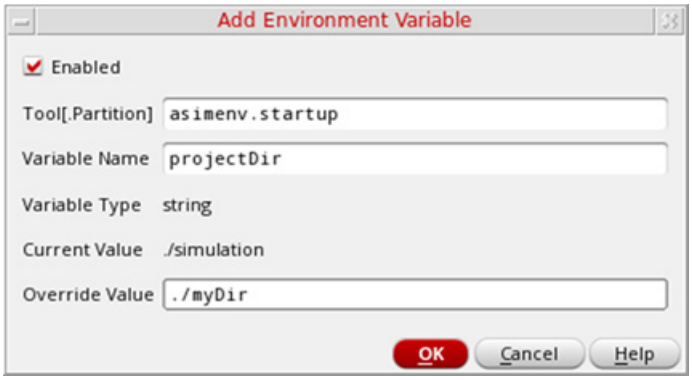
© 2023 All Rights Reserved.

Adding and Removing Environment Variables

To add a new environment variable to the current Verifier session:

1. Right-click anywhere in *Environment Variables* table and choose *Add Environment Variable*.

   The Add Environment Variable form appears.



2. Select the *Enabled* check box to specify a new value to override the current value of an environment variable.
3. In the *Tool.[Partition]* field, specify the tool and partition name of the environment variable that you want to add.
4. In the *Variable Name* field, specify the name of the environment variable.

   On specifying valid values in the *Tool.[Partition]* and *Variable Name* fields, the values for *Variable Type* and *CurrentValue* are populated automatically.
5. In the *Override Value* field, specify a value that will override the *CurrentValue* during simulations.

The specified environment variable is added.

To remove an environment variable in the current Verifier session:

1. In the *Environment Variables* table, select one or more rows of values that you want to remove.
2. Right-click and choose *Remove Environment Variables*.

The selected environment variables are removed.

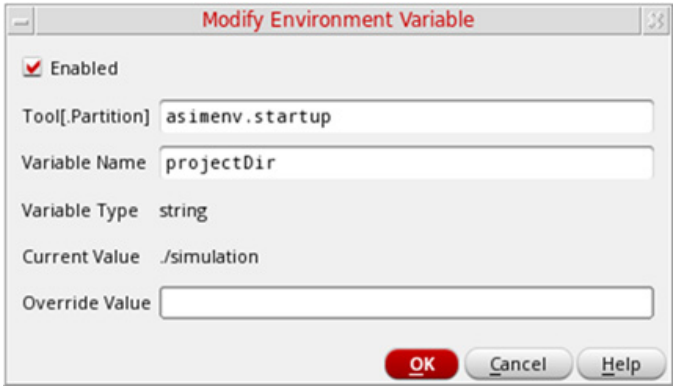
© 2023 All Rights Reserved.

Modifying Environment Variables

To modify an environment variable in the current Verifier session:

1. Right-click an existing row in the *Environment Variables* table and choose *Modify Environment Variable*, or double-click anywhere on a row in the table.

   The Modify Environment Variable form appears.



2. In the form, select or deselect the *Enabled* check box.

   Selecting the check box will allow overriding the current value of an environment variable with the specified value.

3. In the *Tool.[Partition]* field, modify the tool and partition name of the environment variable.

4. In the *Variable Name* field, modify the name of the environment variable.

   On specifying valid values in the *Tool.[Partition]* and *Variable Name* fields, the values for *Variable Type* and *CurrentValue* are populated automatically.

5. In the *Override Value* field, modify the value that will override the *CurrentValue* during simulations.

Saving and Importing Pre-Defined Environment Variables from a CSV File

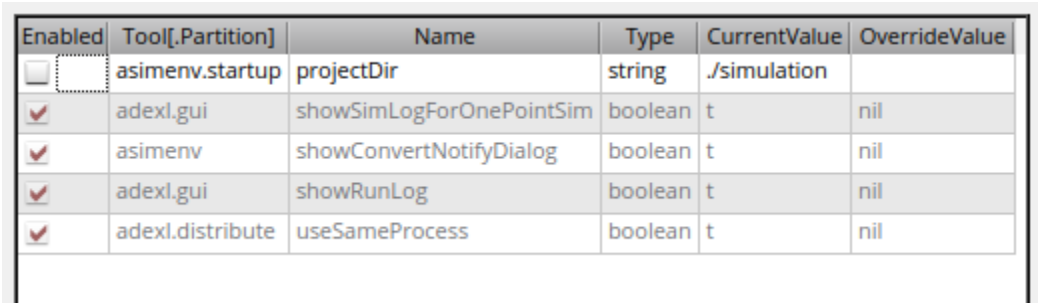
To create a CSV file that contains the environment variables defined in the current Verifier session:

1. Set the envVarConfigFile environment variable.

© 2023 All Rights Reserved.

2. Create a new Verifier cellview.
3. Choose *Edit – Preferences – Run*.

The table loads and displays the environment variables that you have defined in the CSV file. If a CSV file is not specified, the table displays the following default row:



Enabled	Tool[.Partition]	Name	Type	CurrentValue	OverrideValue
☐	asimenv.startup	projectDir	string	./simulation	
☑	adexl.gui	showSimLogForOnePointSim	boolean	t	nil
☑	asimenv	showConvertNotifyDialog	boolean	t	nil
☑	adexl.gui	showRunLog	boolean	t	nil
☑	adexl.distribute	useSameProcess	boolean	t	nil

To import environment variables from an existing CSV file:

1. Right-click anywhere in the *Environment Variables* table and choose *Import Environment Variables from CSV*.

   The Import Environment Variables from CSV file form appears in which you can browse and select a file that you want to add.

2. Click *Open*.

   The environment variables are added to the *Environment Variables* table.

Important

To import environment variables from a CSV file, ensure that the count and names of the column headers in the CSV file and the *Environment Variables* table match.

Exporting Environment Variables to a CSV File

To export environment variables to a CSV file:

1. Right-click anywhere in the *Environment Variables* table and choose *Export Environment Variables to CSV*.

   The Export Environment Variables to CSV file form appears.

2. Browse and specify a file in which you want to export the environment variables.
3. Click *Save*.

© 2023 All Rights Reserved.

The environment variables are exported into the specified file.

Setting the Environment Variable Override Value

To set the override value of an environment variable to its default value:

1. In the *Environment Variables* table, select one or more rows of environment variables for which you want to reset the override value.
2. Right-click and choose *Set Environment Variable Override Value to Default Value*.

The *Override Value* of the selected environment variables is reset to their default value.

To set the override value of an environment variable to its current value:

1. In the *Environment Variables* table, select one or more rows of environment variables for which you want to reset the override value.
2. Right-click and choose *Set Environment Variable Override Value to Current Value*.

The *Override Value* of the selected environment variables is reset to their current value.

© 2023 All Rights Reserved.

Simulation Run Results

After an implementation cellview is run successfully, you can review its run results from Verifier, Virtuoso CIW, and the default Virtuoso application of the implementation cellview. You can also access the simulation log generated by the simulator for an implementation test run.

Related Topics

Viewing Run Results Using the Information Assistant

Viewing Simulation Logs

Viewing Run Results Using the Information Assistant

To view the run results of an implementation cellview using the Information assistant:

1. Open the Information Assistant pane.

   The assistant appears in the right pane.

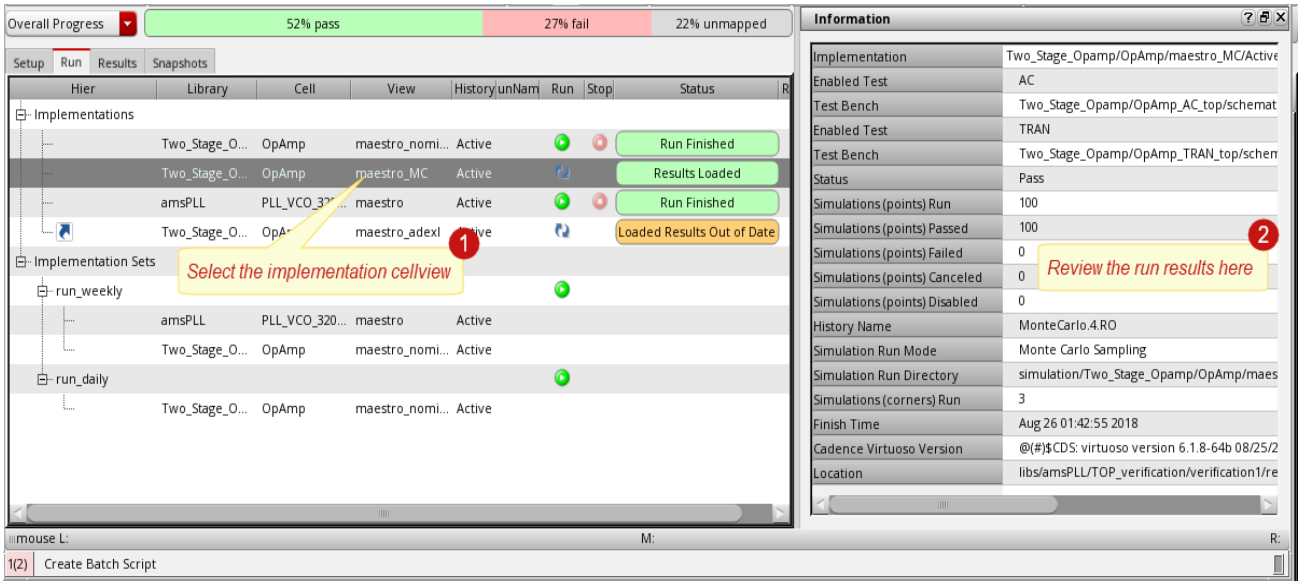
2. Do any of the following, as required:

   - Select the implementation cellview. The Information assistant displays the simulation run details, such as the run mode, tests, simulation corners, run history, and other details.
     - The selected implementation cellview can be a cellview that was run from Verifier, or a cellview whose results were loaded in Verifier.
     - You can copy the value of a result field. For this, right-click the result field and choose *Copy String*. You can also select the field and press `Ctrl+C` to copy the text.
     - When you right-click a result field, Verifier also provides the options to view and edit the results in the main application of the implementation cellview. These options are also available in the popup menu in the *Setup* tab.
   - Right-click the implementation cellview whose results can be loaded in Verifier and select *View Implementation Results*. Verifier launches the application of the implementation cellview, such as ADE Assembler, where the results are displayed. If the implementation cellview uses the integrated history management, you receive a popup asking you to convert the cellview to use the separate history management scheme.

© 2023 All Rights Reserved.

- View the simulation run status and details in the Virtuoso CIW and the simulation application log.

The following figure illustrates how to access the run results of an implementation cellview in the Information assistant.



Related Topics

Results of Failed Outputs or Tests

Viewing Results from an Incomplete History

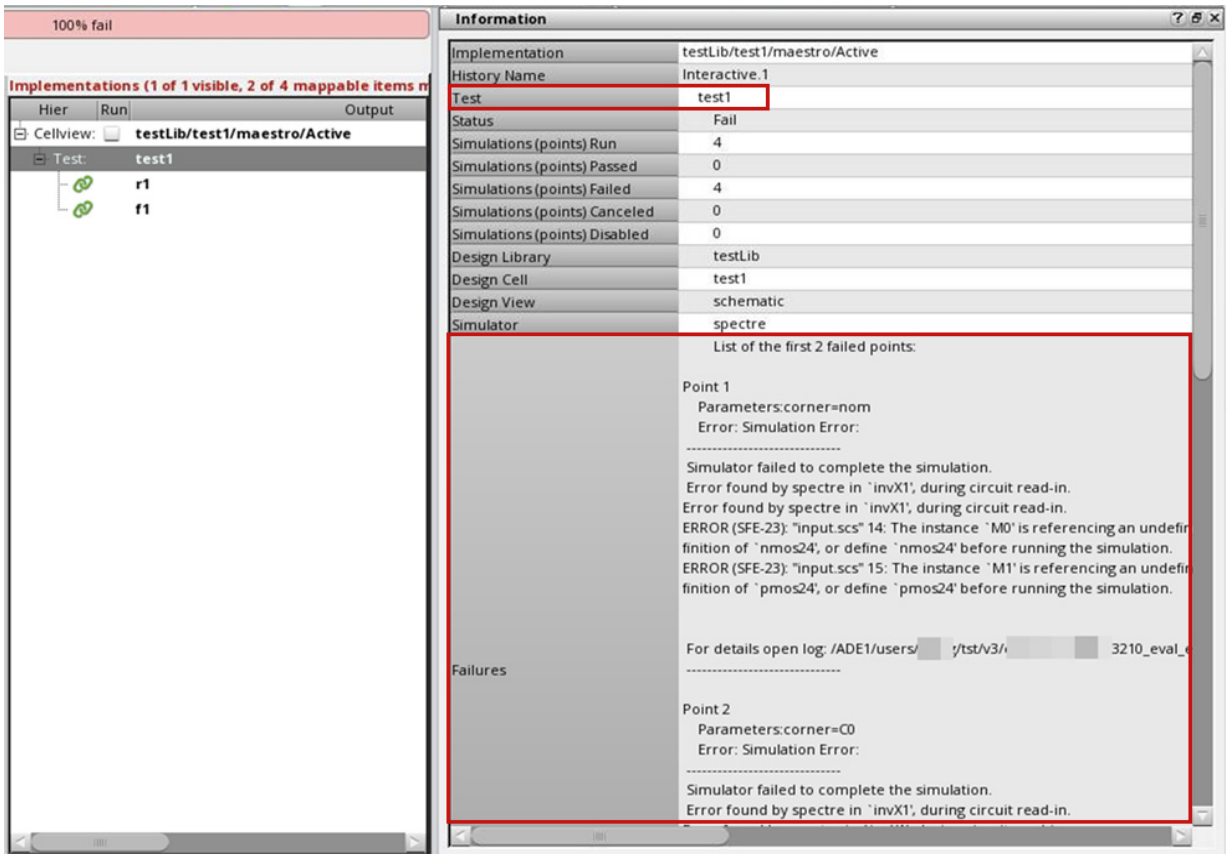
Results of Failed Outputs or Tests

You can view the details of the tests or outputs that have failed simulation checks. These details are shown in the *Failures* row of the Information assistant. The *Failures* row shows the first failure from the complete list. However, you can control the maximum number of failed tests or outputs that are shown by using the maxFailures environment variable.

The following figures show the *Failures* row in the *Information* assistant.

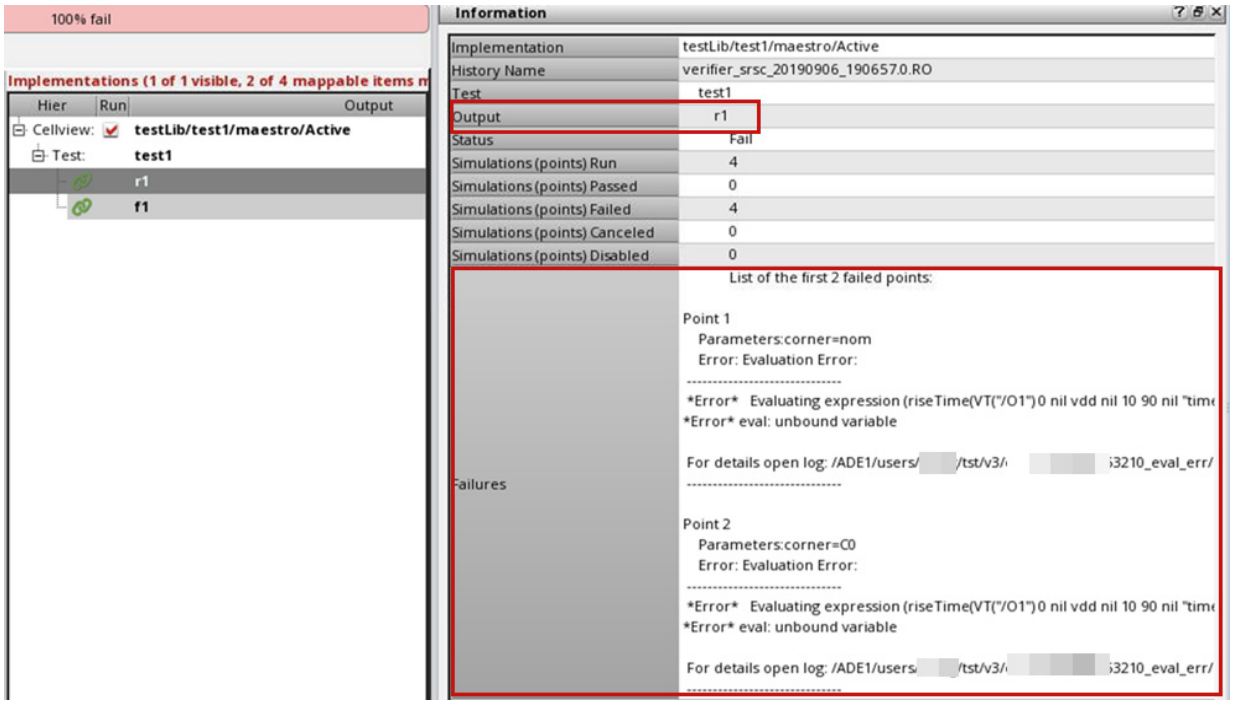
© 2023
All Rights Reserved.

- When `maxFailures=2` for test failures:



© 2023
All Rights Reserved.

- When `maxFailures=2` for output failures:



You can also choose to rerun any unfinished or failed points from the loaded implementation history of selections by choosing the *Rerun Unfinished/Error Points* menu command on the *Run* tab. This menu command is available for local and batch runs, referenced implementations, as well as single or multiple selections on the *Run* tab after the simulation run is complete. Alternatively, you can use the verifRun SKILL function.

Viewing Results from an Incomplete History

In case of a large design testbench or a large design with a run plan, simulations might run for several days. The results become available after the simulation completes and the history is created. Occasionally, it becomes necessary to use an incomplete history to load and view the partially available results of such simulations. Viewing the available results also helps in determining the result trends. Verifier lets you load and view the partial results of in-progress or incomplete simulations.

To load external results from an incomplete history:

1. In the CIW, set the environment variable allowLoadingIncompleteHistory.

   Setting this variable shows the run status from the maestro view in the Information assistant as well as the tooltip of the *Status* column on the *Run* page.

© 2023 All Rights Reserved.

2. In ADE Assembler, run a simulation as follows:

   a. Open a `maestro` cellview.

   b. Click the *Run* button.

3. In Verifier, view the simulation results as follows:

   a. Open a `verifier` cellview that contains the same `maestro` cellview as an implementation.

      Ensure that the corresponding requirements exist in the *Requirements* pane. If not, right-click the implementation and select *Create Requirements Hierarchically*.

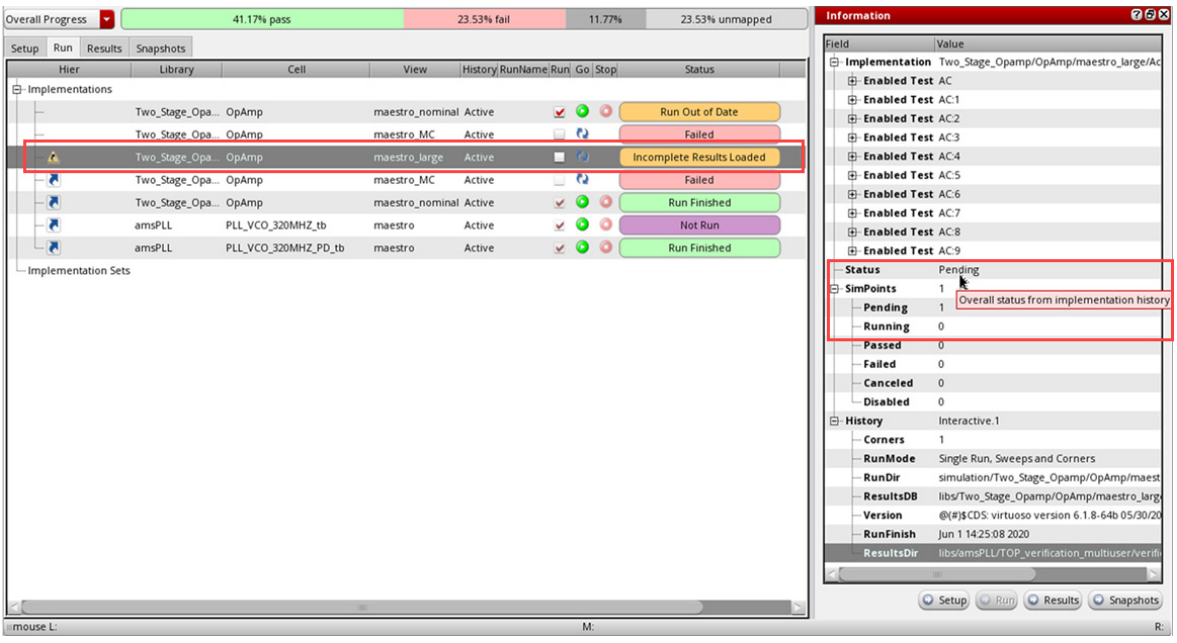
   b. Click the *Reload Simulation Results* button.

   c. The external results are loaded.

   d. Click the *Run* tab.

      The *Run* page shows `Incomplete Results Loaded` in the *Status* column. When you click the implementation, the Information assistant displays *Pending* or *Running* in the *Status* column depending on the number of simulations or points that are still pending or running.

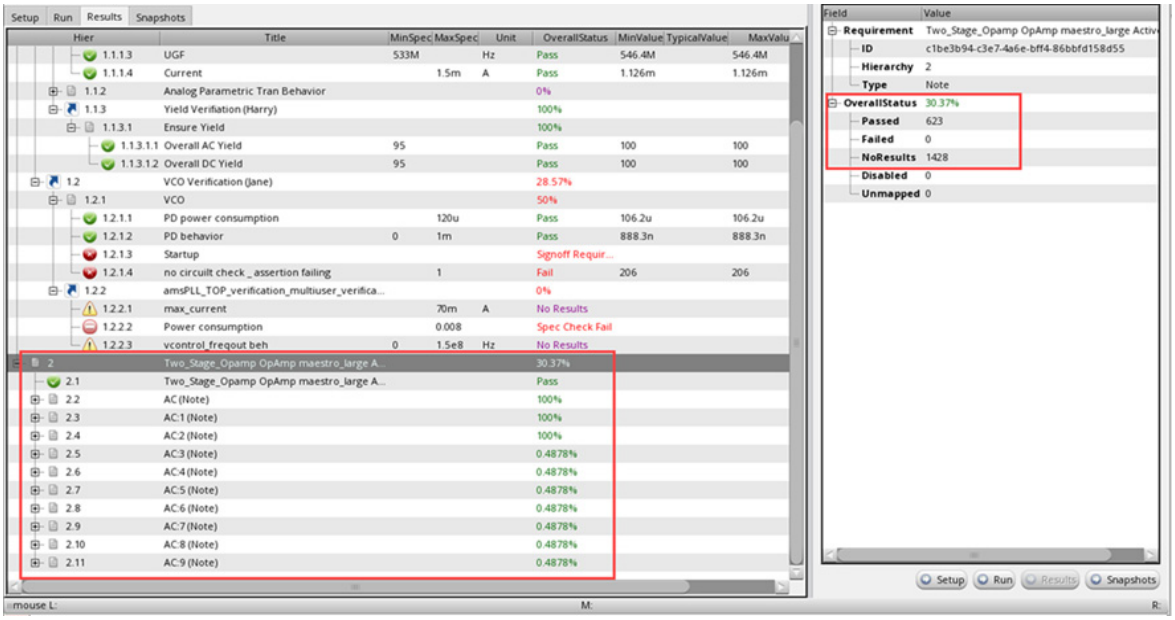
      Note: Only implementations with the *Run* check box deselected are supported.



   e. Click the *Results* tab.

© 2023
All Rights Reserved.

The *Results* page lets you evaluate outputs with partial or incomplete results. When you click the implementation, the Information assistant shows the partially available results in the *OverallStatus* row.



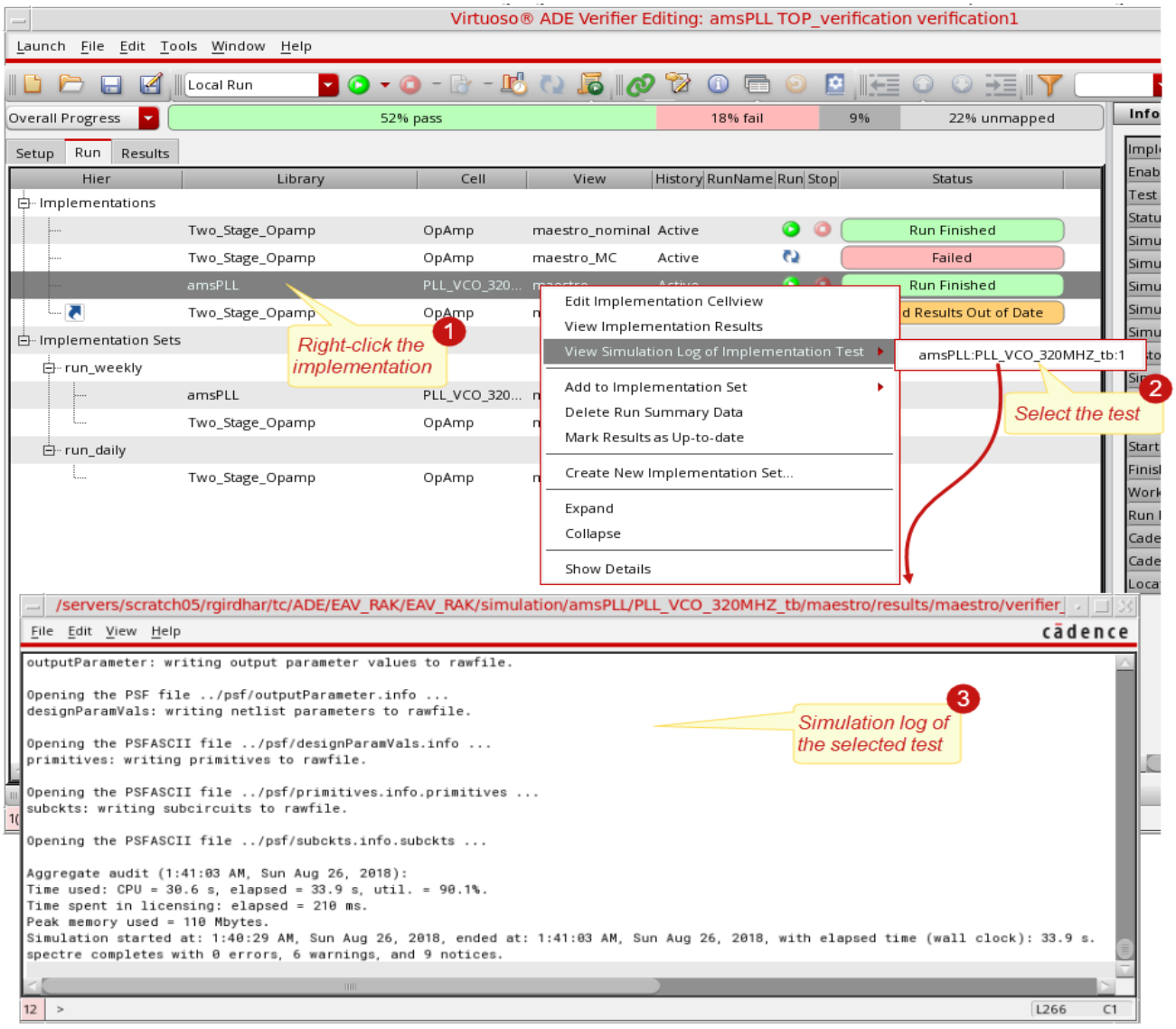
Viewing Job Policies for Implementations

In case of a large design testbench or a large design with a run plan, simulations might run for several days. The results become available after the simulation completes and the history is created. Occasionally, it becomes necessary to use an incomplete history to load and view the partially available results of such simulations. Viewing the available results also helps in determining the result trends.

© 2023
All Rights Reserved.

Viewing Simulation Logs

The simulator application, such as Spectre Circuit Simulator, that runs implementation cellviews logs details of each test run. You can access such simulation logs from Verifier. The following figure illustrates how you can access the simulation log of a test.



You can access the simulation log of an implementation cellview if the simulation has completed and the `results` directory exists. Verifier must have access to the history of the run. The test must be defined and enabled in the history.

© 2023
All Rights Reserved.

To view the simulation log of an implementation test:

1. Do one of the following:
   - Select the implementation from the *Setup* or *Run* tab.
   - Select a requirement mapped to the implementation in the *Setup* or *Results* tab.

   You can also select a status value in the *Run* or *Result* tab.
2. Right-click, select *View Simulation Log of Implementation Test*, and then select the test name.

   If the implementation cellview uses the integrated history management, you receive a popup asking you to convert the cellview to use the separate history management scheme.

The simulation log is displayed in a new text window.

Batch Scripts in ADE Verifier

You can generate a batch script from Verifier, which is also referred to as a regression script. You can use this script to start Verifier from the command line in non-graphical mode with a cellview and run its implementation cellviews.

For information on simulating implementation cellviews from the Verifier graphical user interface, see Simulation of Implementations.

 Video

For a video overview of the Verifier batch scripts and how to set a cron job to run a script periodically, see *Generating and Using Batch Scripts* on Cadence Online Support.

Related Topics

Generating a Batch Script

Reviewing and Refining a Batch Script

Running and Stopping a Batch Script

Scheduling Runs

© 2023 All Rights Reserved.

Generating a Batch Script

Before you generate a batch script, save the Verifier setup for which you want to generate the script to a cellview of type `verifier`.

To generate a batch script for the current cellview:

1. Do one of the following:
   - Click the *Create Batch Script* button on the toolbar: 
   - Choose *Tools – Create Batch Script*.

   The Batch Script File Name dialog box displays.
2. Specify the location and filename of the batch script.

   The default location is the current directory and the default filename is in the format *`currentCellview`*`_run`. For example, `TOP_verification_run`.

   You can specify a new location by changing the directory name. If the specified directory does not exist, it is created automatically.
3. Click *Save*.

Verifier saves the batch script in the specified location using the specified filename.

Reviewing and Refining a Batch Script

You can open the batch script you generated from Verifier in an editor and review the script.

To generate a script:

➡ Choose *Tools – Create Batch Script*.

© 2023 All Rights Reserved.

The following example shows a sample script generated in ADE Verifier.

```
#!/bin/csh -f
#
# Script to run implementations of cellview
opamp090.full_diff_opamp_AC:verification1
setenv vlib  "opamp090"
setenv vcell "full_diff_opamp_AC"
setenv vview "verification1"
virtuoso -nograph -log full_diff_opamp_AC_run.batch.log $* << SKILLSCRIPT
let(((retval 255))
    unwindProtect(
        let((sess failedReqs)
            when(sess = verifOpenCellView("$vlib" "$vcell" "$vview" ?mode "r"
            ?openWindow nil)
                verifRun(sess ?mode "Local Run" ?waitUntilDone t)
                verifSetOptionVal(sess "batchhtmlreport" "$vcell.html")
                verifPublishHTML(sess)
                failedReqs = setof(r verifGetReqs(sess ?mappable t)
                verifGetReqStatus(sess r) != "Pass")
                when(failedReqs
                    info("The following requirements did not pass:\n")
                    foreach(req failedReqs
                        info("\t%L\t%L\n" req verifGetReqStatus(sess req))
                    )
                )
                verifCloseSession(sess ?saveIfModified nil)
                retval = length(failedReqs)
            )
        )
        begin(
            info("Exiting Virtuoso (%d)\n" retval)
            exit(retval)
        )
    )
)
SKILLSCRIPT
```

The script used in this example is standard `.csh` script that sets up some shell environment variables to specify the library, cell, and view. With this script, Virtuoso is launched in the optionally used `-nograph` mode. The script sends the commands between the two `SKILLSCRIPT` segments to Virtuoso as SKILL commands to run.

© 2023 All Rights Reserved.

The Verifier SKILL functions used in this SKILL script do the following:

Function	Description
`verifOpenCellView`	Opens the verifier cellview in read mode without any window.
`verifRun`	Runs all the simulations and ensures waiting before the runs are finished before proceeding. A few options that you can use with `verifRun` are: ■ `?mode`: Change the mode to `Batch Run` or `Batch Run with SPACE`, `Local Run` or `Local with SPACE`. ■ `?implementation`: Specify an implementation or list of implementations to run instead of defaults. ■ `?impSet`: Specify the name of the implementation set to run.
`verifSetOptionVal`	Specifies where a batch report file will be generated.
`verifPublishHTML`	Publishes the report from the location specified by `verifSetOptionVal`.
`info`	Generates a summary of the requirements.
`failedReqs`	Prints out a summary of the requirements that did not pass.
`verifCloseSession`	Closes the session without saving it.
`exit`	Exits with the number of failed requirements as the return value.

The use of `unwindProtect` is to ensure sure that the script always exits even if there is a SKILL error. In case of an error, the return value is `255`.

If required, you can edit the batch script appropriately. For example, if you want to run the implementation cellviews that belong to a specific implementation set, add `-run` *`implementation_set_name`* to the script. For more details on refining the batch script for a scheduled run, see Scheduling Runs.

© 2023 All Rights Reserved.

When a batch script run is completed, the overall verification status is displayed by default. The following snippet shows some example messages that Verifier displays at the end of a batch script run. The last message provides the overall verification status.

```
...
...
Verifier: 11:16:33: INFO (Verifier-80084) The cellview 'amsPLL/
PLL_VCO_320MHZ_PD_tb/maestro/Active' run finished. Summary: Started 'Dec 15
11:16:09 2015'. Finished 'Dec 15 11:16:30 2015'. Took '21s' to run.
Verifier: 11:16:33: INFO (Verifier-80083) Finished running '3' job(s).
Verifier: 11:16:33: Created HTML report file './TOP_verification.html'.
Verifier: 11:16:33: The overall verification status of the project is: FAIL
Verifier: 11:16:33: Verifier is exiting.
```

If you want to view the overall verification status of only specific requirements, instead of the status of the project, use `-returnStatus` *`RequirementIDlist`*. You can specify a list of requirement IDs or an implementation set that contains the requirements.

Running and Stopping a Batch Script

You can run a batch script from the command line. You can also schedule a batch script to run through a cron job. For details, see Scheduling Runs.

To run a batch script:

➡ Run the script from the command line.

If you want to run the implementation cellviews that belong to a specific implementation set, use the optional `-run` argument with the implementation set name. For example:

```
$ ./TOP_verification_run -run run_weekly
```

Verifier runs the specified batch script and displays appropriate messages.

When the batch script run is completed, it either returns `0` to indicate the successful completion of the script or `1` to indicate failure. For example, the following return value indicates that the run was successfully completed:

```
$ echo ?
$ 0
```

To stop a running batch script:

1. Press `Ctrl + Z` to stop the run manually.

   Note: Do not press `Ctrl + C` to stop the script. Using this stops the script, but the simulations continue to run in the Virtuoso session. The Verifier session also does not

© 2023 All Rights Reserved.

stop.

2. Run the following command to kill the Virtuoso and Verifier processes.

   ```
   $ kill -9 $$
   ```

The batch script run stops, along with Virtuoso and Verifier.

Scheduling Runs

You can schedule implementation cellview runs using batch scripts. For this, generate the batch script and set cron jobs to run those scripts, as required. For details on batch scripts, see Batch Scripts in ADE Verifier.

If you want to schedule the run of only specific implementation cellviews, organize those cellviews in an implementation set and edit the script to run that set. Then set a cron job to run the batch script.

The following set of steps describe how to set a batch script and schedule its cron job:

1. Open the Verifier cellview.
2. Organize the implementation cellviews in implementation sets.

   For this example, assume that `run_weekly` has been created and some implementation cellviews are added to this implementation set. For more details, see Implementation Sets.
3. Create a batch script called `weekly_run_script`.

   For more details, see Batch Scripts in ADE Verifier.
4. Edit the script appropriately.

© 2023 All Rights Reserved.

For example, edit `weekly_run_script` to run the implementation cellviews in the set `run_weekly`. The following snippet shows a sample script that uses the `run_weekly` implementation set.

```
#!/bin/csh -f
#
# Script to run implementations of cellview
opamp090.full_diff_opamp_AC:verification1

setenv vlib  "opamp090"
setenv vcell "full_diff_opamp_AC"
setenv vview "verification1"

virtuoso -nograph -log full_diff_opamp_AC_run.batch.log $* << SKILLSCRIPT
let(((retval 255))
    unwindProtect(
        let((sess failedReqs)
           when(sess = verifOpenCellView("$vlib" "$vcell" "$vview" ?mode "r"
           ?openWindow nil)
                verifRun(sess ?mode "Local Run" ?waitUntilDone t ?impSet
                "run_weekly")
                verifSetOptionVal(sess "batchhtmlreport" "$vcell.html")
                verifPublishHTML(sess)
                failedReqs = setof(r verifGetReqs(sess ?mappable t)
                verifGetReqStatus(sess r) != "Pass")
                when(failedReqs
                    info("The following requirements did not pass:\n")
                        foreach(req failedReqs
                        info("\t%L\t%L\n" req verifGetReqStatus(sess req))
                    )
                )
                verifCloseSession(sess ?saveIfModified nil)
                retval = length(failedReqs)
           )
        )
        begin(
           info("Exiting Virtuoso (%d)\n" retval)
           exit(retval)
        )
    )
)

SKILLSCRIPT
```

5. Add the cron job entry, as required.

   The cron job calls do not load any environment settings automatically, such as sourcing `.cshrc`. Therefore, ensure that the requirement settings, and software and project setups are sourced. Also ensure that the current directory is changed appropriately before the batch script is run.

The cron job runs the batch script at the scheduled time.

© 2023 All Rights Reserved.

6

Managing Verification Results

This topic provides information on how you can access the verification status of your project managed through Virtuoso ADE Verifier (Verifier).

The status of your design verification project depends on various factors, such as the number of unmapped or missing requirements. For example, if your verification plan is incomplete, and the overall verification status is 100 percent, the verification project cannot be deemed as complete.

ADE Verifier lets you access the overall verification status of your project. You can drill down to the detailed status of each verification requirement in the project to generate various types of design verification reports.

It is possible to manually override failed requirements by signing them off. You can also sign off the requirements of the type *Manual*.

Related Topics

Factors that Impact Verification Status

Reviewing the Current Overall Verification Status

Verification Status of Requirements

Requirements Signoff in Virtuoso ADE Verifier

Detailed Verification Report

© 2023 All Rights Reserved.

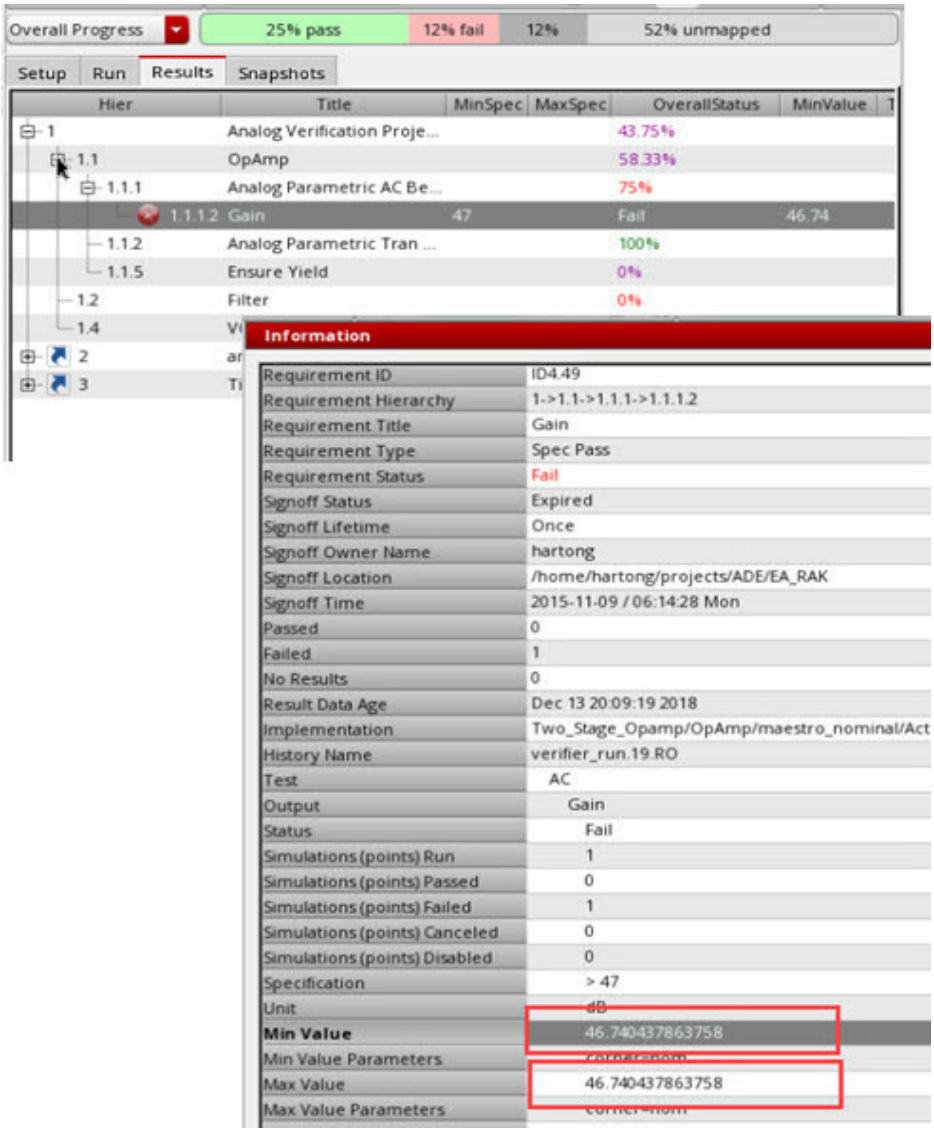
Factors that Impact Verification Status

The status of your design verification project depends on the following factors:

- The number of mapped and unmapped requirements of the type *Spec Pass* or *Ran OK*.

  To get a realistic verification status, map all the requirements of the type *Spec Pass* or *Ran OK* to their implementations.

- The implementation output fails with errors. This indicates that some of the simulation points could not be finished correctly due to some error in ADE Explorer or ADE Assembler, such as a simulation error. The min/max values consider only the finished simulations results. Therefore, these values with errors should be used cautiously.



© 2023
All Rights Reserved.

- The successful completion or failure of the implementation cellview runs.

  If an implementation cellview run fails, the associated requirements of the type *Ran OK* also fail verification. Additionally, the requirements mapped to the outputs of the cellview do not have results for verification checks.

  When you run an implementation cellview, only the enabled tests of that cellview are run. The disabled tests are not run. The overall status of the requirements mapped to a disabled test or its outputs is *No Results*, which impacts the verification project status.

- The results of the successful implementation cellview runs.

  Verifier compares the simulation result of an implementation output with the specification used by the mapped requirements. If the result meets the specification, the mapped requirements pass verification. Otherwise, they fail verification.

  The results of test or outputs are reported as failed when one of the following conditions are met:

  - Number of canceled points for a mappable item is greater than `0`.
  - Number of failed points for an implementation, run, reliability, or test is greater than `0`. Failed points can be a result of simulation errors, netlist errors, upload errors, or other errors.
  - The output or test for a statistical output fails.

  All these errors, except for errors reported for canceled points, can be ignored by setting the environment variable ignoreError to `t`. The *Results* page displays the failed and canceled points for these tests or outputs using a tooltip.

- The manual sign-off status of the failed requirements and the requirements of the type *Manual*.

  You can sign off the requirements of type *Manual*. Verifier also lets you sign off the requirements that failed verification, is not mapped to any implementation items or failed the specification check. Verifier considers signed off requirements as requirements that passed verification.

  For details on the manual sign-off feature, see Requirements Signoff in Virtuoso ADE Verifier.

- The metric prefix from specification units during verification.

  For details on metric prefix evaluation, see Evaluation of Metric Prefix.

- The output expressions for implementations. If an output has no expressions specified in ADE Assembler, the *Overall Status* shows `No Results`.

© 2023 All Rights Reserved.

Changes in these factors impact the overall design verification status. For example, if you sign off some failed requirements, the percentage of the overall verification progress increases.

The verification failures can include requirements that are not mapped, mapped implementations that are not simulated or do not have results, failed specification checks, and cases where simulation results did not match the specifications.

Location of Run Summary Data

Verifier stores the run summary data (PALS) file as `.xml` files in separate locations, based on whether the cellview is editable or read-only.

- When the cellview is editable, Verifier stores the run summary data of a local or referenced implementation in the `results` directory of the current cellview.
- When the cellview is read-only, Verifier stores the run summary data in the following locations:
  - For a local implementation: <*project directory*>/<*library*>/<*cell*>/<*view*>/<*results*>
  - For a referenced implementation: <*project directory*>/<*ref_library*>/<*ref_cell*>/<*ref_view*>/<*results*>

    For more information on project directories, see Project Directory Setup.

When you open an existing Verifier cellview or update a referenced cellview, Verifier loads the existing and last modified run summary data file from the default `results` directory in the current cellview or from these locations.

Evaluation of Metric Prefix

You can check for any mismatch in the implementation specifications with the requirement specifications mapped to that implementation. This happens when the *Requirement specification and check to implementation* option is set in the Preferences form.

However, by default this check only evaluates the specification numbers and the unit. It does not do any calculations to check if the value and unit combination matches.

```
Requirement: MaxSpec = 10m, unit = V;
Implementation: MaxSpec = 10, unit = mV;
```

This will be flagged as a mismatch.

© 2023 All Rights Reserved.

To check for such matches or mismatches, enable the preference option *Evaluate metric prefix* on the *General* tab.

Note: The *Evaluate metric prefix* option is hidden when the *Specification Search Order* is set to *Implementation Specification* because no check for specification is performed in such cases.

Enable *Check unit*, *Evaluate metric prefix* and use *Requirement specification with check* in the *Preferences* form. The metric prefix of the requirement specification unit is used for both specification check and evaluation. Assume that the values are as follows:

```
Requirement: MaxSpec = 10m, unit = V;

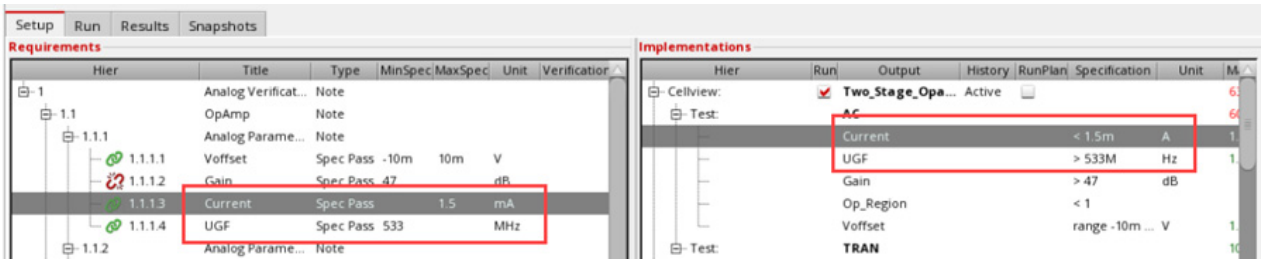
Implementation: MaxSpec = 10, unit = mV, Implementation results: minValue = 1.5,
maxValue = 5.5;
```

When evaluation is done, the status is as follows:

```
Spec Check: Pass (10m V == 10 mV);
```

No specification error is reported. The unit of the requirements is used for the displayed results.

```
Result status shown: Verification: OverallStatus = Pass, MinValue = 1.5m, MaxValue
= 5.5m , Unit = v.
```



In the above figure, the units are different in the requirement and implementation specifications. When you enable *Evaluate metric prefix*, the specification check is 'pass' as the numerical values are identical when the unit prefix is considered.

You can also enable the *Evaluate metric prefix* preference by using the environment variable evaluateMetricPrefix.

The *Evaluate metric prefix* preference only checks the first single prefix, such as:

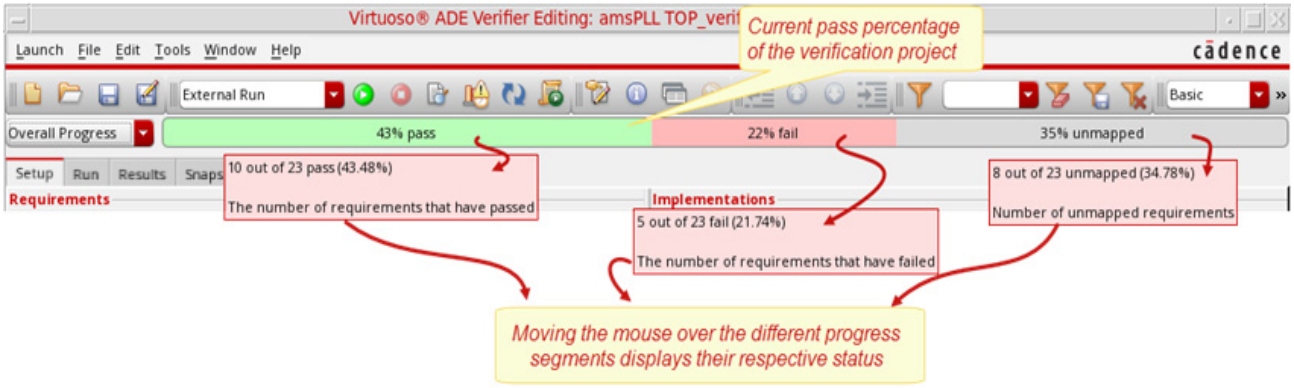
```
Y|Z|E|P|T|G|M|K(?!elvin)|k(?!ilo)|%|m(?!il|persec)|u|µ|n|p|f|a|z|y
```


© 2023
All Rights Reserved.

Here the supported prefix is `K` but not `Kelvin`; `k` but not `kilo`; `m` but not `mil` or `mpersec`. Except the above prefixes, complex combinations are not supported.

Reviewing the Current Overall Verification Status

Verifier displays the current overall design verification status of your project on the *Overall Progress* bar. You can view further details in the status tooltip.



To display the overall verification status, if it is not already displayed:

➡ Choose *Overall Progress* on the toolbar.

The current overall verification status displays with the *pass*, *fail*, and *unmapped* percentage.

To display the status snapshot tip:

➡ Place the pointer over the required segment on the *Overall Progress* bar.

The status snapshot tip includes the following information:

- Requirements information, including the total number of mapped, unmapped, and manual requirements.
- The status of the implementation runs.
- The percentage of requirements that passed verification, along with the number of failed and unmapped requirements, and requirements that do not have results for determining their verification status.

© 2023 All Rights Reserved.

Additional Information

- The percentage for *pass* rate is rounded down whereas the percentage for *fail*, *unmapped*, and *no results* is rounded up. This ensures that the rounding does not lead to overly optimistic view of the results.

  For example:

  - 99.8% *pass* is not rounded off as 100% *pass*.
  - *Pass* rate: 34.899%, *fail* rate: 64.1%, *unmapped*: 1.001% - Here, the rounded values are shown as *pass*: 34% and *fail* rate: 65% and *unmapped*: 2%.

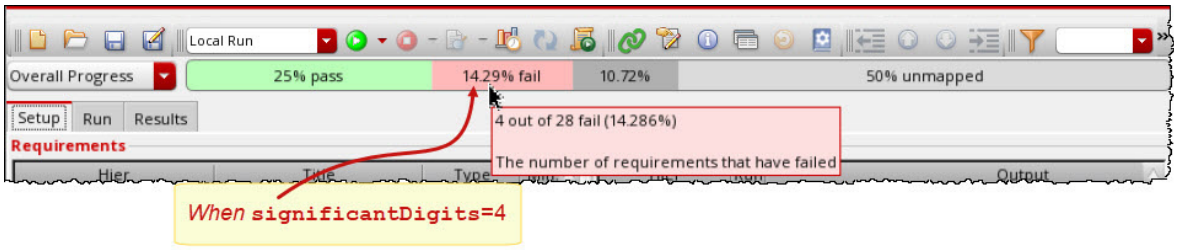
  As a result of the rounding schema used, the sum of all values may deviate from 100%, as shown in the above example.

- To specify the rounding schema for the calculation of *Overall Progress, OverallStatus, MinValue*, *TypicalValue*, and *MaxValue* on the *Results* tab, set the significantDigits environment variable.

  The calculated values are shown as per the rounding schema selected in the *Overall Progress* bar. However, the values displayed in the tool tips are based on the calculation (`significantDigits`+1). For example, when `significantDigits` is set to 2, the pass rate of 34.899% will be displayed as 34% in the *Overall Progress* bar and 34.8% in the corresponding tool tip.

  You can observe the following illustrations to understand the calculation of values on the *Overall Progress* bar using the rounding schema:

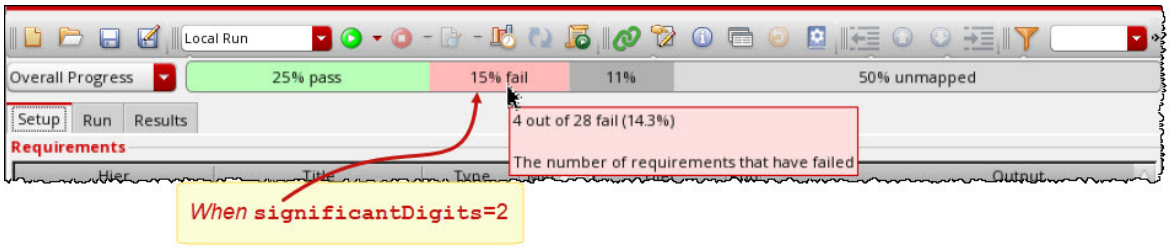
  - When `significantDigits` is set to default value:



  Setting `significantDigits` to `4` means that the *Overall Progress* and *OverallStatus* are calculated by using default value.

© 2023
All Rights Reserved.

- When `significantDigits` is set to 2:



  Setting `significantDigits` to 2 calculates `100%` as '`1e+2%`'.

- Referenced requirements owned by other owners, and not the current user, are excluded from the overall verification progress bar and tool tip. Manual requirements are excluded from the mapped and unmapped requirements count.

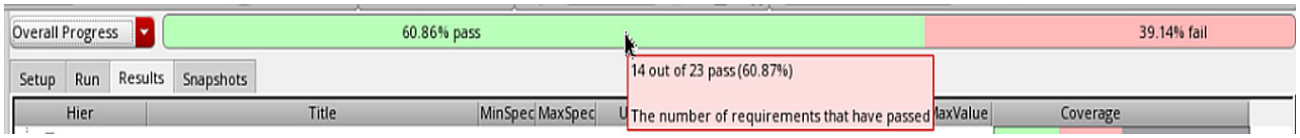
Tip

Click the *Setup* tab and the right and left pane titles display the overall mapping percentage of the implementations and requirements, respectively.

Overall Progress, Coverage, and Overall Coverage

There might be instances when you see a difference in the `Pass` and `Fail` percentages in the values shown for *Overall Progress* and *Overall Coverage* on the Verifier toolbar.

The calculation for *Overall Progress* is based on the number of requirements. For example, the following illustration shows the tooltip as "`14 out of 23 pass`".



In this example, the total number of requirements with results is `23`, and the number of requirements that have passed is `14`. The *Overall Progress* considers only the requirements that have passed or failed.

Coverage in Verifier is calculated by considering all the design point combinations that are added to verification spaces and available in the implementation history. The coverage values are reported as "`x% passed, y% failed`", which can be interpreted as:

`x%` design points from SPACE are `pass --` Green

© 2023
All Rights Reserved.

`y%` design points from SPACE are `fail --` Red

`(100%-(x+y)%)` design points from SPACE are `not covered --` Grey

Note: If a design variable defined in a verification space that is assigned to an implementation, is not needed for simulation then Verifier ignores this variable for calculating the overall coverage.

The *Overall Coverage* value is calculated by considering all the design points specified in the verification spaces and reporting these points as `Pass`, `Fail`, or `No Coverage`.

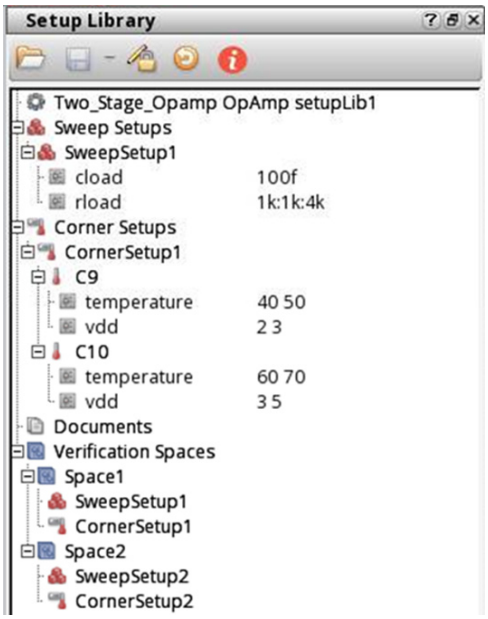
Consider the following figure that shows the tooltip as "`4 out of 45 pass`".



In this example, the total number of design points is `45`, and the number of points that have passed is `4`. The 45 design points include points that are defined in the verification space, as well as those points that are not defined in verification spaces but simulated during a run. The 4 points that show as `pass` are those points that are successfully implemented or covered.

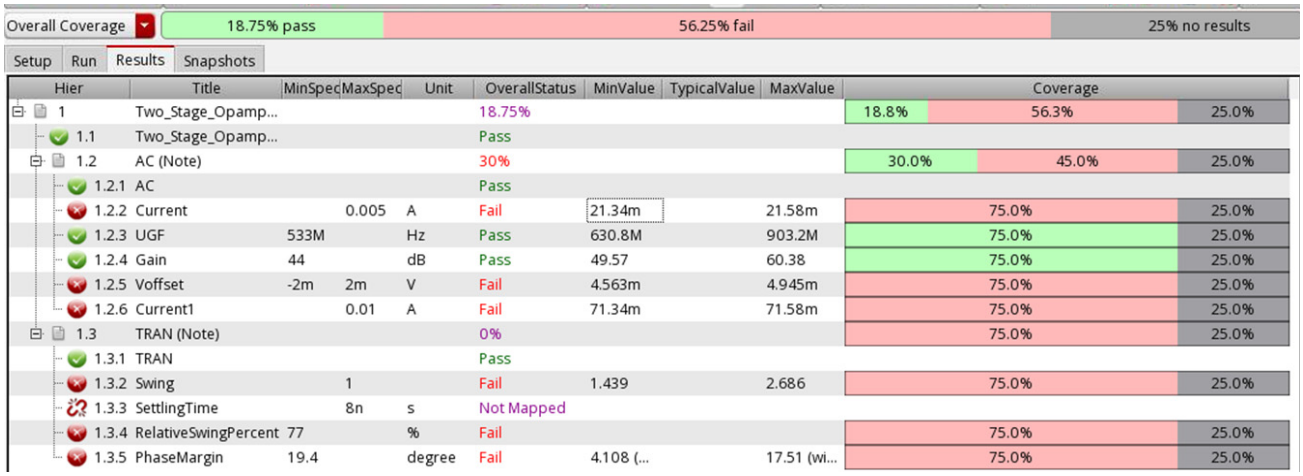
When you select the *Overall Coverage* progress bar, the *OverallStatus*, *MinValue*, and *MaxValue* values are updated. Calculation of these values is based only on the design points specified in the assigned verification space. All other design points are ignored.

For example, consider the following Setup Library shown in the following illustration:



© 2023 All Rights Reserved.

Based on the design points defined in this library, the *OverallStatus*, *MinValue*, and *MaxValue* are as follows:



Hier	Title	MinSpec	MaxSpec	Unit	OverallStatus	MinValue	TypicalValue	MaxValue	Coverage		
1	Two_Stage_Opamp...				18.75%				18.8%	56.3%	25.0%
1.1	Two_Stage_Opamp...				Pass						
1.2	AC (Note)				30%				30.0%	45.0%	25.0%
1.2.1	AC				Pass						
1.2.2	Current		0.005	A	Fail	21.34m		21.58m	75.0%		25.0%
1.2.3	UGF	533M		Hz	Pass	630.8M		903.2M	75.0%		25.0%
1.2.4	Gain	44		dB	Pass	49.57		60.38	75.0%		25.0%
1.2.5	Voffset	-2m	2m	V	Fail	4.563m		4.945m	75.0%		25.0%
1.2.6	Current1		0.01	A	Fail	71.34m		71.58m	75.0%		25.0%
1.3	TRAN (Note)				0%				75.0%		25.0%
1.3.1	TRAN				Pass						
1.3.2	Swing		1		Fail	1.439		2.686	75.0%		25.0%
1.3.3	SettlingTime		8n	s	Not Mapped						
1.3.4	RelativeSwingPercent	77		%	Fail				75.0%		25.0%
1.3.5	PhaseMargin	19.4		degree	Fail	4.108 (...		17.51 (wi...	75.0%		25.0%

Selecting the option *Show results for Verification Space parameter conditions* ensures that Verifier calculates the coverage based on design points from verification spaces for all subsequent simulations. Additionally, this preference ensures the following:

- The Verifier toolbar always shows the *Overall Coverage* progress bar when you close and reopen the Verifier cellview.
- Clicking the *Update Coverage* button on the Verifier toolbar updates the *MinValue*, *MaxValue*, *TypicalValue,* and *OverallStatus* values.
- The *TypicalValue* column shows a value if a typical design point falls in the range of values specified in a verification space. If there are multiple typical design points, the column shows `various`. If there is no typical design point, the cell is empty.

Verification Status of Requirements

In a requirements-driven verification flow, it is important to review the status of each requirement to identify the requirements that need attention. You can access the verification results of each requirements in Verifier. You can also filter the requirements hierarchy based on the verification results.

Typically, you investigate the failed requirements to determine the corrective actions. You can make appropriate changes to the design so that the results meet the specifications, and re-simulate the implementation cellview. Alternatively, if the specifications have scope for updates, you can change the specifications in the implementation cellview and the requirement so that the results are within the new specifications. To facilitate such actions, Verifier lets you view simulation results and edit the implementation cellview in the default

© 2023 All Rights Reserved.

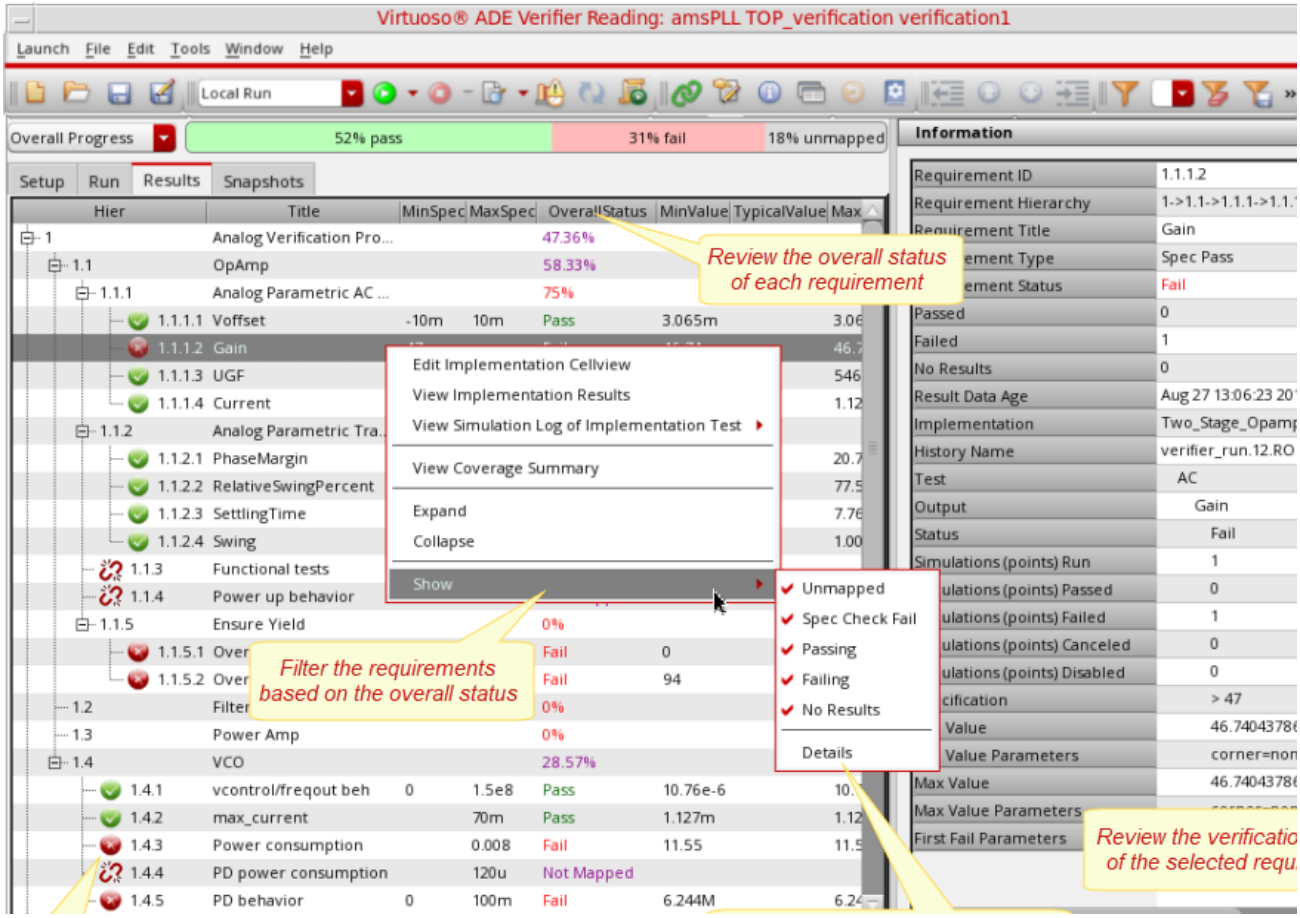
application of that cellview.

When you investigate the verification status, you can also consider making appropriate changes to your verification plan to achieve your design verification goals and quality parameters. For example, you can choose to make the plan more granular by adding new requirements and mapping them with their implementations, or change the specifications of some requirements.

Tip

Typically, the goal of a verification project is to achieve `100%` overall verification status. However, you can consider to tape out when your verification status is `95%` after a careful analysis and a good understanding of the missing `5%` of the verification objectives. By analyzing the verification status of the requirements in Verifier, you can achieve this understanding. If required, you can choose to manually override those `5%` of requirements.

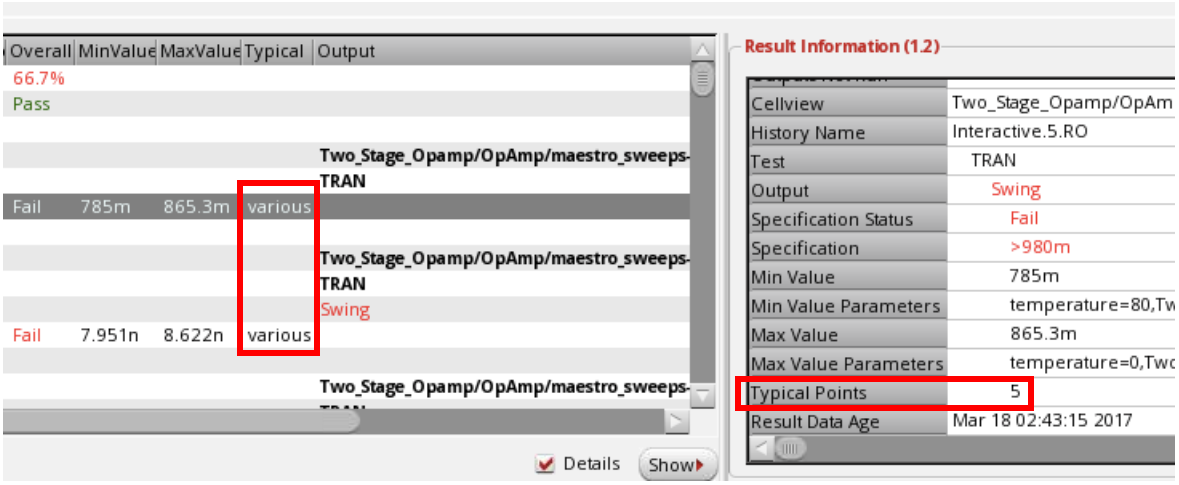
The following figure illustrates how you can identify the verification status and get relevant details from the *Results* tab.



© 2023
All Rights Reserved.

The *Results* tab contains the following columns:

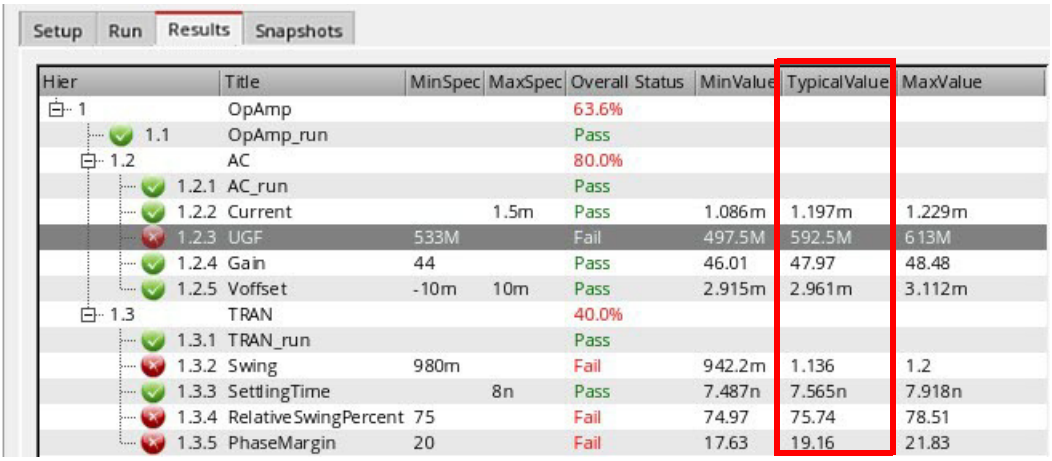
- *Hier*
- *Title*
- *MinSpec*
- *MaxSpec*
- *Unit*
- *Overall Status*
- *MinValue* - Shows the overall minimum value for multiple mappings.
- *TypicalValue* - Shows the typical value for the mapped output according to the typical settings in *Tools – Typical Value Setup*. A blank indicates no matched point is found or the requirement is not mapped to the output. `various` indicates there are multiple matched points or the mapped output has a different typical value.



Verifier allows you to set up the typical and get the value from unique matched point. The typical value is derived by selecting one value out of the matrix of corner and sweep values and matching the specified 'typical' parameters inside Verifier. For each mapped requirement, Verifier analyzes implementation results and chooses one result value from the corner or sweep values being specified. This value is used as the 'typical' value in Verifier. If matching sweep or corner parameter is not found, the *TypicalValue* column can be left blank. Similarly, the column is left blank if multiple values are found, which match the typical parameters. In this case, modify the typical setting accordingly so that

© 2023
All Rights Reserved.

a unique match is found. Typical values are not used for Monte Carlo implementations or when more than one implementations are mapped to one requirement.



Hier	Title	MinSpec	MaxSpec	Overall Status	MinValue	TypicalValue	MaxValue
1	OpAmp			63.6%			
1.1	OpAmp_run			Pass			
1.2	AC			80.0%			
1.2.1	AC_run			Pass			
1.2.2	Current		1.5m	Pass	1.086m	1.197m	1.229m
1.2.3	UGF	533M		Fail	497.5M	592.5M	613M
1.2.4	Gain	44		Pass	46.01	47.97	48.48
1.2.5	Voffset	-10m	10m	Pass	2.915m	2.961m	3.112m
1.3	TRAN			40.0%			
1.3.1	TRAN_run			Pass			
1.3.2	Swing	980m		Fail	942.2m	1.136	1.2
1.3.3	SettlingTime		8n	Pass	7.487n	7.565n	7.918n
1.3.4	RelativeSwingPercent	75		Fail	74.97	75.74	78.51
1.3.5	PhaseMargin	20		Fail	17.63	19.16	21.83

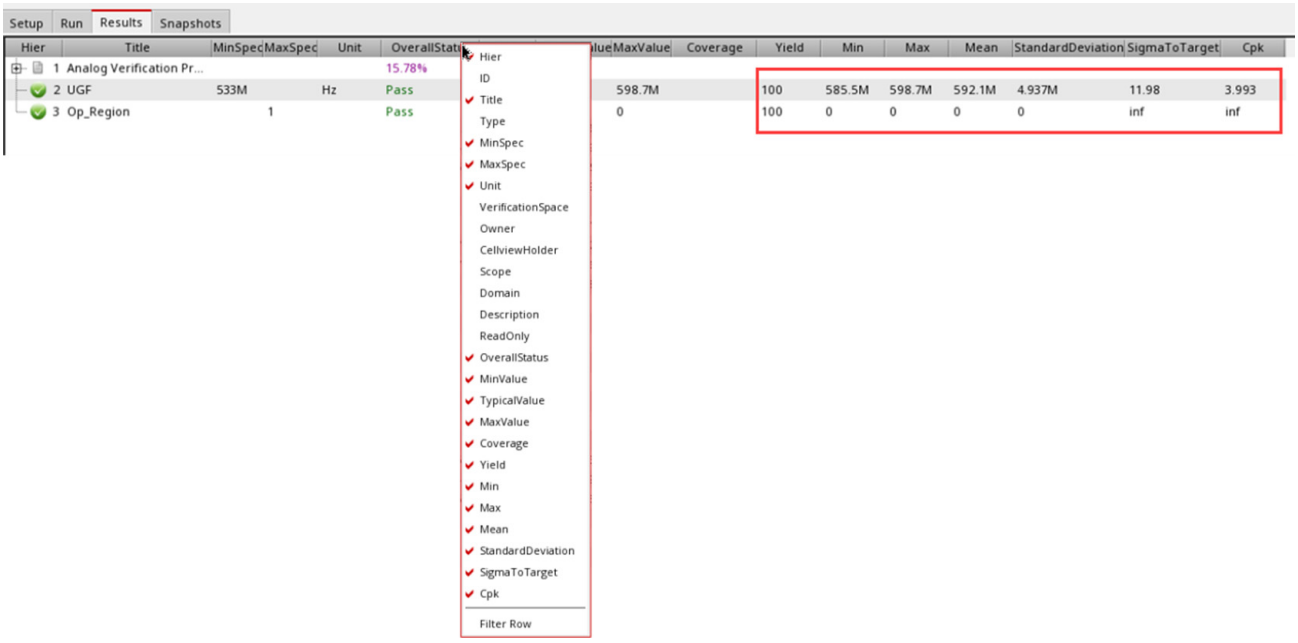
- *MaxValue* - Shows the overall maximum value for multiple mappings.
- *Coverage*

The *Results* tab can also display the following optional columns:

- *ID*
- *Type*
- *VerificationSpace*
- *Owner*
- *CellviewHolder*
- *Scope*
- *Domain*
- *Description*
- *ReadOnly*
- *Mapped*
- *Signoff* - Shows the comments that are added in the Sign Off form. A tooltip also displays the requirement ID with the signoff duration, the name of the person who has given the signoff, comments, name of the requirement owner, location of the verifier cellview, and time of signoff.
- *Yield*

© 2023 All Rights Reserved.

- *Min*
- *Max*
- *Mean*
- *StandardDeviation*
- *SigmaToTarget*
- *Cpk*



To view these optional columns, right-click anywhere in the *Results* tab header row, and select the required columns.

To view additional details related to the implementations mapped to the requirements, right-click anywhere in the *Results* tab, choose *Show*, and select the filter options as required.

In case of Monte Carlo runs with sweeps or *SigmaToTarget* values defined with range specifications, you can get different *Min* or *Max* values for each statistical value. In such scenarios, the values are displayed as <`minValue`>, <`maxValue`> values. In all other cases, the *Min* and *Max* values remain the same, and show a single value that uses the same notation as the requirement specifications. You can also use the calcStatisticsForNonMC environment variable to save or load the statistical values to or from the run summary data file. In case of Monte Carlo runs, the statistical values are always non-empty.

© 2023
All Rights Reserved.

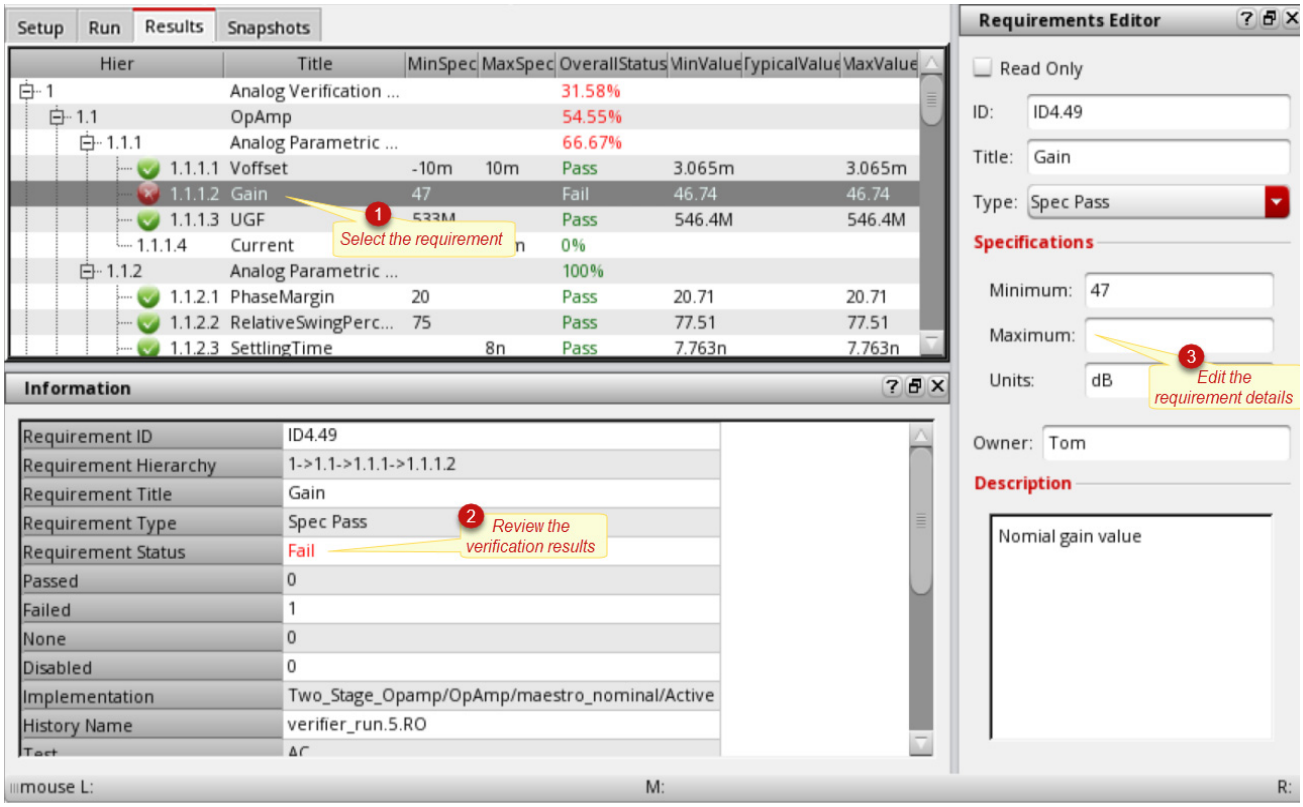
The *Results* tab also displays the detailed verification results of the selected requirement in the Information assistant, as well as the corresponding requirement details in the Requirement Editor Assistant.

You can show or hide the columns of the requirements list in the *Results* tab. For this, right-click anywhere on the column heading in the results area and select the columns you want to show. Verifier hides the columns that are not selected.

To sort columns, you can use the drag-and-drop method and change the display of columns. Additionally, you can specify the default column order and hidden columns by using the environment variables resColumnOrder and resHiddenColumns.

Viewing Verification Results in the Information Assistant and Requirements Editor

The following figure shows how you can review the verification results of a requirement in the Information assistant and Requirement Editor.



© 2023
All Rights Reserved.

To review the verification results:

➡ Select the requirement in the *Results* tab.

- ❑ The Information assistant displays the requirements hierarchy with the verification status and details such as type, status for the requirement, specifications, histories, minimum value, maximum value for the corresponding implementation. For more details, see Viewing Run Results Using the Information Assistant.
- ❑ The Requirement Editor displays the requirement currently selected in the *Results* tab. You can make required changes to your verification plan to achieve your design verification goals. These changes can be made to either one of requirements or implementations.

➡ Select a requirement to view its verification details in the *Information* assistant.

You can copy the value of a field in the assistant. For this, right-click the result field and choose *Copy String*. You can also select the field and press `Ctrl+C` to copy the text.

To view the mapped implementation, along with its specification and simulation status, select the *Details* check box.

For requirements that have passed verification, the results meet the specifications.

To filter the requirements based on overall status:

➡ Click *Show* and select the filter options as required.

For example, if you want to hide the requirements that are not mapped, ensure that *Unmapped* is not selected. The hidden requirements are indicated next to the overall progress bar.

To open the implementation cellview mapped with a requirement:

➡ Right-click the requirement and choose *Edit Implementation Cellview*.

The implementation cellview opens in edit mode in its default application.

© 2023 All Rights Reserved.

Verification Status Indicators

The following table lists the keywords and icons used to indicate the verification status of requirements. The icons display with the hierarchical numbers, and the keywords display in the *Overall Status* column.

Icon	Indicators	Description
	Pass	The requirement passed verification. **Note:** If a requirement is mapped to multiple implementation outputs, the *OverallStatus* column shows Pass as its status only when all implementation outputs mapped to it have also passed.
	Fail	The requirement failed verification. A tooltip displays the failed and canceled points for the failed tests or outputs.
	Signoff Required	The requirement is of the type *Manual* and it is not signed off.
	Not Run	The implementation mapped to the requirement has not been run and must be run to capture simulation results.
	Spec Check Fail	The specification checks failed for the requirement. For details, see the corresponding requirement icon.
	Not Mapped	The requirement of the type *Ran OK* or *Spec Pass* is not mapped to any implementation. It needs to be mapped to its corresponding implementation.
	No Results	Verifier does not have the simulation results of the implementations associated with the requirement.
	Test Disabled	The test mapped to the requirement is disabled in the implementation cellview or the specified history, and cannot be run.

Requirements Signoff in Virtuoso ADE Verifier

Verifier lets you sign off requirements of type *Manual* and requirements that have overall status of `Fail`, `Not Mapped`, `Spec Check Fail`, or `Signed Off`. Verifier considers signed off requirements as requirements that passed verification. Therefore, when you sign off a requirement, the overall verification progress percentage increases. However, requirements can be signed off only if the corresponding simulation points are covered in verification.

© 2023
All Rights Reserved.

Requirements mapped to implementations that have uncovered points are not signed off. For details, see Coverage Calculation in Virtuoso ADE Verifier.

The procedure to sign off the failed, unmapped, or spec check failed requirements is similar to signing off a requirement of the type *Manual*. After signing off a requirement, select it in the requirements hierarchy of the *Results* tab and review the signoff details in the *Information* assistant.

You can specify the validity period of a requirement signoff. For example, you can choose to retain the signoff indefinitely or for a specific period.

The signed off requirements are documented in the verification reports. For details on these reports, see Detailed Verification Report.



When you change the mapping of a signed off requirement, the signoff gets removed because it loses relevance with the new map.

Signing off Requirements

To sign off a failed requirement:

1. In the *Results* tab, right-click the failed requirement and select *Signoff*.

   The Sign Off Form form displays.

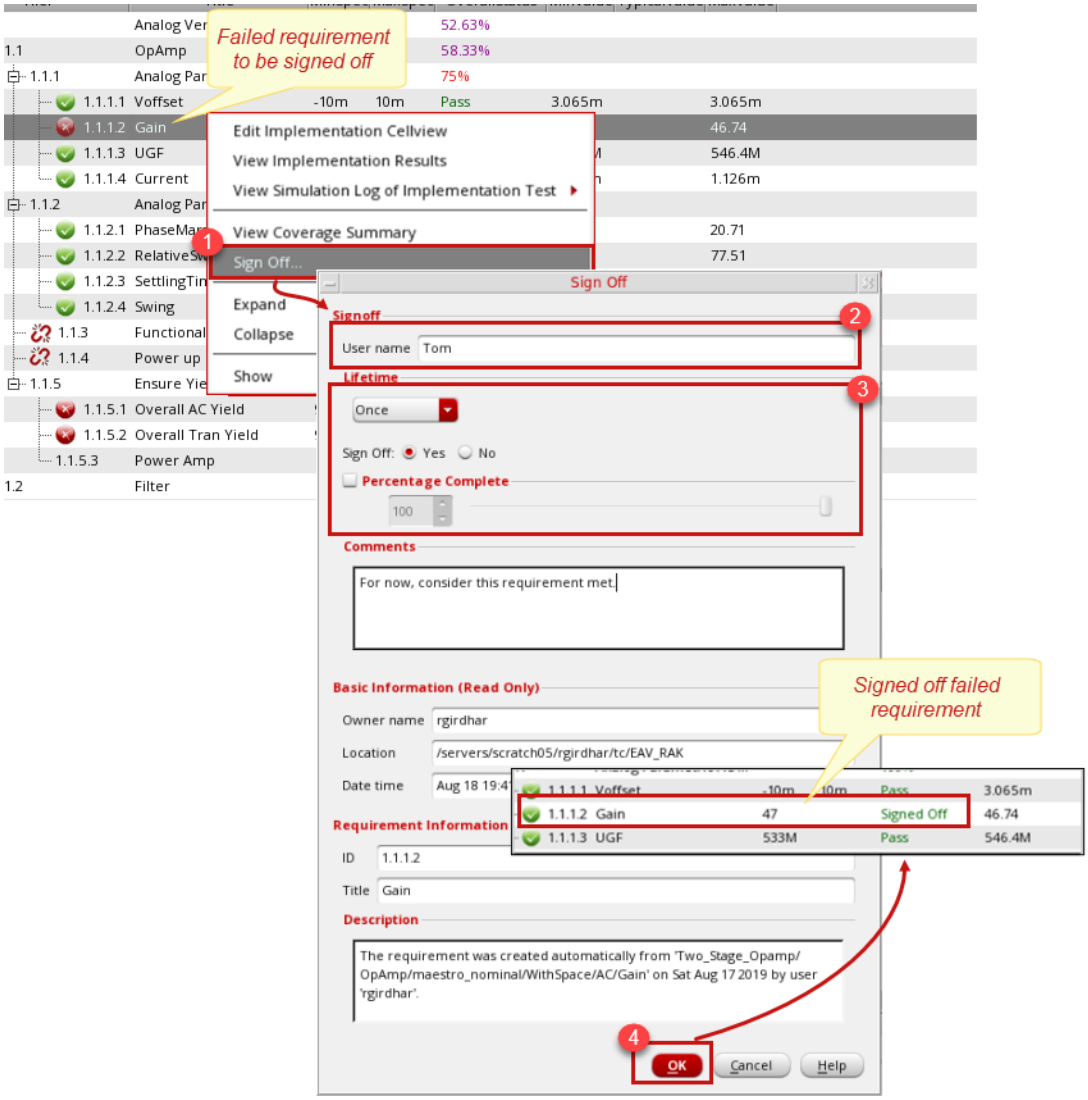
2. Type a name in the *User Name* field.
3. Specify the valid period of the signoff from the *Lifetime of Signoff* drop-down list. The available options are:
   - *Once*: The signoff is valid only for a specific set of simulation results and will become disabled when newer simulation results become available.
   - *Indefinitely*: The signoff is valid for an infinite time.
   - *Until Date*: The signoff is valid up to the date specified in the corresponding field.
4. Specify if sign off information is stored after applying, or without applying the signoff to the requirement.
5. Type details of the signoff, such as the reason, in the *Comments* field.

© 2023 All Rights Reserved.

6. Click *OK*.

The status of the requirement changes to *Signed Off*.

The following figure illustrates how you can sign off a failed requirement.



Additional Information

- In a multi-user verification setup, only the requirement owners can sign off their requirements in the owner cellviews. The project manager can review the sign-off details from the master cellview, but cannot sign off the requirements. See Synchronization Between the Master and Referenced Cellviews.

© 2023
All Rights Reserved.

- Specifying the sign off lifetime as `Once` for a requirement essentially means that the signoff is valid only for the current simulation results. However, an unmapped requirement does not generate results. Therefore, the requirement with sign off set to `Once` may continue to remain signed off unless it is mapped to an implementation.
- When you copy or reference requirements from another Verifier cellview, these requirements include the signoff details that were added in the previous Verifier cellview. These details are also displayed in the Information assistant. You can only modify the signoff for copied requirements. For referenced requirements, the signoff details cannot be modified or deleted.'

Updating the Completion Progress of Requirements Manually

In the Sign Off form, you can add or update the completion percentage for requirements of type *Manual*. This indicates that verification is in progress, but not complete.

To add the completion percentage for a *Manual* requirement:

1. Select the *Percentage Complete* check box.

   This enables a list box and a slider.

2. Do one of the following:

   - In the list box, select the percentage of completion progress that you want to specify.
   - Move the slider to select the completion percentage that you want to specify.

The specified percentage is used to calculate the status of *Manual* requirements and also impacts the status of *Note* requirements. Both these values are shown in the *OverallStatus* column. For example, if you specify the completion percentage as `50%`, the requirement shows in the *Results* tab with `50%` value for *OverallStatus*. If you specify the value as `100%` then the requirement status shows *Signed Off*, and if you specify `0%`, the status shows as *Fail*.

You can select either *Sign Off* or *Percentage Complete* to specify the sign off for your requirements. When you select the *Percentage Complete* check box, the *Sign Off* radio options are disabled. A percentage value less than `100%` is still considered as a `Fail` for the overall status of the requirements. If you move the percentage to `100%`, the requirement status automatically shows as `Pass`.

© 2023 All Rights Reserved.

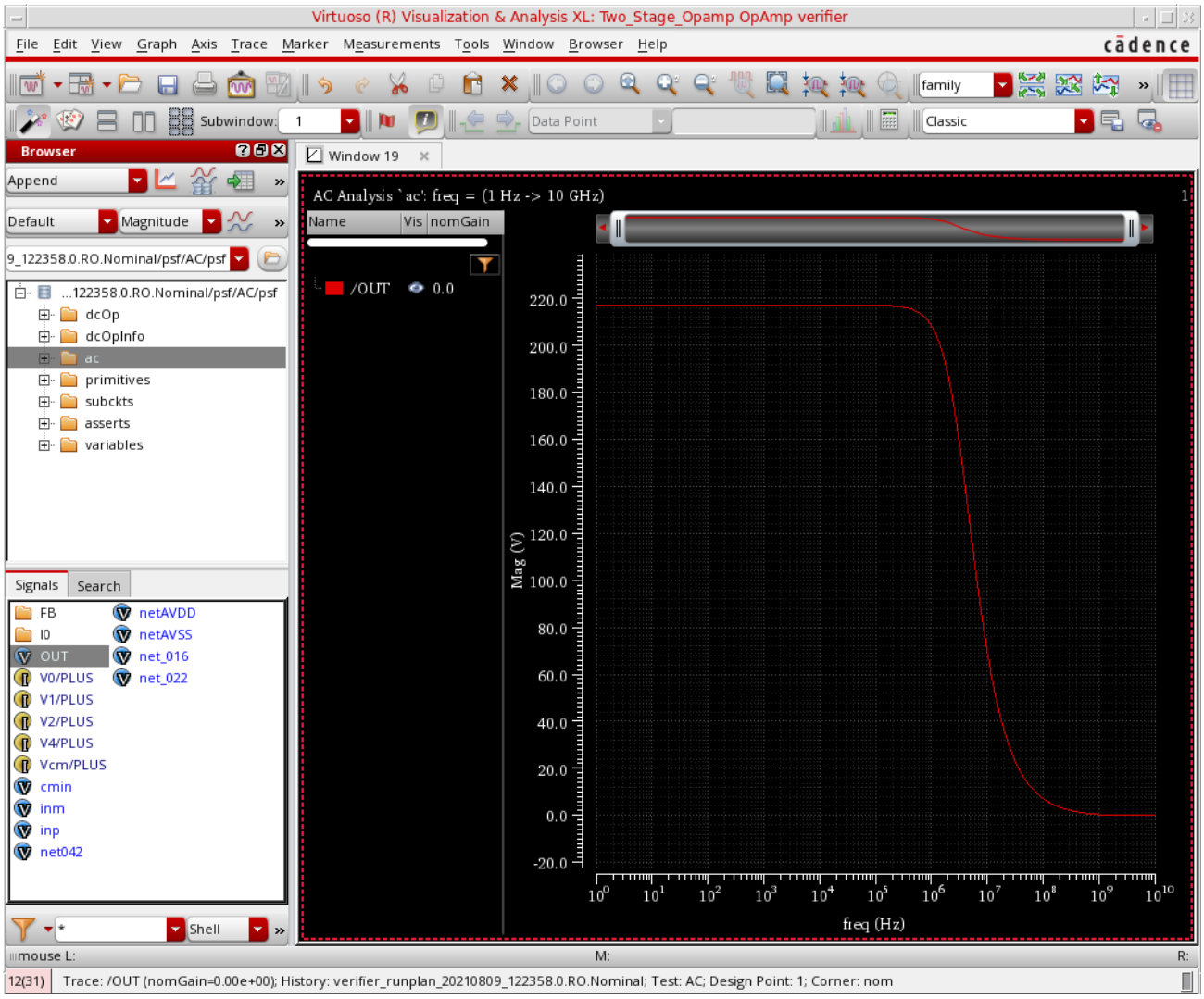
Viewing Plotted Results

The *Results* tab in Verifier helps you to view the results of your verification setup. Additionally you can also view the results with their waveforms in the Virtuoso Visualization & Analysis window when the *Save Simulation Data* option is set to *All* in the Save Options form.

To view the results in the Virtuoso Visualization & Analysis window:

1. Select a requirement from the *Results* page.
2. Choose *Open Results Browser* from the shortcut menu.

The Virtuoso Visualization & Analysis window is displayed with the results.



© 2023
All Rights Reserved.

The Virtuoso Visualization & Analysis window title displays the name of the Verifier cellview. If the implementation has waveform data, the loaded results are displayed in the form of a waveform, and the Browser assistant in the window displays the results directory path in the following format:

RunDir/psf/*list of tests*/psf

If the implementation has no waveform data, only the results information is displayed in the Virtuoso Visualization & Analysis window. For Monte Carlo simulations, the results might not be displayed because by default the *Save Simulation Data* option is disabled.

Detailed Verification Report

Verifier creates HTML reports that contain the overall verification status and the verification details of each requirement, including the mapped implementations and verification results. These reports are available in the following formats:

- Basic Format
- Interactive Format

You can only publish a report in one of these formats at once. By default, Verifier uses the default web browser to display the HTML report in basic format. You can specify the required settings in the *Reporting* tab of the Preferences form.

Related Topics

HTML Reports in Basic Format

HTML Reports in Interactive Format

HTML Reports in Basic Format

The HTML report in basic format lets you view the detailed verification status by including the overall verification and run status, requirement hierarchy with links and status, and the detailed verification status table of each requirement.

Verifier stores this report in the current directory using the naming format *cell*.html. For example, verification.html. If required, you can specify a custom name and location for the report.

© 2023 All Rights Reserved.

Important

You can use the verification report to monitor the implementation run status and verification status when a batch script is running. The report updates after each implementation cellview run is completed to let you monitor the progress. For details, see Scheduling Runs.

The verification report in basic format includes the following elements:

- **Report header**

  The header information includes the report filename, location, creation time, and name of the creator. It also includes the overall verification status.

- **Run information**

  The run information includes the implementation cellviews and their simulation details.

- **Requirements-based verification details**

  These verification details include the requirements, their verification status, and the implementation details with status.

Viewing the HTML Report in Basic Format

To view the verification report:

1. In the CIW, set the following environment variable.

   ```
   envSetVal("verifier.html" "useTemplate" 'boolean nil)
   ```

2. Choose *Tools – Publish HTML Report*.

© 2023 All Rights Reserved.

The report is displayed in the browser window.

TestVerification Report

Report created on 27 Aug 2018 21:47:42 by [redacted] in '/servers/scratch05/[redacted]/tc/ADE/EAV_RAK/EAV_RAK' using sub-version IC6.1.8-64b[redacted]

Overall Status

Passed (3/4); Failed (1/4)

75%

Run Information

Implementation	History Item	Status	Total Points	Failed Points	Duration	Start	Finish	User	IC Version	IC Installation	Directory
Two_Stage_Opamp/OpAmp /maestro_adexl/Active	verifier_run.7.RO	Run Out of Date	1	0	26	Apr 17 20:25:15 2018	Apr 17 20:25:41 2018	rgirdhar	@(#)$CDS: virtuoso version 6.1.8-64b 04/16/2018 08:45 (ip-172-18-22-100) $	/servers /inciccm_v31 /ciccm_dev /IC6.1.8 /EA4/lnx86 /64/180416-127 /tools.lnx86/dfII	/servers/scratch05 /rgirdhar/tc/ADE /EAV_RAK /EAV_RAK /simulation /Two_Stage_Opamp /OpAmp /maestro_adexl /results/maestro /verifier_run.7.RO

Two_Stage_Opamp/OpAmp/maestro_adexl/Active (36.36%)

Implementation	Test	Output	Mapped
Two_Stage_Opamp/OpAmp/maestro_adexl/Active (36.36%)	AC (40%)	Op_Region	No
		Current	07a8f2d8-82a1-4430-b428-07b76a3fea0a
		UGF	No
		Gain	No
		Voffset	1222e921-cf48-4f1d-8518-fadc1c4534f8
	TRAN (50%)	Swing	e1d77adb-09db-409a-9389-97122522a884
		SettlingTime	No
		RelativeSwingPercent	No
		PhaseMargin	019f2fe5-192b-41e9-94fe-efda644bbf38

Result Information

1 TestRequirements (75%)
- 1.1 AMP (100%)
  - 1.1.1 AMP1 (Pass)
- 1.2 PLL (50%)
  - 1.2.1 PLL1 (Pass)
  - 1.2.2 PLL2 (Signoff Required)
- 1.3 Yield (100%)
  - 1.3.1 Yield1 (Pass)

1 TestRequirements

Back to Table of Contents

Goal Type: Note, Summary: 75% Pass Rate, 4 Requirements: 1 Failed, 3 Passed, 0 No Results, 0 Unmapped,

© 2023 All Rights Reserved.

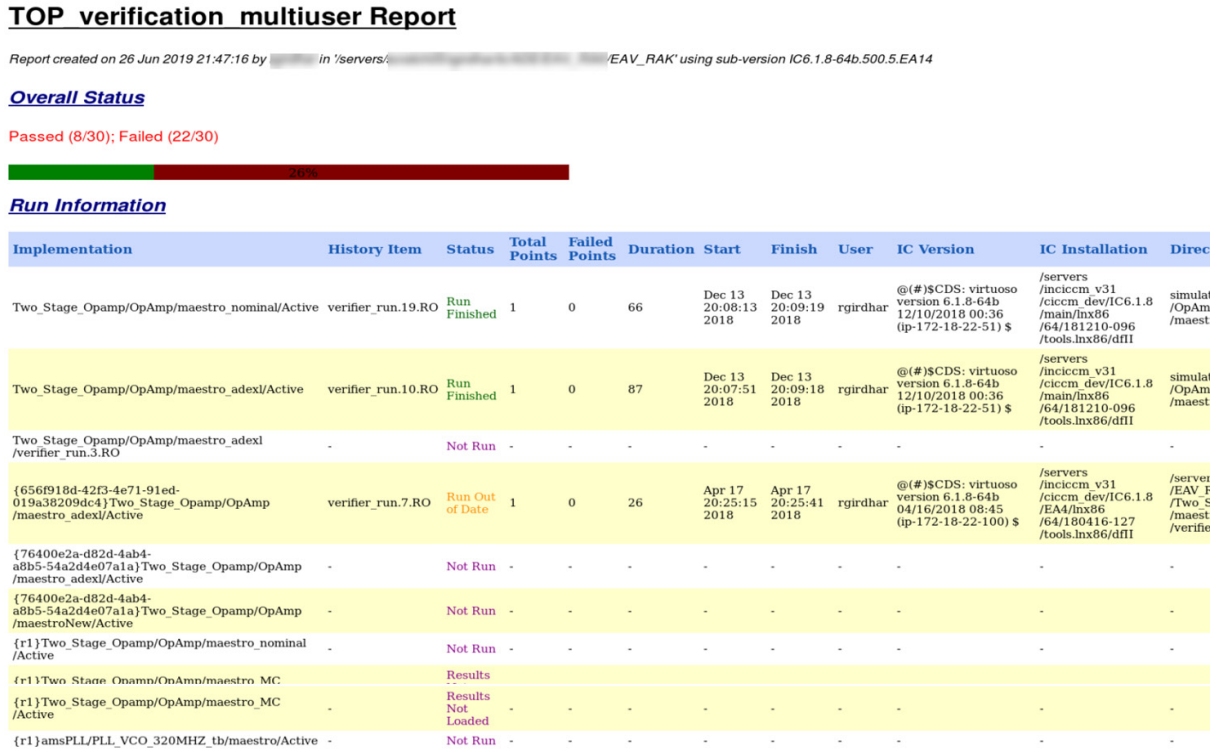
Typically, the requirement IDs are hidden in the HTML report. To display the requirement ID as shown in the following illustration, set the environment variable includeRequirementId to `t`.



Result Information

1[4fe83a9b-a306-4452-b548-5ad18c785106] -- Two_Stage_Opamp OpAmp maestro_MC Active (Note) (98%)
1.1[633d530f-d1f5-4bd6-8bb7-69f91a85afe5] -- Two_Stage_Opamp OpAmp maestro_MC Active (Pass)
1.2[82fa51c1-83c6-47a0-a58c-c4d496e625d6] -- AC (Note) (100%)
1.2.1[f0f0a1c4-7d12-48cf-a810-50e059b720d1] -- AC (Pass)
1.2.2[10c14f18-8ae4-4509-a7dd-08d8483e87c8] -- Op_Region (Pass)
1.2.3[1d1393a6-271d-47ee-a395-302f476c746c] -- Current (Pass)
1.2.4[5714746f-c38a-4b40-a15d-4eecb6f90e9a] -- UGF (Pass)
1.2.5[fc830754-613e-49af-bc6b-718073bd7da9] -- Gain (Pass)
1.2.6[1584528a-5275-479b-a5b9-81a0192c5fd7] -- Voffset (Pass)
1.2.7[5f31bd99-cc85-4cc2-a2e3-9908aafe50b3] -- AC::Overall_Yield (Pass)
1.2.8[ca3106eb-b2ae-4267-b4c5-98fefd33536f] -- Op_Region::Yield (Pass)
1.2.9[09bfcb8b-57c3-4716-9a51-e3b52e85fdd2] -- Op_Region::Mean (Pass)
1.2.10[147a6d37-9d5d-4f7f-ad10-56124aa51c2d] -- Op_Region::StandardDeviation (Pass)
1.2.11[779b41af-744f-4612-b05b-e631fa2138d8] -- Op_Region::Cpk (Pass)
1.2.12[c096c1d9-a467-4dfd-a556-47e06471845a] -- Current::Yield (Pass)
1.2.13[944588fa-745c-416a-b4c0-b41bf748974f] -- Current::Mean (Pass)
1.2.14[5ec3eb0d-18c3-417b-ae86-4e748511a9f0] -- Current::StandardDeviation (Pass)
1.2.15[78321174-daf7-4184-bf76-8a0be319c044] -- Current::Cpk (Pass)
1.2.16[99b85910-f650-4a92-a0cb-b947c8a69efc] -- UGF::Yield (Pass)
1.2.17[fe786f73-ec9c-41f1-a092-d142c2c2add5] -- UGF::Mean (Pass)
1.2.18[e9ce4053-695a-4462-abf7-db71f0f4137d] -- UGF::StandardDeviation (Pass)
1.2.19[5ef0e4f5-50b8-439c-b487-c4616c5716a9] -- UGF::Cpk (Pass)
1.2.20[fa8febb7-46c9-4bd0-9f53-264c4f3a6687] -- Gain::Yield (Pass)
1.2.21[df7ccc6b-c311-46aa-9b98-5294624fca6e] -- Gain::Mean (Pass)
1.2.22[304b66f2-6a83-4237-805b-64ec51b6d657] -- Gain::StandardDeviation (Pass)
1.2.23[12b55df1-badc-49ca-97ad-90dfea875767] -- Gain::Cpk (Pass)
1.2.24[e659b9c6-1b56-4554-86b3-a7fa6c6ab7f0] -- Voffset::Yield (Pass)
1.2.25[06d212d7-d4c3-4623-b6f9-6d2c6cb93a7f] -- Voffset::Mean (Pass)
1.2.26[55dbc3c3-b635-493d-b729-2aa41f0b1a24] -- Voffset::StandardDeviation (Pass)
1.2.27[da3cedc5-18cf-451b-942d-66303dd5d0c0] -- Voffset::Cpk (Pass)
1.3[ee7e5d51-8ef2-46f2-aa76-1ab6539568cf] -- TRAN (Note) (95.45%)

In a Verifier setup with referenced cellviews, the default behavior of the basic HTML report is to hide the ExtRef ID from the beginning of referenced implementation LCVH. You can display the ExtRef ID as shown in the following illustration by setting the environment variable includeReferenceIDPrefix to `t`.



TOP_verification_multiuser Report

Report created on 26 Jun 2019 21:47:16 by [redacted] in '/servers/[redacted]/EAV_RAK' using sub-version IC6.1.8-64b.500.5.EA14

Overall Status

Passed (8/30); Failed (22/30)

26%

Run Information

Implementation	History Item	Status	Total Points	Failed Points	Duration	Start	Finish	User	IC Version	IC Installation	Direc
Two_Stage_Opamp/OpAmp/maestro_nominal/Active	verifier_run.19.RO	Run Finished	1	0	66	Dec 13 20:08:13 2018	Dec 13 20:09:19 2018	rgirdhar	@(#)$CDS: virtuoso version 6.1.8-64b 12/10/2018 00:36 (ip-172-18-22-51) $	/servers /incicm_v31 /ciccm_dev/IC6.1.8 /main/lnx86 /64/181210-096 /tools.lnx86/dfII	simulat /OpAm /maest
Two_Stage_Opamp/OpAmp/maestro_adexl/Active	verifier_run.10.RO	Run Finished	1	0	87	Dec 13 20:07:51 2018	Dec 13 20:09:18 2018	rgirdhar	@(#)$CDS: virtuoso version 6.1.8-64b 12/10/2018 00:36 (ip-172-18-22-51) $	/servers /incicm_v31 /ciccm_dev/IC6.1.8 /main/lnx86 /64/181210-096 /tools.lnx86/dfII	simulat /OpAm /maest
Two_Stage_Opamp/OpAmp/maestro_adexl /verifier_run.3.RO	-	Not Run	-	-	-	-	-	-	-	-	-
{656f918d-42f3-4e71-91ed-019a38209dc4}Two_Stage_Opamp/OpAmp /maestro_adexl/Active	verifier_run.7.RO	Run Out of Date	1	0	26	Apr 17 20:25:15 2018	Apr 17 20:25:41 2018	rgirdhar	@(#)$CDS: virtuoso version 6.1.8-64b 04/16/2018 08:45 (ip-172-18-22-100) $	/servers /incicm_v31 /ciccm_dev/IC6.1.8 /EA4/lnx86 /64/180416-127 /tools.lnx86/dfII	/server /EAV_F /Two_S /maest /verifi
{76400e2a-d82d-4ab4-a8b5-54a2d4e07a1a}Two_Stage_Opamp/OpAmp /maestro_adexl/Active	-	Not Run	-	-	-	-	-	-	-	-	-
{76400e2a-d82d-4ab4-a8b5-54a2d4e07a1a}Two_Stage_Opamp/OpAmp /maestroNew/Active	-	Not Run	-	-	-	-	-	-	-	-	-
{r1}Two_Stage_Opamp/OpAmp/maestro_nominal /Active	-	Not Run	-	-	-	-	-	-	-	-	-
{r1}Two_Stage_Opamp/OpAmp/maestro_MC		Results									
{r1}Two_Stage_Opamp/OpAmp/maestro_MC /Active	-	Results Not Loaded	-	-	-	-	-	-	-	-	-
{r1}amsPLL/PLL_VCO_320MHZ_tb/maestro/Active	-	Not Run	-	-	-	-	-	-	-	-	-

© 2023 All Rights Reserved.

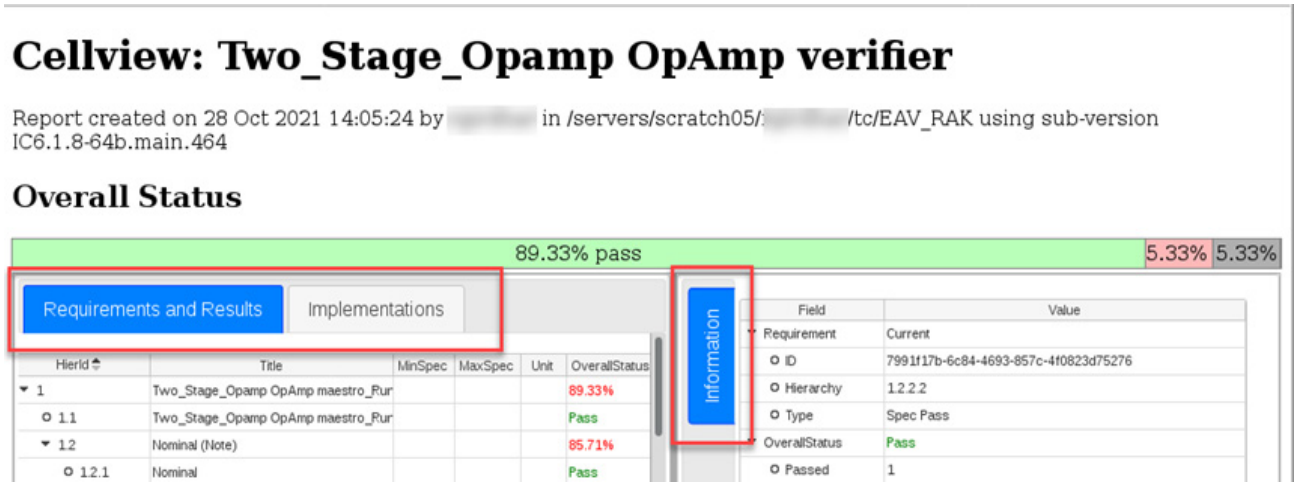
Related Topics

Detailed Verification Report

HTML Reports in Interactive Format

HTML Reports in Interactive Format

You can use the interactive format to display more structured HTML reports. In this format, the report contains two panes. The left pane displays two tabs, *Requirements and Results* and *Implementations*. When you click a requirement or implementation on the left pane, its corresponding verification information is displayed in the vertically placed *Information* tab on the right.



Important

Cadence recommends using a browser with JavaScript support to view reports in interactive format.

The verification report in interactive HTML layout includes the following elements:

- **Report header**

  The header information includes the report filename, location, creation time, and name of the creator. It also includes the overall verification status.

- **Requirements, Results, and Implementations**

  The left pane of the verification report displays the setup details in two tabs. The *Requirements and Results* tab in the report displays a mix of requirements and results

© 2023 All Rights Reserved.

that are available on the *Requirements* pane of the *Setup* tab and the *Results* tab. The *Implementations* tab in the report displays the implementations with their test and outputs that are available on the *Setup* tab.

- **Detailed Information**

  The *Information* tab on the right displays all the details of the selected item from the requirements, implementation cellviews and their simulation details, similar to the Information assistant.

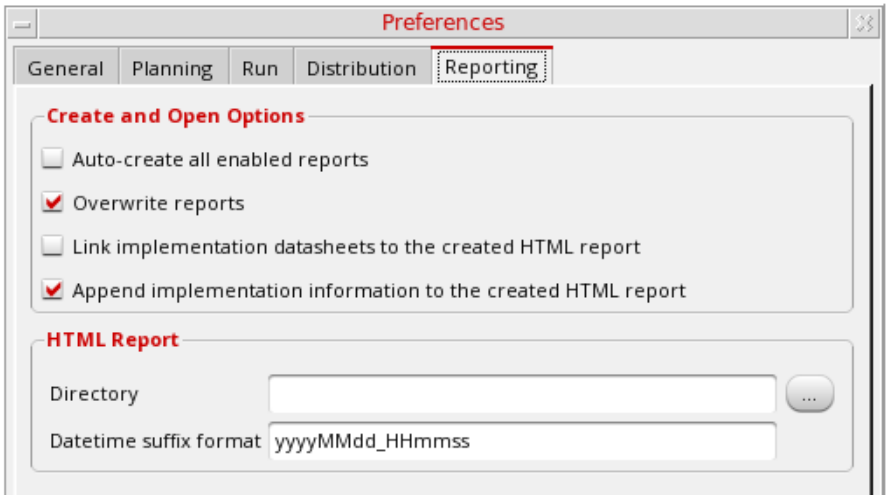
Viewing HTML Reports in Interactive Format

To view the verification report:

1. In the CIW, set the following environment variable.

   ```
   envSetVal("verifier.html" "useTemplate" 'boolean t)
   ```

   Setting this variable to `t` updates the *Reporting* preferences as follows:



2. Specify a directory name for the HTML report in the *Directory* field.

   If the specified directory does not exist, it is created automatically.

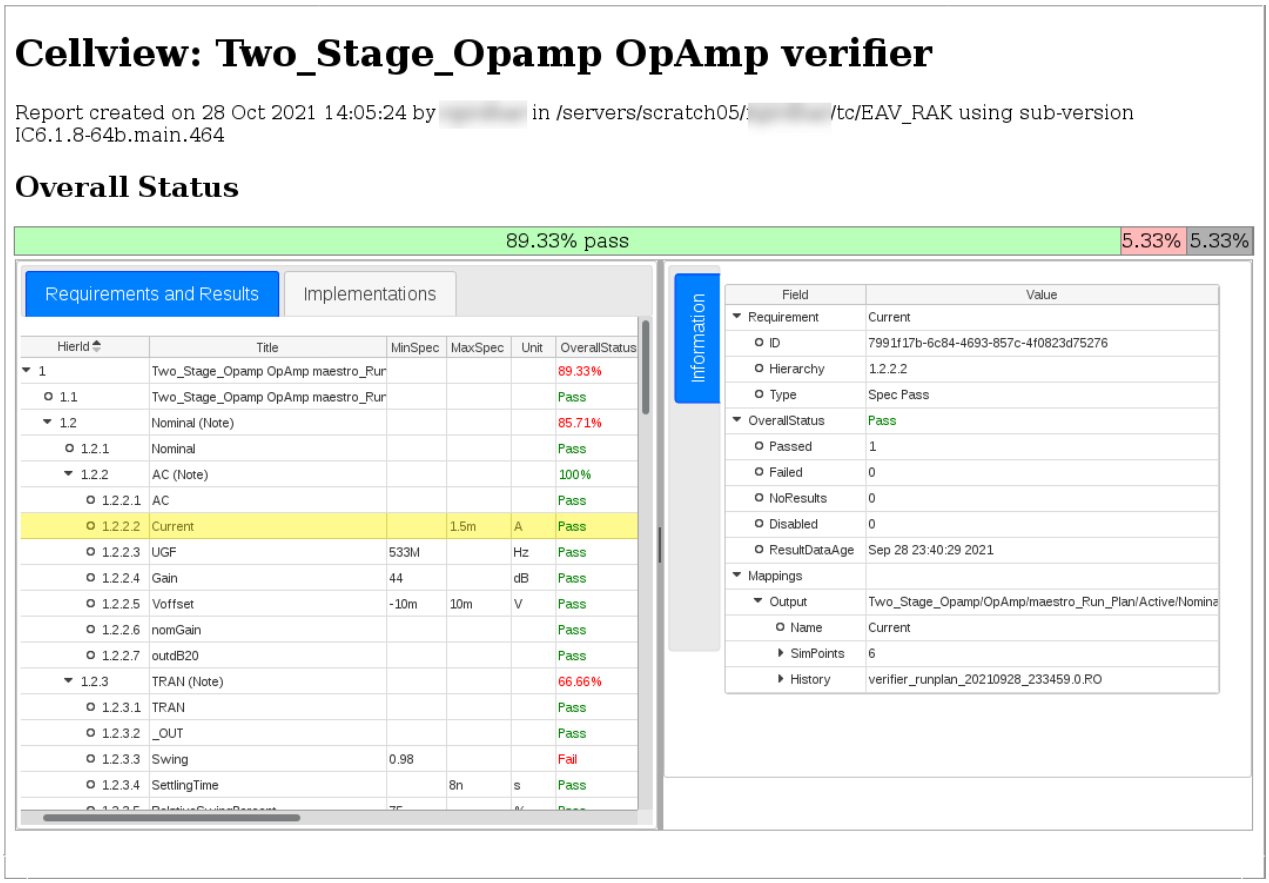
3. Specify a datetime suffix format in the *Datetime suffix format* field.

© 2023
All Rights Reserved.

The default format is `yyyyMMdd_HHmmss`.

4. Choose *Tools – Publish HTML Report*.

The report is displayed in the browser window.



Viewing Additional Columns in the Verification Report

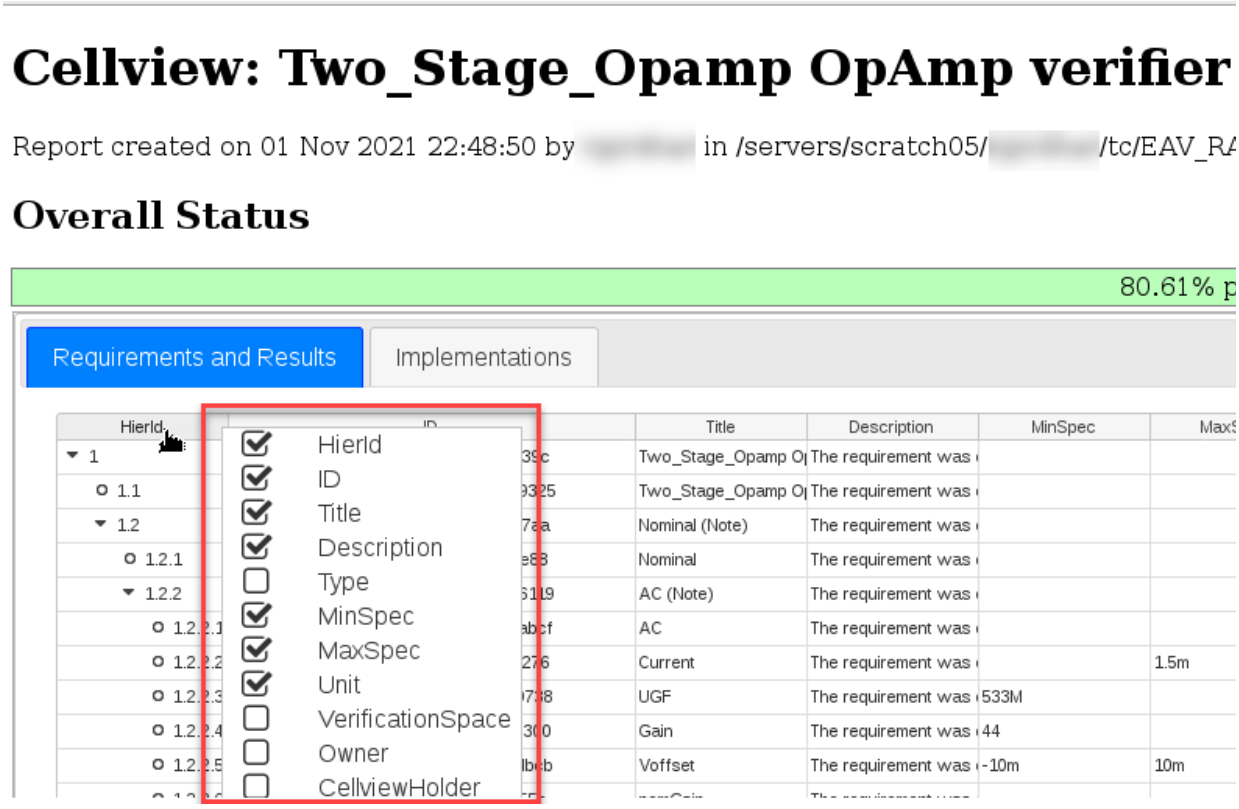
By default, the HTML report displays the `HierID`, `Title`, `Description`, `MinSpec`, `MaxSpec`, `Unit`, `OverallStatus`, `MinValue`, `TypicalValue`, `MaxValue`, and `Coverage`.

To view additional columns in the HTML report:

1. Right-click anywhere on the column header area in the report.

© 2023 All Rights Reserved.

A checklist is displayed with all columns. The visible columns are selected and the additional columns are unselected.



2. Select the required columns from the checklist.

The additional columns are displayed.

You can sort the displayed columns by clicking the required column header. Additionally, you can drag a column header to the another column to reorder the displayed columns.

Related Topics

Detailed Verification Report

HTML Reports in Basic Format

Customization of HTML Reports in Interactive Format

Reporting

© 2023
All Rights Reserved.

Customization of HTML Reports in Interactive Format

When you set the `useTemplate` environment variable to `t`, and run the *Tools – Publish HTML* command, Verifier generates a template-based interactive report by performing the following operations in the given order:

1. Creates a report directory where the published HTML report is saved. The directory name and location is determined by the *Directory* and *Datetime suffix format* fields in the *Reporting* tab of the Preferences form.

   If the specified directory does not exist, it is created automatically.

2. Exports a `.json` file for the session into the newly created report directory. For example, *`HTML_dir`*`/export.json`.

3. Saves ADE Assembler datasheets into the same report directory. This is optional.

   Use the *Create ADE Assembler datasheet at run or reload time* option to enable or disable the creation of these datasheets.

4. Copies any documents or images from the Verifier session into the report directory.

5. Copies and extracts the data from `/dfII/verifier/templates/` located by the Cadence Search Framework (CSF) into the report directory.

   This extracted data is customizable. The default data and any additional information, such as the `stylesheet.css` or JavaScript used by the template, is available within *`install-path`*`/share/cdssetup/dfII/verifier/templates`.

   If you have saved customized templates in a custom directory such as `.cadence/dfII/verifier/templates`, the data from this directory is also copied into the report directory.

6. Runs the Python script file `publishHtml.py`, which is available in `/dfII/verifier/scripts/` located by CSF. The `publishHtml.py` file is created using the Jinja scripting language.

   You can customize a copy of the script by saving the script file `publishHtml.py` at a location such as `.cadence/dfII/verifier/scripts/publishHtml.py`.

   The `publishHtml.py` script uses the *`HTML_dir`*`/export.json` and `/dfII/verifier/templates/`*`templateName`* identified by CSF. Here, the value of <*`templateName`*> is set using following environment variable.

   ```
   envSetVal("verifier.html" "templateName" 'string "index.html")
   ```

7. Applies the template defined by the script and uses the `export.json` to expand the contents into the final output *`HTML_dir`*`/index.html`.

© 2023 All Rights Reserved.

You can modify the HTML report name, `index.html`, by setting the `templateName` environment variable with the new filename. The specified filename should exist in the template directory.

The architecture provided by the Jinja template language is highly flexible and can let you export your report to multiple formats, such as `.txt`, `.csv`, `.pdf`, and more. For example, the template contains the following source code to display the report header:

```
<h1>Cellview: {{ data["cellview"][0] }} {{ data["cellview"][1] }}
{{data["cellview"][2]}}</h1>
<p>Report created on {{ data["export"]["date"] }} by {{ data["export"]["username"]
}} in {{ data["export"]["cwd"] }} using {{ data["export"]["subVersion"] }}</p>
```

Here, {{}} indicates expressions defined in the Jinja template language and `data` indicates the contents of the `export.json` file. After the template is rendered, the final HTML output contains the following details:

```
<h1>Cellview: opamp090 full_diff_opamp_AC verification1</h1>
<p>Report created on 13 Oct 2021 13:29:16 by Tom in /projects/sco_ams/users/Tom/
work/adexl/designs/verifier/V3_Examples using sub-version  IC6.1.8-xxx.xxxx.xxx </
p>
```

The `export.json` file contains all your verification data and, with advanced usage of the template, can be used to display in-depth and highly accurate information in the final HTML report.

Related Topics

Detailed Verification Report

HTML Reports in Interactive Format

Customizing the Interactive HTML Report Template

To customize the template of your interactive HTML report:

1. Type the following commands in the terminal window:

   ```
   cp -r $CDSHOME/share/cdssetup/dfII/verifier/templates ./.cadence/dfII/
   verifier
   cp -r $CDSHOME/share/cdssetup/dfII/verifier/scripts ./.cadence/dfII/verifier
   ```

2. Create a copy of the default template file, `index.html`, and save it as `myIndex.html` in `./.cadence/dfII/verifier`.

3. Set the following variables in the Virtuoso CIW:

   ```
   envSetVal("verifier.html" "useTemplate" 'boolean t)
   envSetVal("verifier.html" "templateName" 'string "myIndex.html")
   ```


© 2023 All Rights Reserved.

4. Open and edit the `myIndex.html` file.

The interactive HTML report is customized, as required.

Related Topics

Detailed Verification Report

HTML Reports in Interactive Format

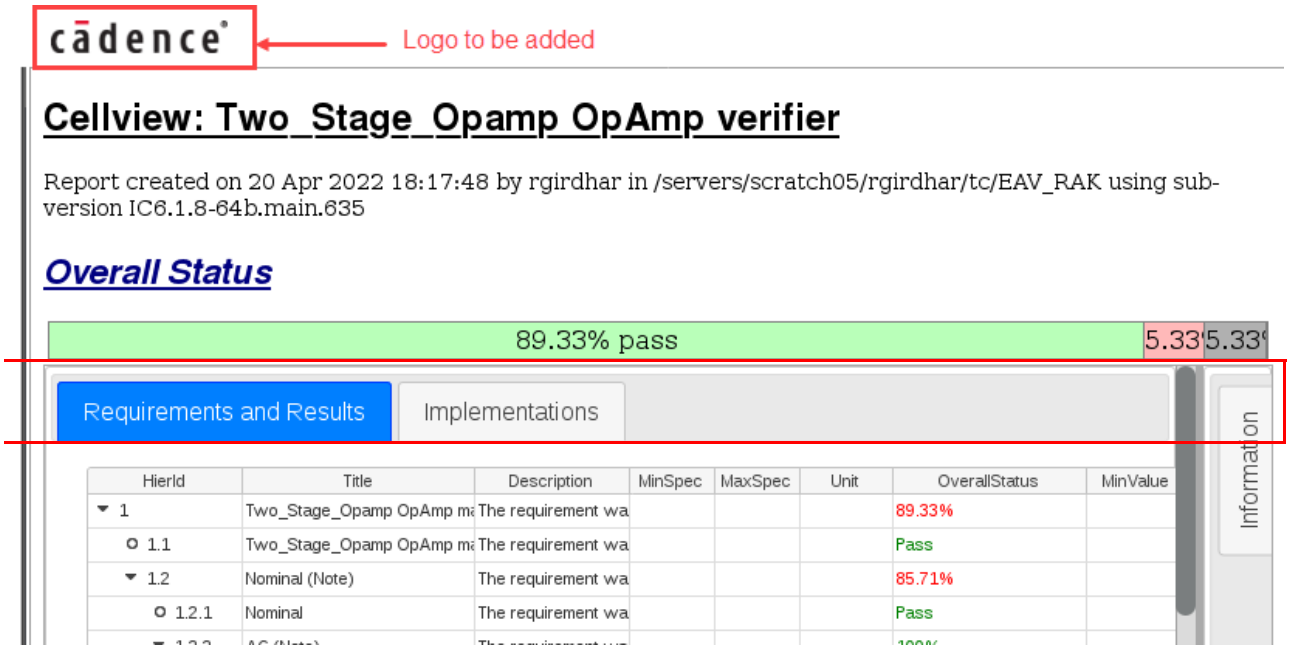
Customization of HTML Reports in Interactive Format

Examples of Interactive HTML Report Customization

This section provides some examples of how you can customize your interactive HTML reports.

Example: How to add the Cadence logo above the header of an interactive HTML report

Consider that you want to add the Cadence logo to the HTML report by modifying the report template written in the Jinja language.



© 2023
All Rights Reserved.

Jinja is a mix of the target language and some extra entries for data processing. Here, the target language in use is HTML.

```
<!DOCTYPE html>
<html>
...
<body>
<img src="https://www.cadence.com/content/dam/cadence-www/global/en_US/images/
site-images/icons/navigation-icons/cadence-logo-black.png" alt="Home" title=""
class="cq-image-placeholder">
        <h1>Cellview: {{ data["cellview"][0] }} {{ data["cellview"][1] }}
{{data["cellview"][2]}}</h1>
        <p>Report created on {{ data["export"]["date"] }} by {{
data["export"]["username"] }} in {{ data["export"]["cwd"] }} using {{
data["export"]["subVersion"] }}</p>
```

In the above example, the code enclosed within the `<img src>` tag adds the Cadence logo to the HTML report.

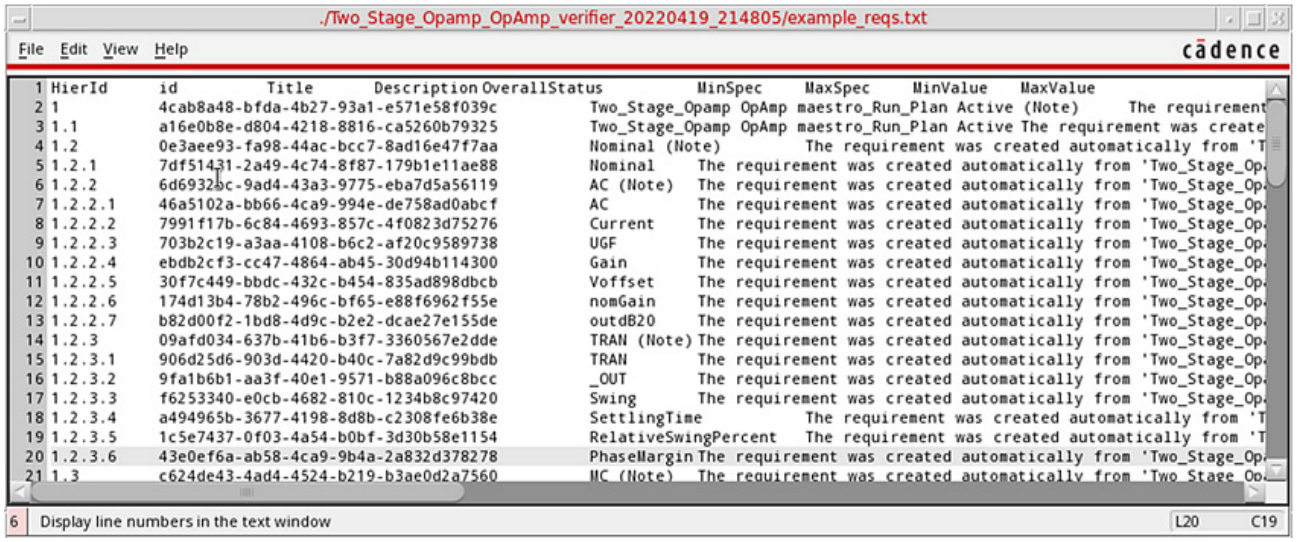
Example: How to use the export functionality to create alternative outputs

The template directory, *install-path*`/share/cdssetup/dfII/verifier/templates`, contains a few examples of alternative export formats, such as the `example_reqs.txt` and `example_reqs.json` template files.

Specify the following in the Virtuoso CIW:

```
envSetVal("verifier.html" "templateName" 'string "example_reqs.txt")
```

Now, when you create the report using the *Publish HTML Report* command, instead of an HTML report, the report is created in `text` format.



© 2023 All Rights Reserved.

Similarly, if you set the `templateName` environment variable to `example_reqs.json`, the report is created in `json` format.



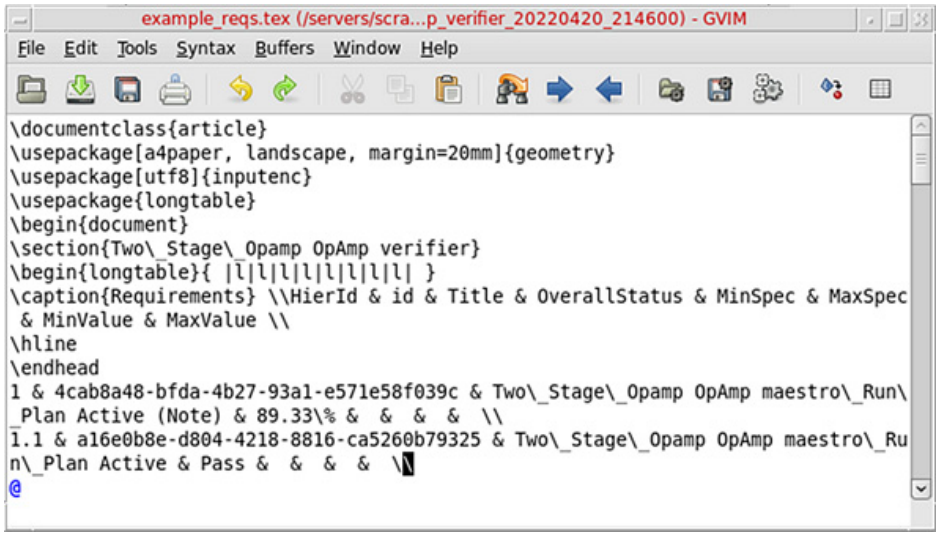
Example: How to use the template to create custom template and process languages

Consider that you want to create a PDF report. The creation of PDF using the HTML template involves a two-step process.

The following CIW command exports the template in the intermediary LaTex format.

```
envSetVal("verifier.html" "templateName" 'string "example_reqs.tex")
```

Clicking the *Publish HTML Report* command after this displays the LaTex report.

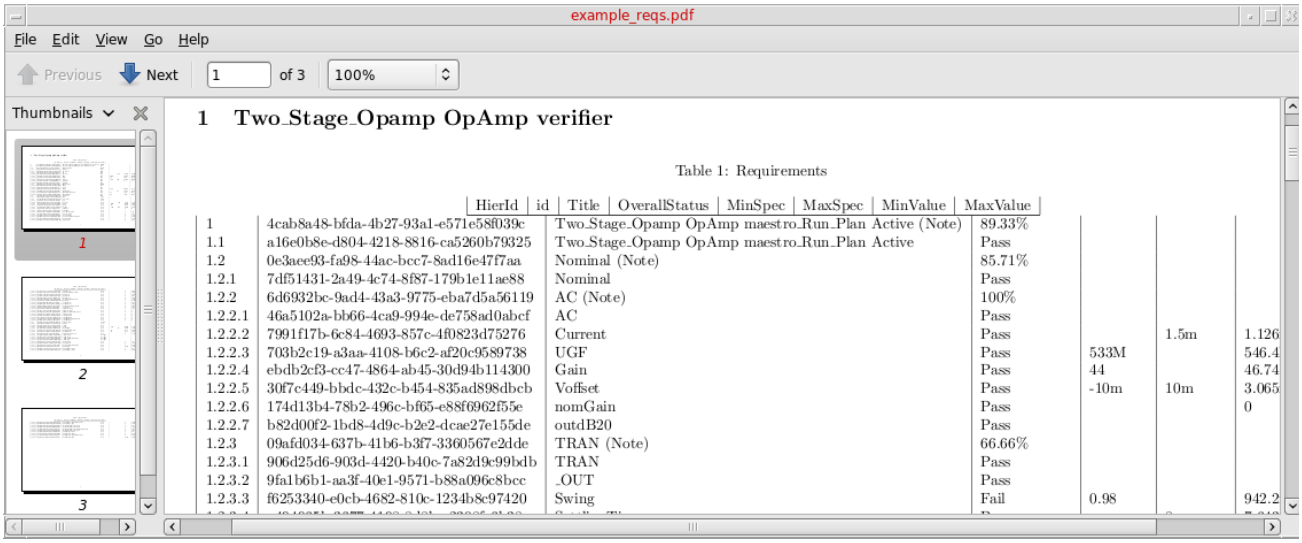


© 2023 All Rights Reserved.

Now, you can specify the following command in the CIW to convert the LaTex report to PDF format.

```
envSetVal("verifier.html" "templateName" 'string "example_reqs.tex pdf")
```

This will save the report in the PDF format.



You can check the `./scripts/publishHTML.py` file to observe the changes made by the conversion.

```
if os.path.exists(args.output):
                if is_latex and args.convert == "pdf":
                    subprocess.call(['pdflatex', '-interaction', 'nonstopmode', '-output-directory',
                                     os.path.dirname(args.output), args.output])
                else:
```

Similarly, you can create your custom report template using the Jinja template examples and change the environment variable settings accordingly.

© 2023 All Rights Reserved.

Example: Post-processing of the custom template

Consider that you have created a custom template and now you want to define the post-processing steps. For this, you can overwrite the `publishHTML.py` file as follows:

```
if os.path.exists(args.output):
        if is_latex and args.convert == "pdf":
            subprocess.call(['pdflatex', ...)
        elif args.convert == "txt2":
            with open(os.path.splitext(args.output)[0]+'.txt2', 'w') as fout:
                proc =  subprocess.call(['grep', '1.1', args.output], stdout=fout)
                return_code = proc.wait()
        else:
```

This example shows how to filter the text export using the string `1.1` and saves the results in a file with the extension `.txt2`.

© 2023 All Rights Reserved.

7

Verifying the Design Against the Specified Setup

When you manage a project, before proceeding with the sign-off for each block, you need to ensure that all the blocks successfully meet the specified setup requirements, which include variables, corners, and so on.

The Setup Library assistant helps you define a project-specific (master) setup for the design blocks in ADE Assembler and then reuse the specified setup to create or modify the current simulation setup in the Data View assistant. The setup in this assistant is generic and can be applied on any design. The Setup Library assistant can be accessed using ADE Assembler and ADE Verifier.

Following are the main tasks that you can perform using this assistant:

- Specify a master project-specific setup, which includes sweep variables, corners, model files, and so on.
- Save the specified setup in a cellview independent of the `maestro` view, and thus can be conveniently shared with designers working on different blocks.
- Create a verification space and then use these spaces to calculate the coverage percentage in ADE Verifier.

The scope and capabilities of this assistant are defined based on your work requirements and the platforms you are working on:

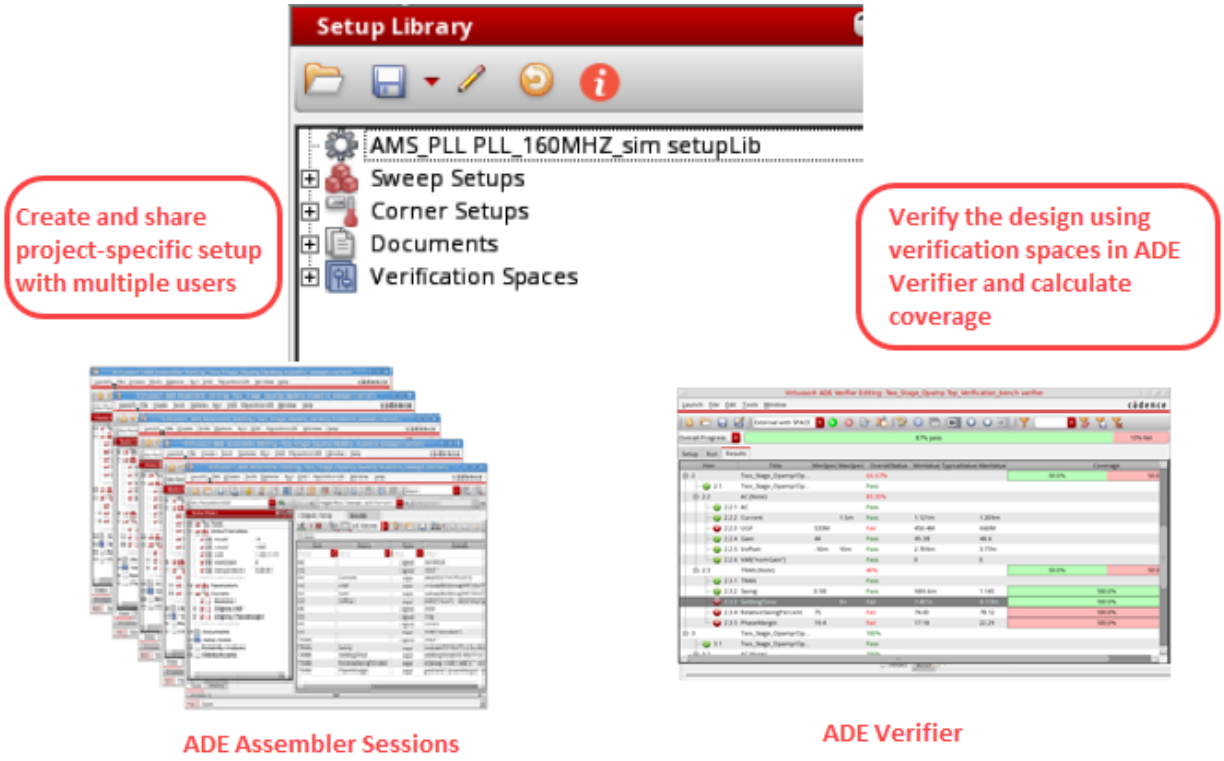
- If you work at the design level (block-level) for analog designing, then you can open this assistant in ADE Assembler to reuse the specified setup to create the setup in the Data View assistant.
- If you work on the top-level design verification (managing the overall project), then you can open this assistant in ADE Verifier to create verification spaces and ensure full coverage of the analog design blocks.

  In ADE Verifier, you can open the Setup Library assistant, load a predefined Setup Library cellview, create verification spaces, and then assign these spaces to different

© 2023 All Rights Reserved.

blocks in ADE Verifier. You can then calculate the coverage percentage and generate a coverage report to ensure that whether the specified setup values in verification spaces are run or not.

For more information on how this assistant works in Virtuoso ADE Assembler, see Working with the Setup Library.



Related Topics

Setup Library Assistant Workflow

Graphical User Interface of Setup Library Assistant

Coverage Calculation in Virtuoso ADE Verifier

Running Simulations with Verification Spaces

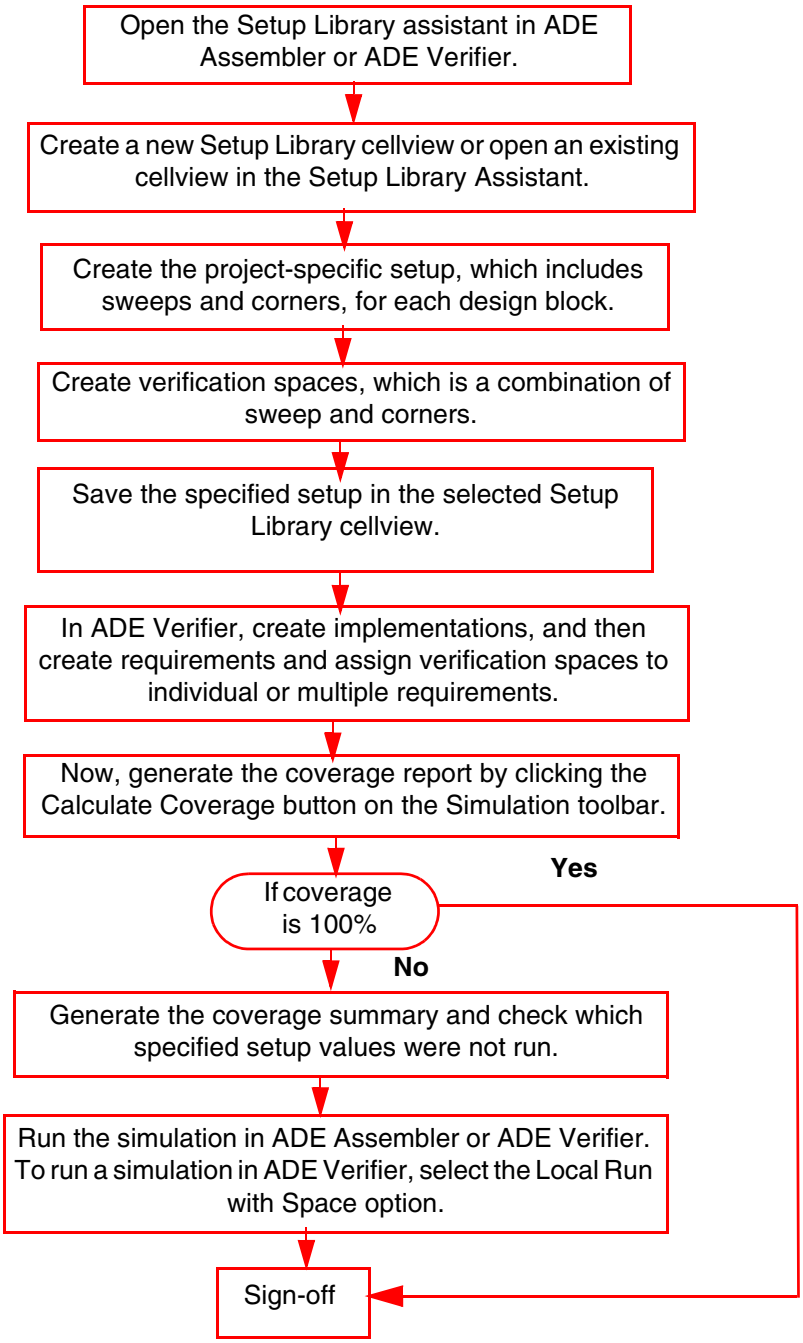
Updating Implementation Cellviews with Verification Spaces

Setting ADE Verifier Preferences for Coverage

© 2023 All Rights Reserved.

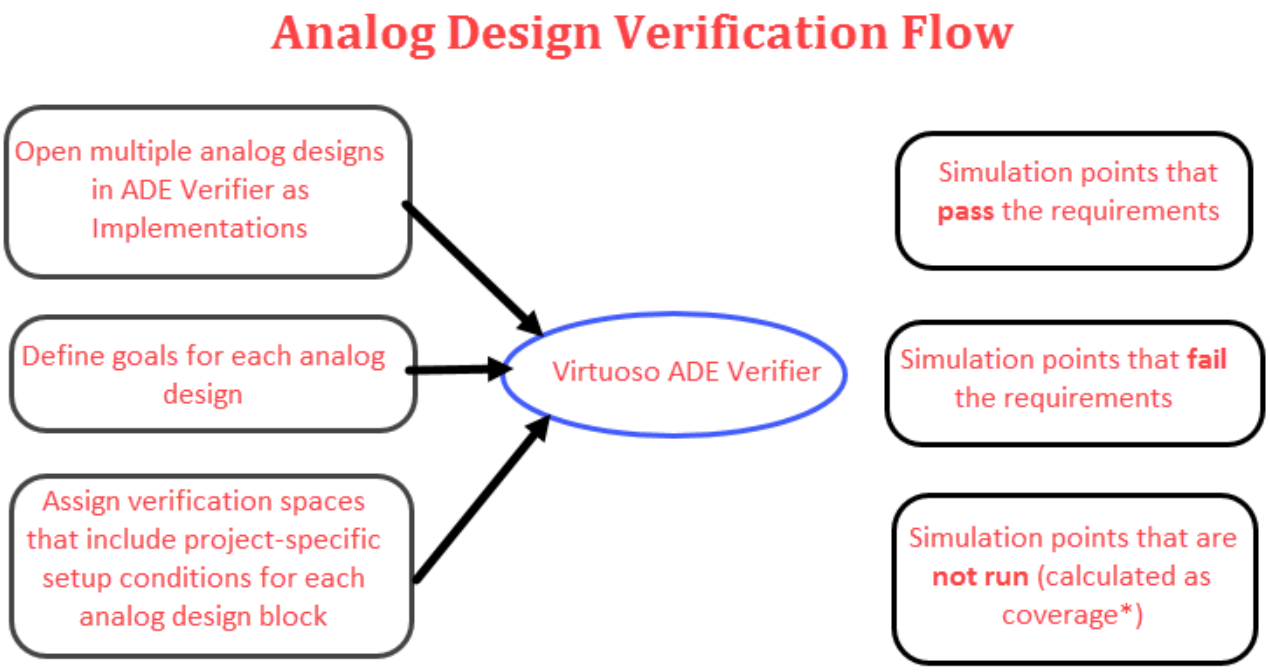
Setup Library Assistant Workflow

The following figure illustrates how you can use the Setup Library assistant in ADE Assembler and ADE Verifier.



© 2023
All Rights Reserved.

The following figure describes the analog design verification workflow in ADE Verifier:



**Coverage is a scale to measure the completeness of analog design in reference to the project-specific setup conditions*

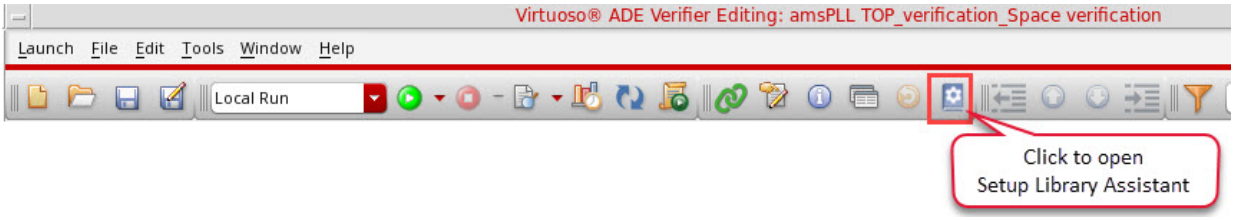
© 2023
All Rights Reserved.

Opening the Setup Library Assistant

You can open the Setup Library assistant using ADE Assembler and ADE Verifier.

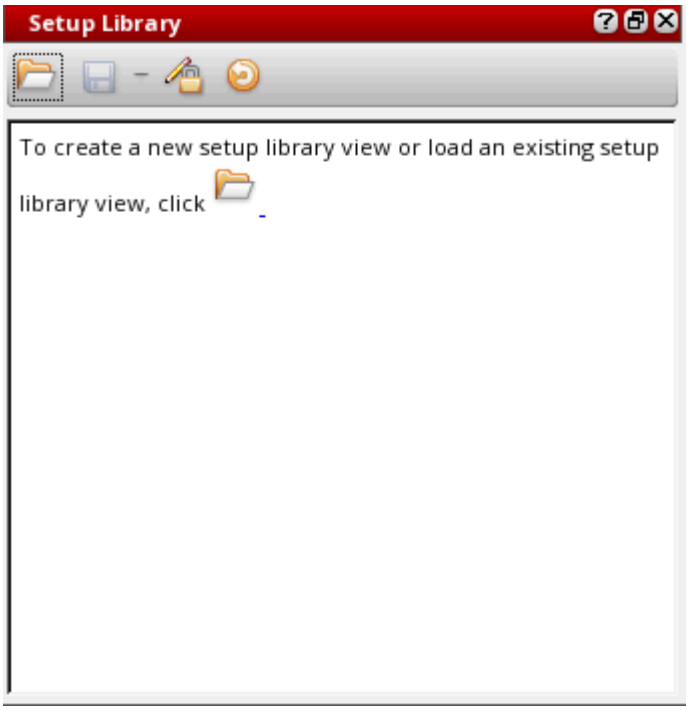
To open Setup Library assistant using ADE Verifier, do one of the following:

- Right-click anywhere on the menu bar or toolbar area and choose *Assistants – Setup Library*.
- Click the Setup Library Assistant icon ()



Alternatively, choose *Windows – Assistants – Setup Library*.

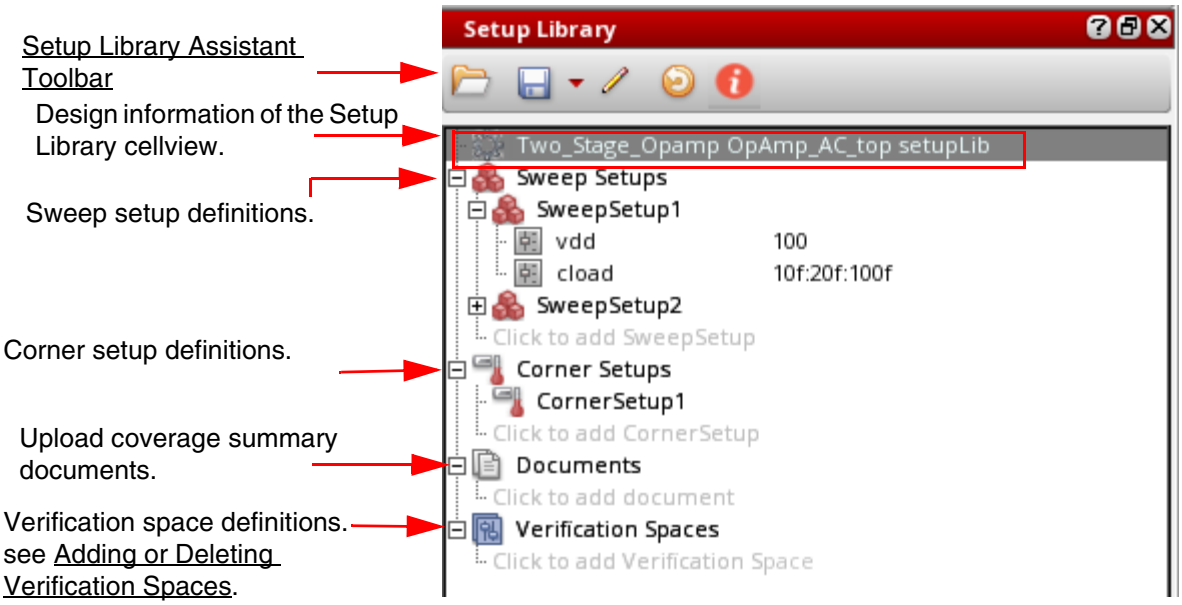
The Setup Library assistant appears on the right of the ADE Verifier window.



© 2023
All Rights Reserved.

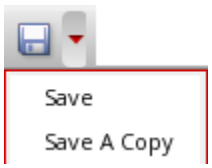
Graphical User Interface of Setup Library Assistant

The Setup Library assistant includes a toolbar and an interface for defining and displaying the setup elements. The figure below explains the graphical user interface of this assistant:



Setup Library Assistant Toolbar

The following table describes the buttons and options available on the toolbar of the Setup Library assistant.

Command	Description
	Creates a new Setup Library cellview or opens an existing Setup Library cellview in the Setup Library assistant.
	Saves the changes made to the setup elements in the same or a different Setup Library cellview. This drop-down list includes the following two options: ■ *Save* – Saves the changes in the same cellview. ■ *Save A Copy* – Saves the changes in another cellview. When you click this option, the Save As Setup Library CellView form appears in which you can specify the library, cell, and view name to save the setup.

© 2023
All Rights Reserved.

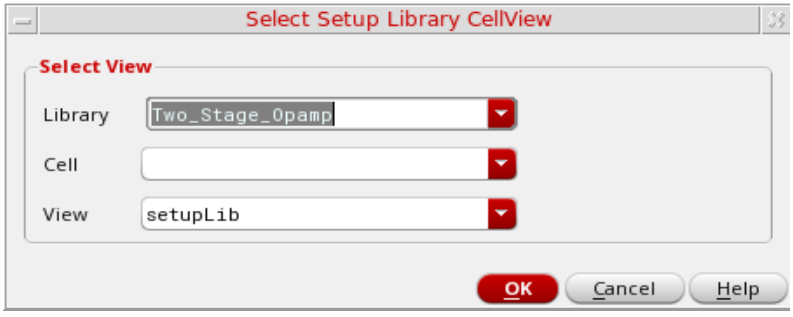
Command	Description
	Controls the editing of setup loaded in Setup Library assistant. When the setup is loaded, by default a lock symbol is displayed on this button to indicate that the mode is read-only. If you have the permissions, you can edit the setup. When you click the Toggle Edit Mode button, the lock symbol disappears and mode is changed to edit. In addition, this lock also prevents you to make changes simultaneously when someone else is editing the Setup Library cellview. If the cellview is being edited by someone, a warning message is displayed to inform you that you cannot edit the setup in the cellview. **Note:** When you switch the Setup Library assistant from edit mode to read-only mode, a message appears that suggest you to save the updates made in the Setup Library.
	Refreshes the setup with the latest changes. A warning icon on this button indicates that the setup has been modified. When you click the button, the setup is updated to show the latest changes.
	Opens the Task Assistant that provides quick help about the Setup Library assistant.

Creating or Loading a Setup Library Cellview

To open a Setup Library cellview:

1. Click the folder icon () on the toolbar.

   The Select Setup Library CellView form appears.



2. In the *Library* and *Cell* lists, select a library and a corresponding cell.

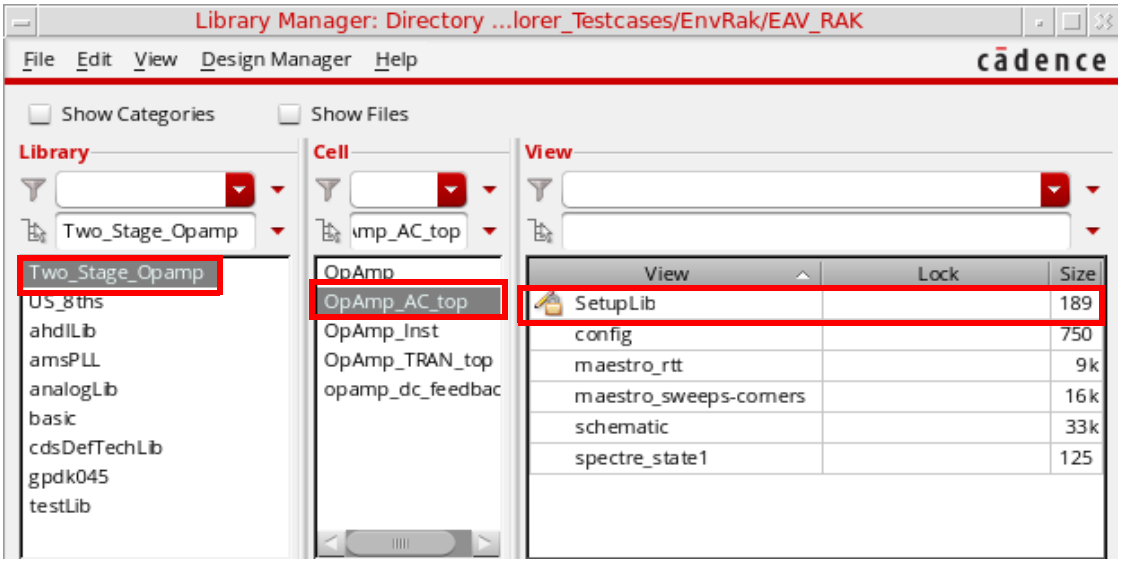
© 2023
All Rights Reserved.

3. In the *View* list, all the existing cellviews of type `setupLib` are displayed. By default, the view type is displayed as `setupLib` when you open the form.
   - ❑ To create a new cellview, specify a cell name for the Setup Library view.
   - ❑ To load an existing cellview, select the cellview name from the *View* list.
4. Click *OK*.

The specified Setup Library cellview is created or loaded in the Setup Library assistant.



The new Setup Library cellview you create is saved and displayed in the *View* section of Library Manager. For example, if you create a new cellview `SetupLib` for library `Two_Stage_Opamp` and cell `OpAmp_AC_top`, this cellview is displayed in the Library Manager window.



© 2023
All Rights Reserved.

Modification of the Setup Library Cellview

When a new Setup Library cellview is created, it does not include any setup elements. You need to edit the setup to add elements to that cellview. You might also want to modify an existing setup to add, delete, or modify elements.



Tip

Before you proceed with editing, ensure that the cellview is enabled for editing. The lock symbol on the Toggle Edit Mode button () on the toolbar indicates that the cellview is read-only. Click this button to enable the edit mode.

By default, all the corners or variables in the Setup Library Assistant are considered enabled even if the set of variables or corners that are dragged into or imported in the Setup Library Assistant are disabled.

In all operations related to creating or editing setups, valid characters must be used to specify the name of the sweep, corner, or simulation setup. Invalid characters include the following:

`space, *, =, <, >, ?, |, ', \, ", ~, (, ), /,` and `; .`

You can create a new project-specific setup, which includes sweep and corners, using ADE Assembler. For information about how to add new sweep and corner combinations in ADE Assembler, see Editing the Setup.

Once the setup is created and saved in a Setup Library cellview, you can open the cellview in ADE Verifier and calculate the coverage to check the completeness of the project setup requirements.

Related Topics

Adding New Sweep Setups

Adding New Corner Setups

Adding New Simulation Setups

Adding New Simulation Setups

Importing Sweeps and Corners in the Setup Library

© 2023 All Rights Reserved.

Adding New Sweep Setups

To add a new sweep setup, do one of the following:

- Right-click *Sweep Setups* and choose *Create Sweep Setup*.
- Expand the *Sweep Setups* node and click the `Click to add SweepSetup` text.

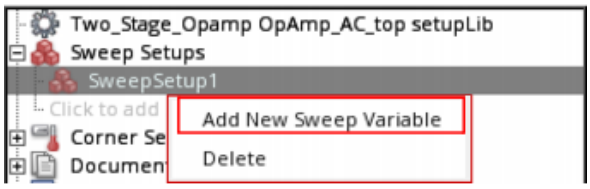
The sweep setup is added and displayed in the node hierarchy. The default sweep setup name is `SweepSetup1`. You can also rename it if required.

Ensure that the sweep setup name is a non-empty string which is not '.' or '..' and does not contain whitespaces or the characters: '~*/<>'?|\"()=;&$'."

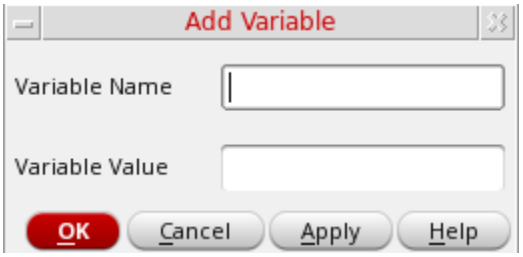
After the sweeps are added in the Setup Library assistant, you can edit them by double-clicking the sweep name and sweep value.

To add a sweep variable in this sweep setup:

➥ Right-click the sweep setup name, such as `SweepSetup1`, and choose *Add New Sweep Variable*.



The Add Variable form appears in which you can specify the sweep variable name and value.



The specified variable is listed under `SweepSetup1` in a new row.

To delete a sweep variable:

➥ Right-click the variable and choose *Delete*.

To delete the entire sweep setup:

© 2023 All Rights Reserved.

➥ Right-click the sweep setup node and choose *Delete*.

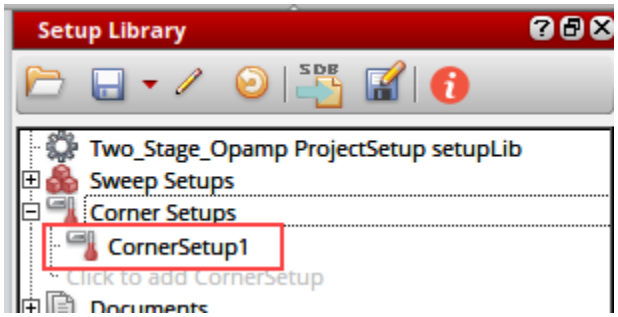
Adding New Corner Setups

To add a new corner setup, do one of the following:

- Right-click *Corner Setups* and choose *Create Corner Setup*.
- Expand the *Corner Setups* tree and click the `Click to add CornerSetup` text.

The corner setup is added and displayed in the hierarchy. The default corner setup name is `CornerSetup1`. You can also rename it if required.

Ensure that the corner setup name is a non-empty string which is not '.' or '..' and does not contain whitespaces or the characters: `'~\\*/<>'?|\"()=;&$'."`



After the corner is added in the Setup Library Assistant, you can do one of the following:

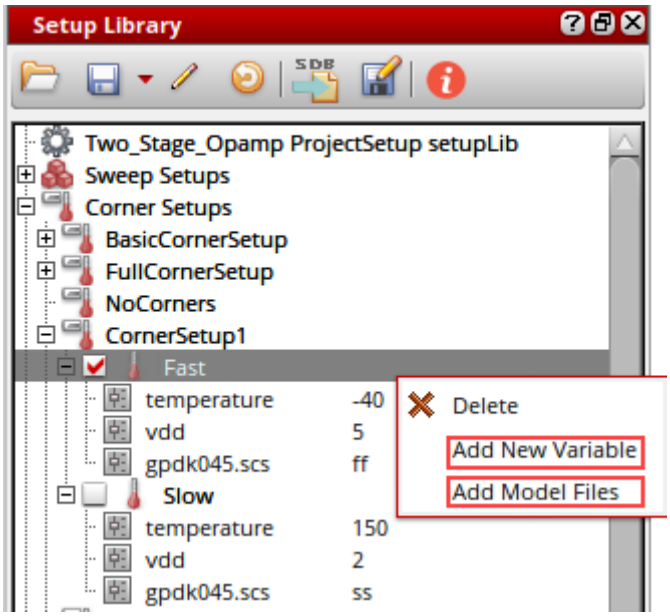
➥ Import corners from a CSV file by right-clicking the corner setup name and selecting *Import Corners*.

➥ Add the corner in the ADE Assembler Setup that uses the same Setup Library cellview. See Adding New Corner Setups.

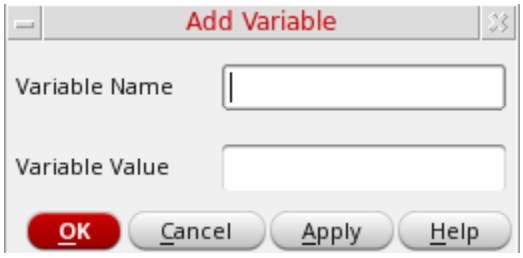
To add a corner variable in this corner setup:

© 2023 All Rights Reserved.

➡ Right-click a corner name under `CornerSetup1`, and choose *Add New Variable*.



The *Add Variable* form appears in which you can specify the corner variable name and value.

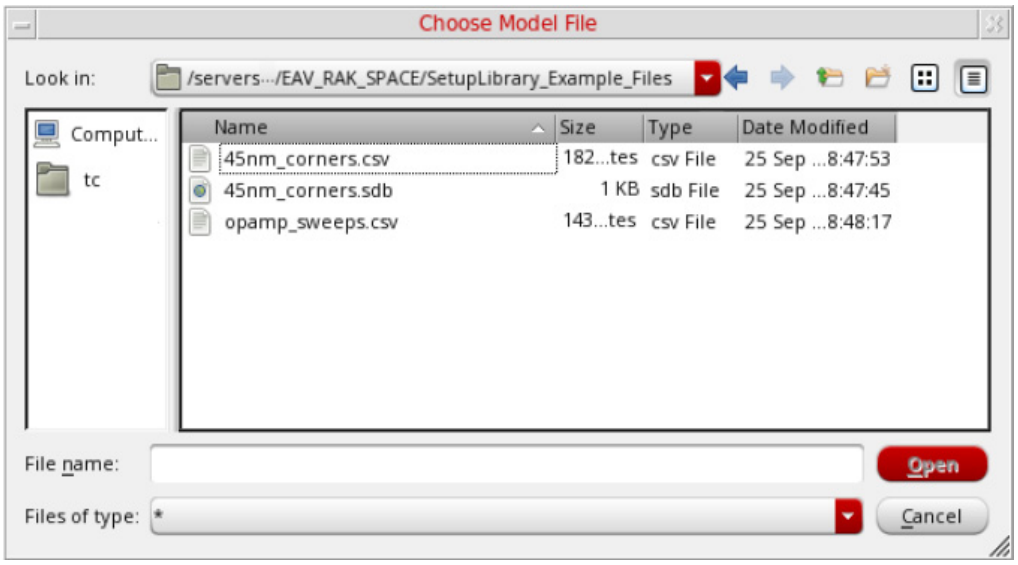


The specified variable is listed under `CornerSetup1` in a new row.

To add a model file in this corner setup:

© 2023
All Rights Reserved.

➡ Right-click the corner name under `CornerSetup1`, and choose *Add Model Files*.



The *Choose Model File* window appears in which you can specify the model file name.

To delete a model file:

➡ Right-click the file name and choose *Delete*.

To delete a corner variable:

➡ Right-click the variable and choose *Delete*.

To delete the entire corner setup:

➡ Right-click the corner setup grouping and choose *Delete*.

© 2023
All Rights Reserved.

Adding New Simulation Setups

Important

Currently, the Setup Library assistant supports only Monte Carlo simulation setups. You can add a Monte Carlo simulation setup only when the currently opened setup library view is editable.

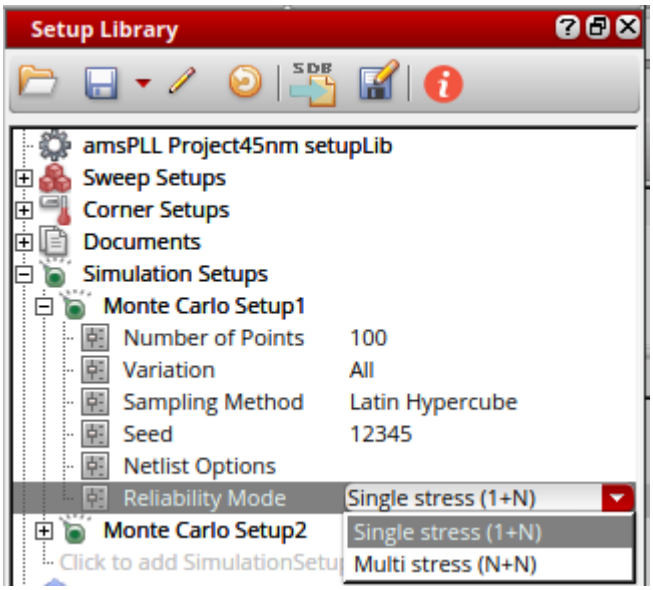
To add a new simulation setup, do one of the following:

- Right-click *Simulation Setups* and choose *Create New Monte Carlo Setup*.
- Expand the *Simulation Setups* node and click the `Click to add SimulationSetup` text.

  The Monte Carlo simulation setup is added and displayed in the *Simulation Setups* section. The default name of the new simulation setup is `Monte Carlo Setup1`, which you can rename as required.

  Ensure that the corner setup name is a non-empty string which is not '.' or '..' and does not contain whitespaces or the characters: '~*/<>'?|\"()=;&$'."

  The setup contains the variables *Number of Points*, *Variation*, *Sampling Method*, *Seed*, *Netlist Options*, and *Reliability Mode*.



© 2023
All Rights Reserved.

You can specify custom values for the variables in the simulation setup that will let you run a Monte Carlo simulation.

To specify values for the variables in the simulation setup:

1. Double-click the *Number of Points* variable row and specify the number of points.
2. From the *Variation* list, select the method. Possible values are *All*, *Mismatch*, and *Process*.
3. From the *Sampling Method* list, select the mode. Possible values are *Random*, *Latin Hypercube*, and *Low-Discrepancy Sequence*.
4. Double-click the *Seed* variable row and specify the seed value.
5. Double-click the *Netlist Options* variable row and specify the netlisting option.
6. From the *Reliability Mode* list, select the reliability mode. Possible values are *Single stress (1+N)* and *Multi stress (N+N)*.

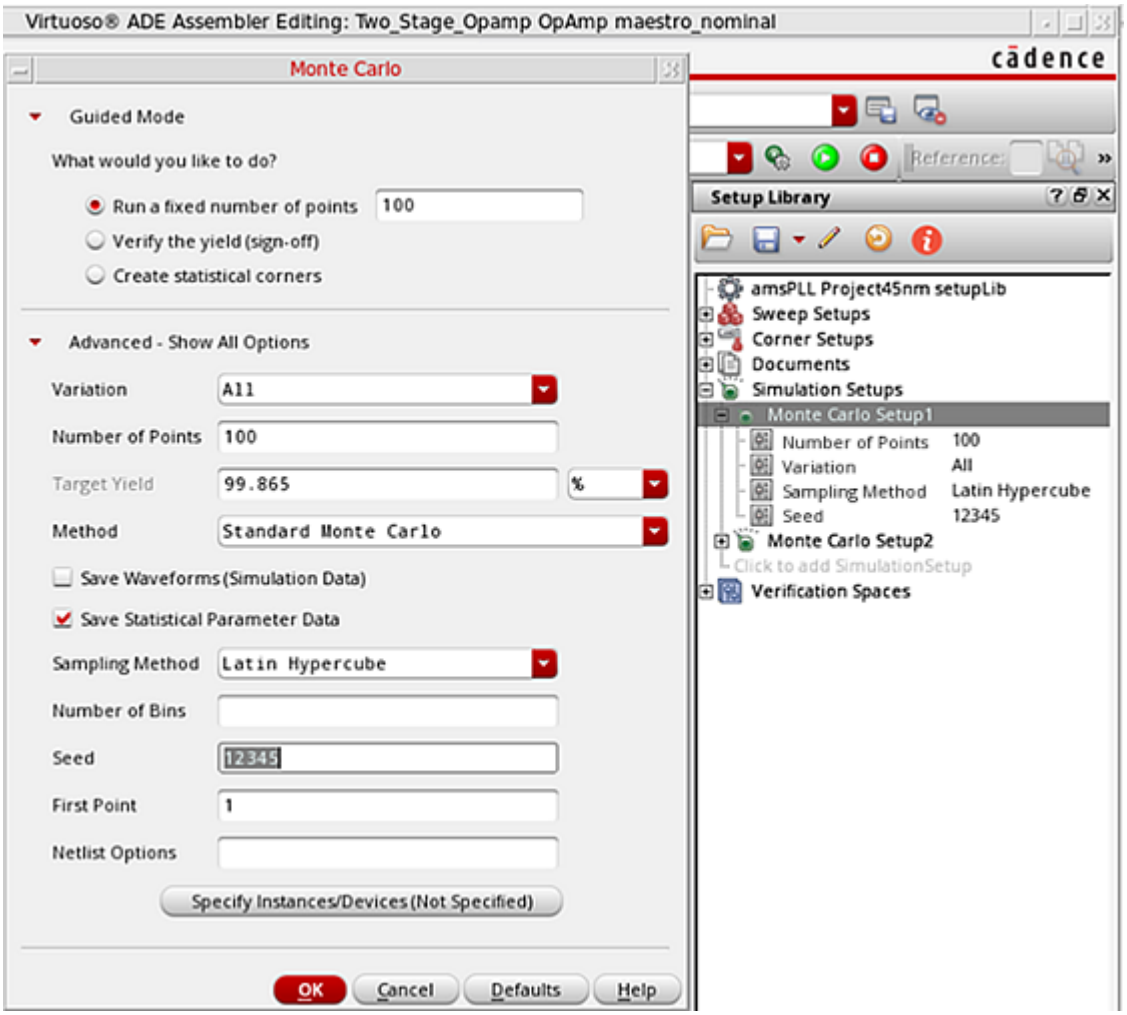
The specified variable values are listed in the `Monte Carlo Setup1` group. You can double-click the displayed variable value to change it directly in the variable row.

You can add more simulation setups in your setup library view.

When you drag a Monte Carlo simulation setup from the Setup Library assistant to the Data View assistant in ADE Assembler, the run mode changes to *Monte Carlo Sampling*.

© 2023 All Rights Reserved.

Accordingly, the run options are displayed in the Monte Carlo form. The form shows the values that you have specified in `Monte Carlo Setup1`.



To delete a specific simulation setup from the Setup Library assistant:

➡ Right-click the setup name and choose *Delete*.

To view all the variables in a simulation setup:

➡ Right-click the simulation setup name and choose *Expand All*.

To hide all the variables in a simulation setup:

➡ Right-click the simulation setup name and choose *Collapse All*.

© 2023
All Rights Reserved.

Important

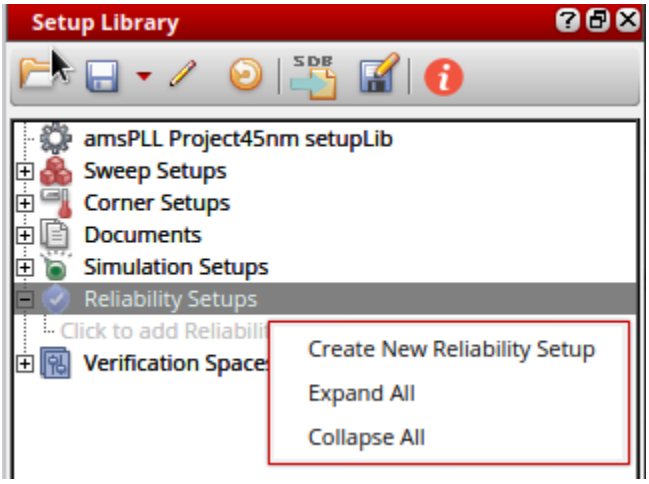
Currently, the run mode *Batch with SPACE* does not support simulation setups.

Adding New Reliability Setups

To add a new reliability setup, ensure that the setup library cellview is editable and do one of the following:

- Right-click *Reliability Setups* and choose *Create New Reliability Setup*.
- Expand the *Reliability Setups* node and click the `Click to add Reliability Setup` text.

A new reliability setup is added and displayed in the *Reliability Setups* section. The default name of the new reliability setup is `ReliabilitySetup1`, which you can rename as required. The second reliability setup that you create, is by default named `ReliabilitySetup2`, and so on.

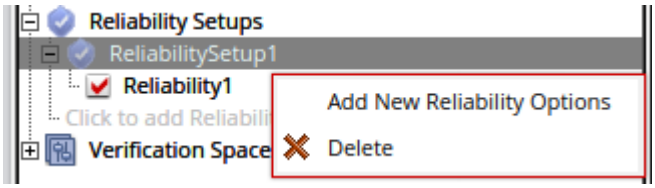


Ensure that the reliability setup name is a non-empty string, which is not `'.'` or `'..'` and does not contain whitespaces or these unsupported characters: `'~\\*/<>'?|\"()=;&$'."`

© 2023 All Rights Reserved.

To add reliability options to a reliability setup, do one of the following:

➡ Right-click the setup name and select *Add New Reliability Options*.



The default name of the first set of reliability options is `Reliability1`. You can add more reliability options in your reliability setup. The second reliability options node that you add is suffixed with an incremental number `Reliability`*`N`*, to keep it unique.

➡ Drag one or more reliability options from Data View in ADE Assembler and drop them to any reliability setup or reliability options node in the setup library cellview.

This process copies all information except the test name, stress file, config type, model files, and placement type. If the *Reliability Setups* group contains multiple reliability options, you can enable the required options and disable the others.

To enable or disable a set of reliability options, do one of the following:

- Right-click the reliability options node and select *Enable* to enable the reliability options. Alternatively, select the check box for this set of reliability options.
- Right-click the reliability options node and select *Disable* to disable the reliability options. Alternatively, deselect the check box for this set of reliability options.

The *Enable* and *Disable* commands are available for multiple selection when the setup library cellview is editable.

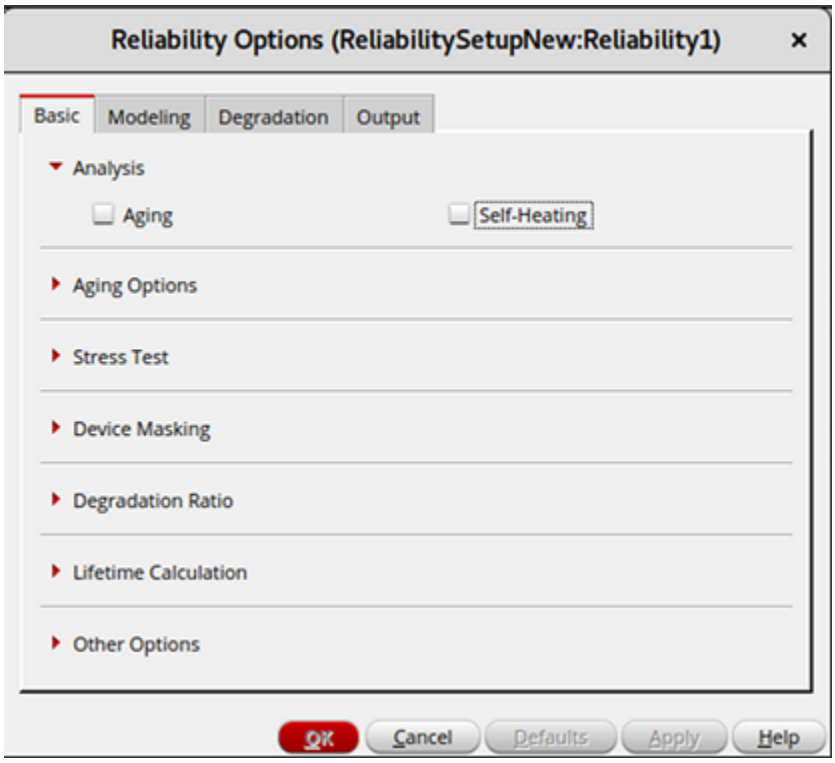
To edit the reliability options in a reliability setup:

➡ Right-click the reliability options node and select *Edit Reliablity Options*.

The *Edit Reliability Options* command is only available when the setup library cellview is editable.

© 2023 All Rights Reserved.

The Reliability Options form is displayed.



This form is the same as the Reliability Options form that is opened from the Data View in ADE Assembler. The form name is suffixed with the names of the reliability setup and reliability options in the following format.

`Reliability Setup_name:ReliabilityOptions_name`

All options in this form are set to the default values that are read from the `spectre/.cdsenv` file. You can modify the values, if required.

To view the reliability options in the read-only mode:

➡ Right-click the reliability setup name and choose *View Reliability Options*.

This command is available only when the setup library view is in read-only mode. It is not available when you select multiple reliability options.

The Reliability Options form is displayed in view-only mode. The *OK*, *Defaults*, and *Apply* buttons are disabled. This means that even if you modify any values in the form, the settings are not saved in the reliability setup.

To delete a specific reliability setup from the Setup Library assistant:

© 2023 All Rights Reserved.

➡ Right-click the setup name and choose *Delete*.

To view all the reliability options in a reliability setup:

➡ Right-click the reliability setup name and choose *Expand All*.

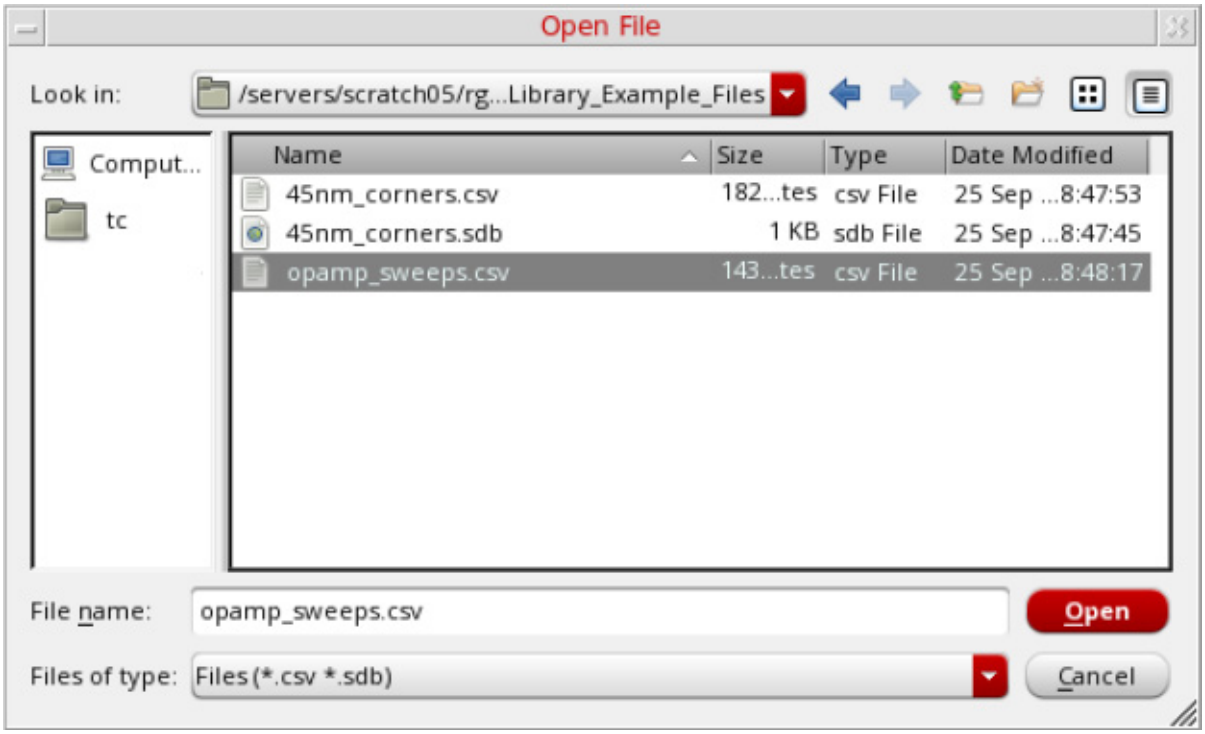
To hide all the reliability options in a reliability setup:

➡ Right-click the reliability setup name and choose *Collapse All*.

Importing Sweeps and Corners in the Setup Library

To import a sweep setup:

➡ Right-click the sweep setup name, such as `SweepSetup1`, in the Setup Library assistant and choose *Import Sweeps*.



The *Open File* window appears in which you can specify the CSV or SDB file containing the sweep setup. If the imported file contains parametric sets, the row containing the parametric set details must contain the same number of fields as those of the variables.

Similarly, to import a corner setup:

© 2023
All Rights Reserved.

➡ Right-click the corner setup name, such as `CornerSetup1`, and choose *Import Corners*.

The *Open File* window appears in which you can specify the CSV or SDB file containing the corner setup.

Additional Information

- If a variable already exists with the same name as that of a variable being imported, it is overwritten by the imported variable.
- The enabled or disabled status of the imported variable is also imported and set accordingly.
- If the information about parametric sets is available in the CSV file, the parametric sets are also created after importing the variables.

Exporting and Importing SDB Setups

You can export the sweep setups, corner setups, simulation setups, and verification spaces from an open setup library view to an SDB file, which is also saved within the same setup library cellview. Similarly, you can import the sweep setups, corner setups, simulation setups, and verification spaces from an SDB file to an open setup library view.



To export all the setups:

1. In the Setup Library assistant toolbar, click *Export setups to SDB file*.

   The Specify a File dialog appears.
2. Specify the `.sdb` file to which the setups must be exported.
3. Click *Save*.

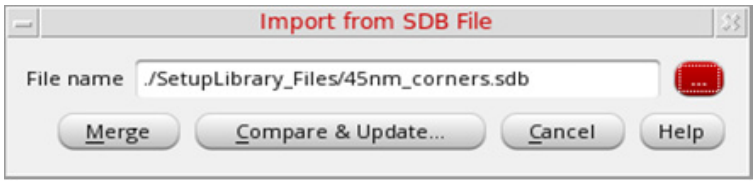
All setups from the setup library cellview are exported to the specified SDB file.

To import setups:

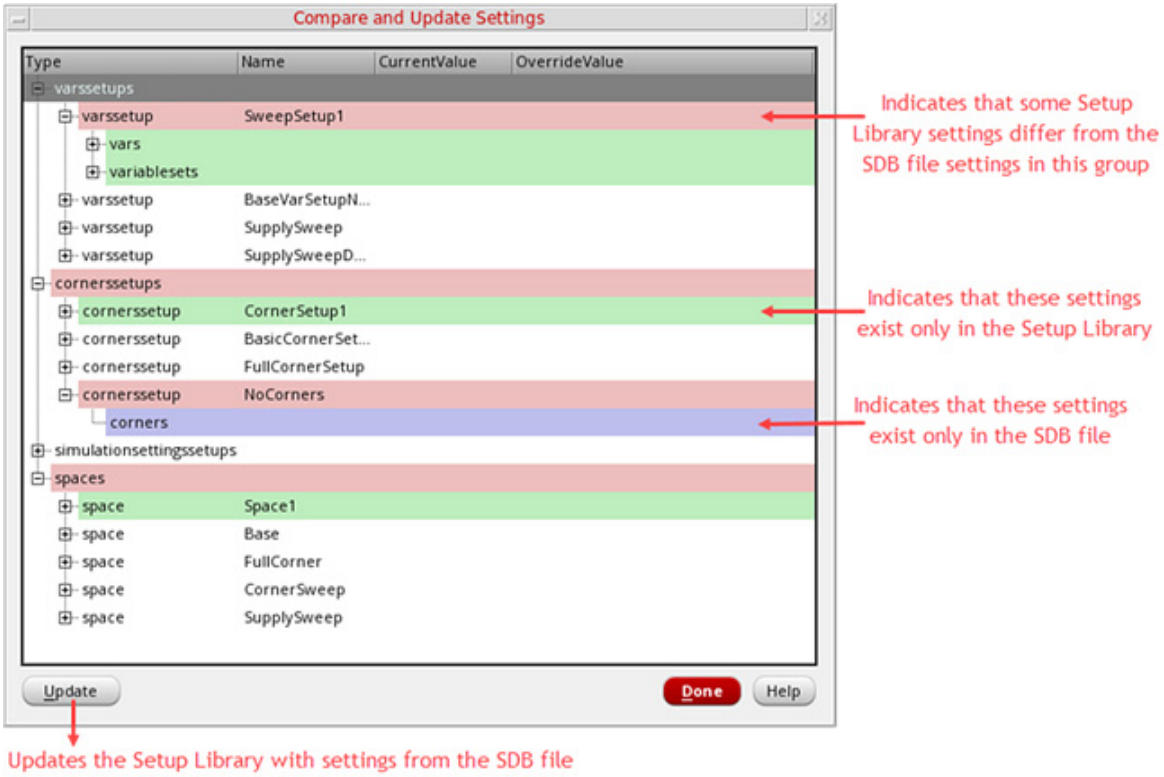
1. In the Setup Library assistant toolbar, click *Import setups from SDB file*.

© 2023 All Rights Reserved.

The Import from SDB File form appears.



2. Specify a `.sdb` file from which the setups must be imported.
3. Click *Merge* to merge the new and modified settings from the SDB file into the Setup Library. In this case, no visual indicators are displayed.
4. Click *Compare & Update* to open the Compare and Update Settings form. This form displays a comparison of the Setup Library and the SDB file settings where differences are color-coded and appropriate tooltips are displayed.



5. Do one of the following:
   - Click *Update* to make the following changes:
     - The settings highlighted in purple are added to the Setup Library.
     - The settings highlighted in green are removed from the Setup Library.

© 2023
All Rights Reserved.

- The settings highlighted in pink are updated or modified in the Setup Library.

- Click *Done* to close the Compare and Update Settings form.

The Setup Library is updated with the setups imported from the specified SDB file.

Deleting a Setup Library Cellview

To delete a Setup Library cellview:

1. Open the Library Manager window.
2. In the *Library* and *Cell* drop-down lists, select the required library and a corresponding cell.
3. In the *View* drop-down list, choose the required cellview of type `setupLib` that you want to delete.
4. Do one of the following:
   - Choose *Edit – Delete*.
   - Right-click the required cellview in the *View* pane and select *Delete*.

The specified Setup Library cellview is deleted from the Library Manager.

Important

If you open a `setupLib` cellview using the Setup Library Assistant, which has been deleted from the Library Manager, the default home screen is displayed in the Setup Library Assistant. To set up the association between the `setupLib` and Setup Library Assistant, you need to load the `setupLib` cellview again.

© 2023 All Rights Reserved.

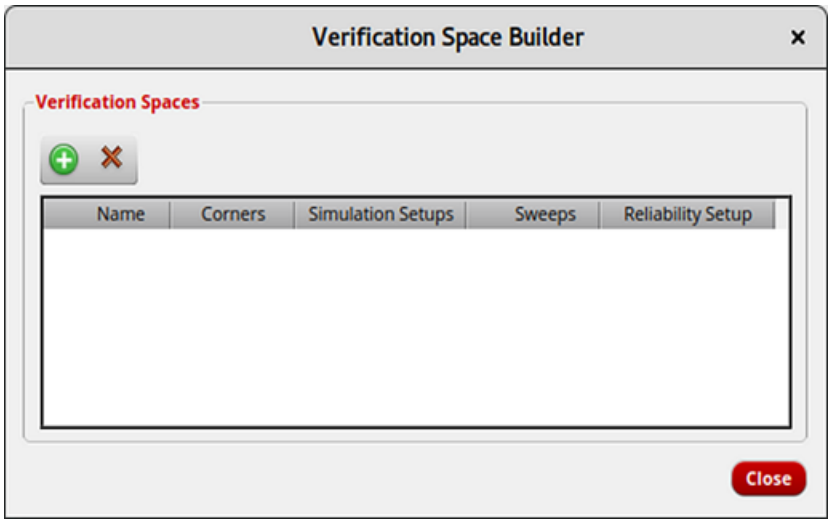
Adding or Deleting Verification Spaces

A verification space is a combination of sweeps, corner values, and simulation setups that you specify in the Setup Library assistant. You can assign these verification spaces to different design blocks and use them to calculate the coverage.

To create verification spaces, do one of the following:

- Right-click the *Verification Spaces* node and choose *Create Verification Space*.
- Click the `Click to add Verification Space` text displayed under the *Verification Spaces* node.

  The Verification Space Builder form appears.



To add a new verification space,

1. Click *Create New Space* ( ).

   A new space row is added and named as `Space1`. You can double-click the space name and rename it if you want.

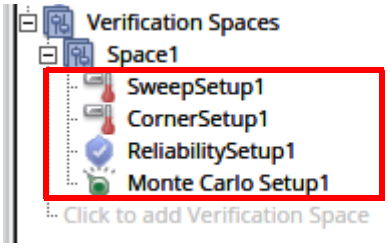
2. Select the following, as required:
   - a corner setup from the *Corners* list
   - a sweep setup from the *Sweeps* list
   - a Monte Carlo simulation setup from the *Simulation Setups* list

© 2023
All Rights Reserved.

- a reliability setup from the *Reliability Setup* list

Name	Corners	Simulation Setups	Sweeps	Reliability Setup
Space1	CornerSetup1	Monte Carlo Setup1	SweepSetup1	ReliabilitySetup1

The selected corner, simulation setup, sweep combinations, and reliability setup are added to the selected space and displayed in the *Verification Spaces* tree, as shown in the figure below:



Similarly, you can create more verification spaces and they are by default named as `Space2`, `Space3`, and so on.

To delete a verification space:

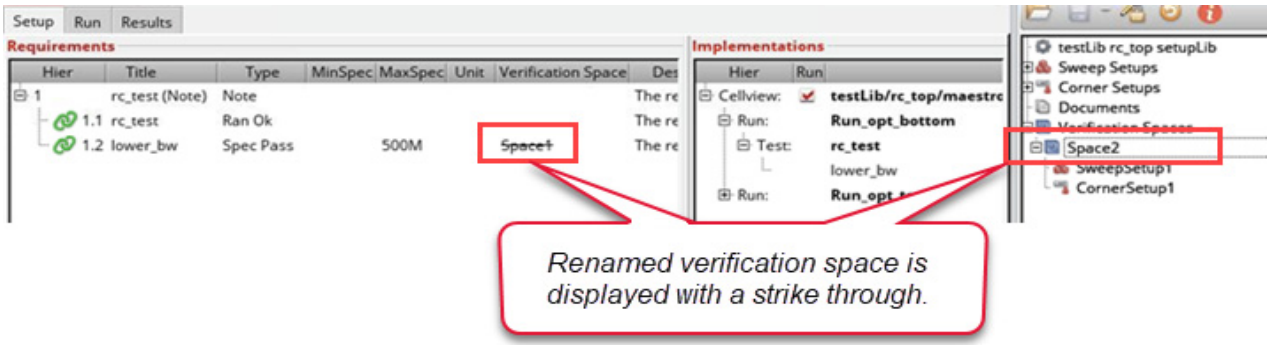
➡ Right-click the space in the Setup Library assistant and click *Delete*.

Alternatively, you can open the Verification Space Builder form by clicking the `Click to add Verification Space` text displayed under the *Verification Spaces* node. You can select a space row and click *Delete*.

In an existing setup, you might want to modify or rename the verification space. If you rename a verification space in the Setup Library Assistant, and this verification space is being used in ADE Verifier, the previous space name is displayed as struck out in ADE Verifier.

© 2023 All Rights Reserved.

For example, you might rename `Space1` to `Space2` in Setup Library Assistant, and ADE Verifier contains a requirement assigned to `Space1`. ADE Verifier synchronizes with the changes in the Setup Library by displaying `Space1` with a strike through in the cellview.



Parametric Sets in Setup Library Assistant

You can create parametric sets in the Setup Library Assistant when you need to run simulations for a specific set of sweep combinations for different parameters. These combinations are created by picking values from the same ordinal position of all variables or parameters in that parametric set. This reduces the number of design points, and therefore, reduces the number of simulations.

To know more on creating and removing parametric sets, see Working with Parametric Sets and Reusing the Setup from Data View and Setup Library Assistant.

Adding or Removing a Verification Space for Each Block

In ADE Verifier, first you need to add implementations in the *Setup* tab. For information about how to add implementations, see Implementations and Mappings in Virtuoso ADE Verifier.

Once the implementations are loaded, select the histories. You can add or modify the requirements if you want. Now, you can add verification spaces to each requirement.

To add verification spaces for one or more requirements:

1. Ensure that the verification spaces are already created in the Setup Library assistant. For more information, see Adding or Deleting Verification Spaces.
2. Right-click a requirement row in the *Setup* tab and choose *Assign Space to Requirements* – `VerificationSpace`.

© 2023 All Rights Reserved.

To specify a verification space for multiple requirements, select the requirements using the `Ctrl` key and choose *Assign Space to Requirements* – `VerificationSpace`.

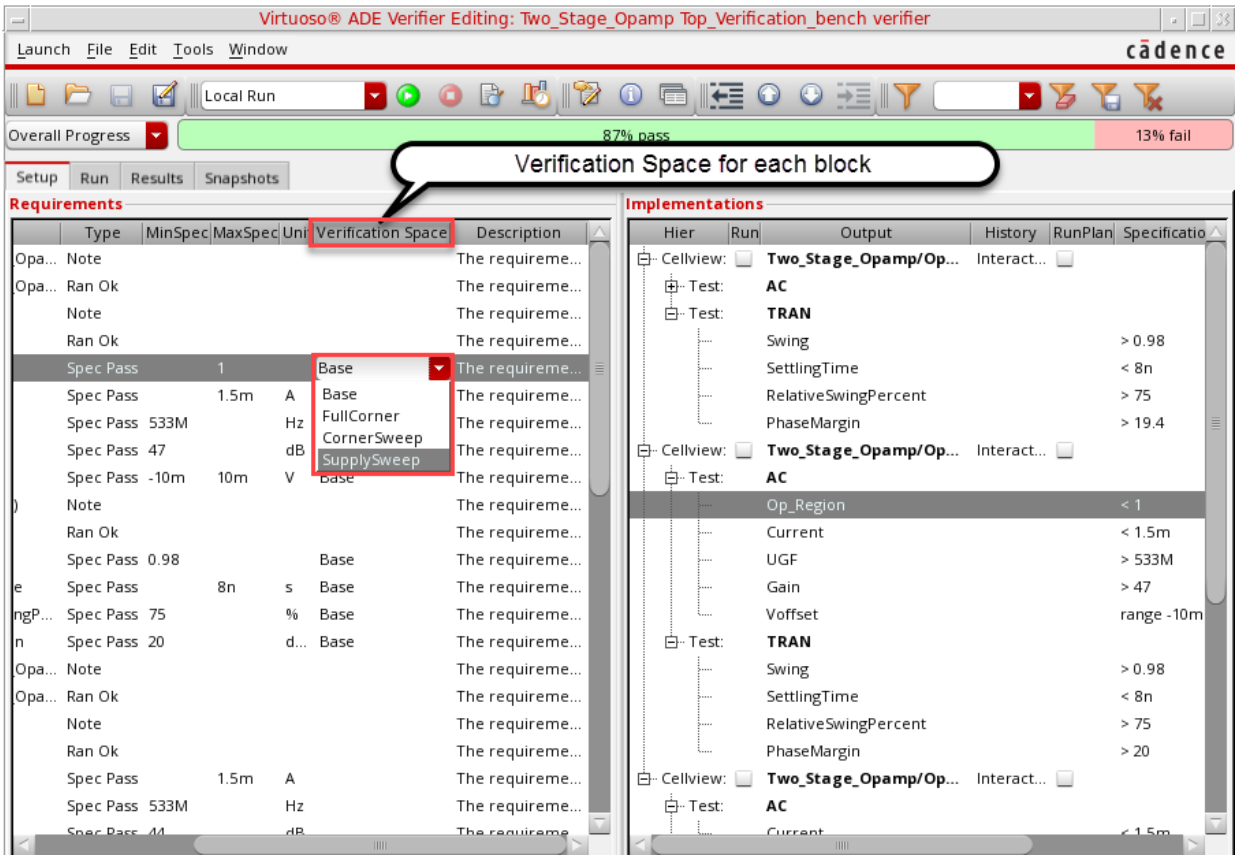
Assign Space to Requirements is disabled when:

- The requirement is read-only or referenced
- The requirement is of type *Note*, *ExtRef*, or its domain is *NON-ANALOG*
- The setup library view is not defined

ADE Verifier returns an error if you try to assign the verification space using SKILL functions.

Note: You can also assign a verification spaces to a statistical output from an implementation that uses a Monte Carlo simulation setup.

3. In the *Verification Space* column, select a verification space from the drop-down list against each value where the specifications are applied.



© 2023
All Rights Reserved.

To remove the verification spaces from one or more requirements:

➡ Hold down the `Ctrl` key, select the requirements, right-click the selection, and choose *Remove Space from Requirements*.

Remove Space from Requirements is disabled when:

- The requirement is read-only or referenced
- A verification space has not been assigned to the requirement

ADE Verifier returns an error if you try removing the verification space using SKILL functions.

Running Simulations with Verification Spaces

You can run simulations in ADE Verifier considering the sweeps and corners defined in the verification space, which saves time and ensures 100% coverage of the analog design.

To run such a simulation:

1. On the *Simulation* toolbar, from the *Run Mode* list, choose *Local with SPACE*. To run multiple simulations in batch mode, choose *Batch with SPACE*.
2. Click the *Run Simulation* button on the Verifier toolbar.

   The simulation for the selected implementation is run using the sweeps and corners specified in the verification space. The results are updated in the *Results* tab in ADE Verifier.

Additional Information

- All requirements mapped to an implementation must be assigned to the same verification space. Otherwise, an error message is displayed and the simulation of implementations mapped to requirements with multiple verification spaces is stopped. All other implementations simulate successfully.
- After you run a simulation in ADE Verifier using the data from the Setup Library assistant, ADE Assembler is opened in the background in the read-only mode and the results are displayed in the *Run* tab. In this case, the current implementation cellview is not updated with the project-specific simulation setup.
- After you run the simulation and open the updated cellview in ADE Assembler, the sweeps and corners from the selected verification space are merged with the sweeps and corners in Data View assistant in ADE Assembler.
  - The same sweeps and corners are overridden.

© 2023 All Rights Reserved.

- The additional sweeps and corners in the Data View assistant remain unchanged.
- The additional sweeps and corners from the Setup Library assistant are appended in Data View.

Updating Implementation Cellviews with Verification Spaces

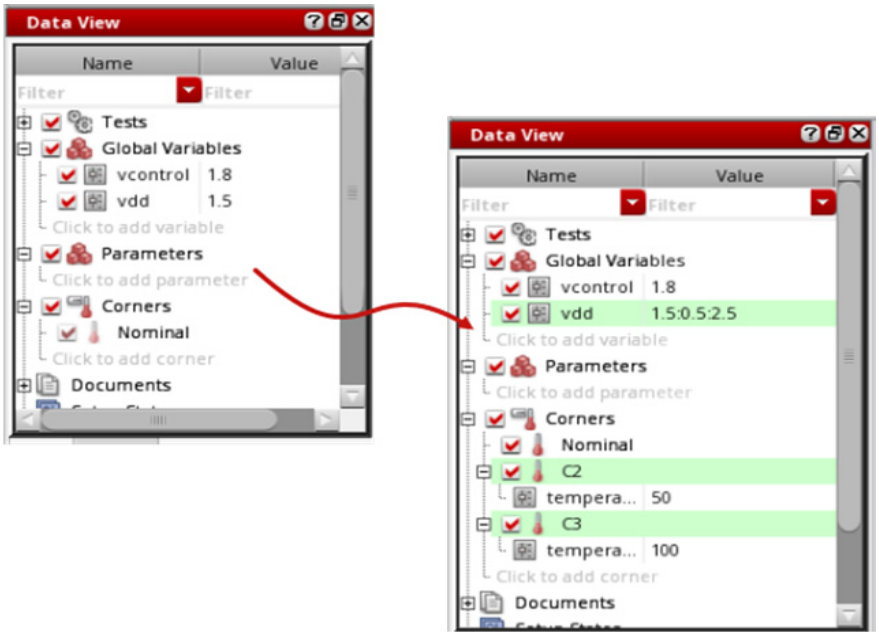
After running a simulation in ADE Verifier, you can perform the following steps to directly update the current implementation cellview with a verification space:

1. On the *Run* tab, right-click any *Implementation* row and choose *Show Details*.
2. In the expanded implementation view, right-click a row and choose *Push Assigned Space in Implementation Cellview – Corner*/*Sweep*/*Simulation Setup/All*.
   - Select *Corner* to update only the corner setup.
   - Select *Sweep* to update only the sweep setup.
   - Select *Simulation Setup* to update only the simulation setup.
   - Select *All* to update sweep, corner, and simulation setups.

   A message box appears confirming the action. When you click *Yes*, the ADE Assembler cellview is updated with the specified corner, sweep, and simulation setup from the

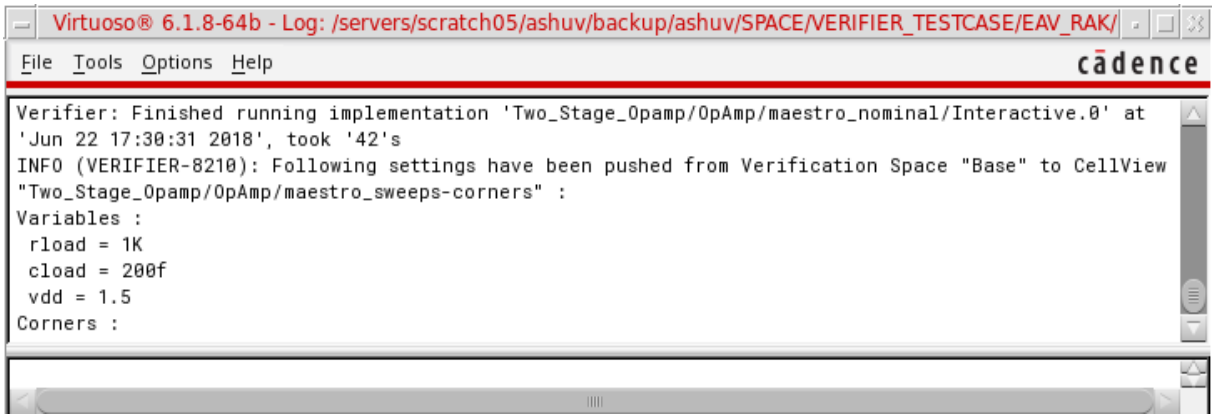
© 2023 All Rights Reserved.

verification space. The Data View assistant highlights the updated corner, sweep, and simulation setup in green, indicating that these are referenced.



The *Verification Space* column on the *Run* tab displays the verification space specified for the implementation. This column is read-only. You can edit the verification space in the *Setup* tab.

In addition, the summary of changes made to the sweep and corner setup for the selected cellview is also displayed in CIW.



© 2023
All Rights Reserved.

Additional Information

- You can update a `maestro` cellview with the project-specific settings only when you have edit permissions on that cellview.
- All the requirements mapped to the implementation cellview, in which you are updating the project-specific values, should be assigned with the same verification space. Otherwise, an error message is displayed.
- After you run the simulation and open the updated cellview in ADE Assembler, the sweeps and corners from the selected verification space are merged with the sweeps and corners in Data View in ADE Assembler.
  - The same sweeps and corners are overridden.
  - The additional sweeps and corners in Data view remain unchanged.
- The additional sweeps and corners from Setup Library assistant are appended in Data View.
- Corners editing capability is available in Setup Library Assistant.

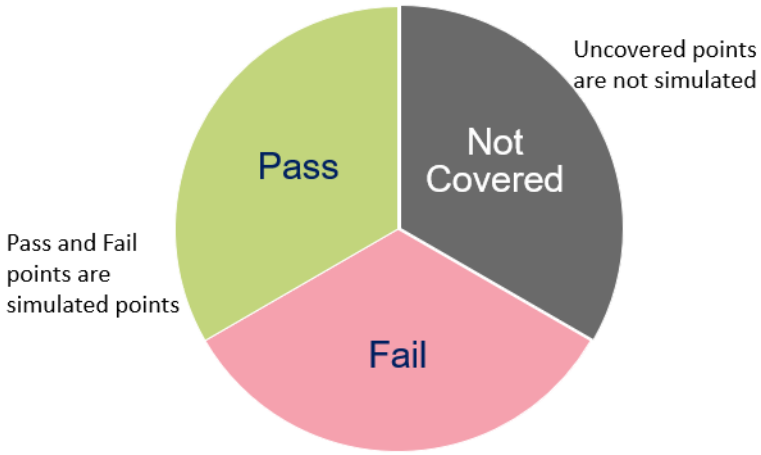
© 2023 All Rights Reserved.

Coverage Calculation in Virtuoso ADE Verifier

In ADE Verifier, you can load the history results, create verification spaces, and verify whether all the setup specifications have been met by calculating the coverage, and generate the coverage report in the *Result* tab of ADE Verifier.

The coverage measures the completeness of the verification setup, which means it identifies if all the project-specific setup requirements defined in the Setup Library cellview have been covered in the simulation results for each block.

The simulation results can include the pass and fail percentage of all the points that have been simulated during the simulation runs. When the coverage is generated in ADE Verifier, it calculates the percentage of points that are not simulated.



For example, if the specifications include 250 corners and only 100 of them are simulated, you get the coverage of 40% on the given design. This indicates that 60% of the corners have not been run, regardless of the number of outputs passing the specifications.

The coverage can be calculated as follows:

$$Coverage = \frac{AchievedVerificationGoals}{TargetVerificationGoals}$$

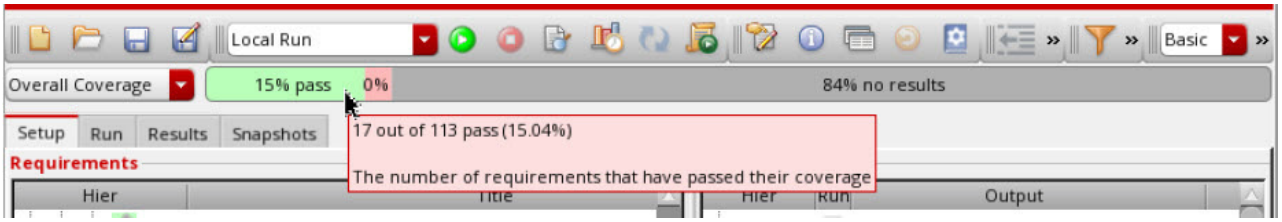
These coverage results are saved in and then loaded from the *cellview_dir*`/results/coverages.json` file.

- For read-only Verifier cellviews, the coverage is saved in the *project_dir*`/`*cellview_dir*`/results/coverages.json` file.

© 2023 All Rights Reserved.

- For referenced cellviews in a multi-user setup, the coverage results are saved in and the loaded from the *referenced_cellview_dir*/results/coverages.json file.

You can also view the Overall Coverage in a progress bar on the *Setup* tab. The following illustration shows the number of requirements that have been successfully covered in the verification:



You can manually sign off failed requirements to increase the coverage of your verification project. However, failed requirements that are mapped to implementations with uncovered points are not included in the coverage calculation. For details, see Requirements Signoff in Virtuoso ADE Verifier.

To know more about Overall Coverage, see Overall Progress, Coverage, and Overall Coverage.

Calculating the Coverage Percentage

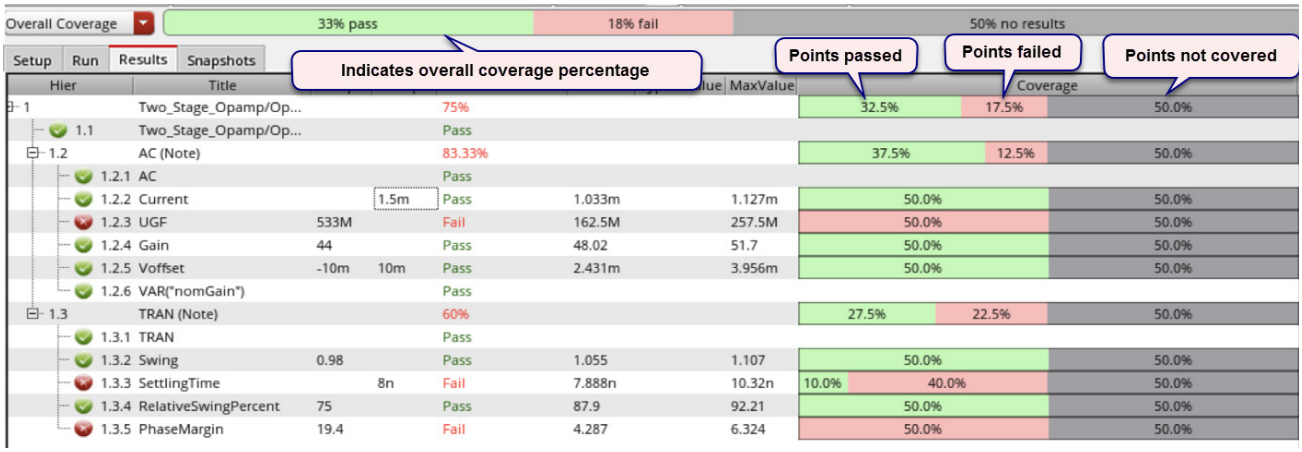
After the verification space is specified for each block requirement, you can generate the coverage report in the *Result* tab. To do this:

➡ Click *Calculate Coverage* ( ) available on the ADE Verifier toolbar.

To indicate that the coverage might have changed and needs to be recalculated, *Calculate Coverage* is displayed as . The coverage may also change if you modify a corner or variable that is a part of an existing verification space used in Verifier. Update the coverage by clicking *Calculate Coverage* ( ).

© 2023 All Rights Reserved.

The coverage report is displayed in the *Results* tab. For example, the following figure shows the coverage report for the specified `opamp` requirement.



In this report, the bar on the top shows the overall coverage, which indicates that 33% of the points have passed the specifications, 18% failed, and the remaining 50% of the points have not run. This coverage information about pass, failed, and uncovered points are also displayed in a tooltip when you place the mouse pointer on the coverage bar.

In the figure above, the requirement `swing` of test TRAN covered only 50% of the simulation points and 50% of the points have not been covered in the simulation run. However, the requirement `settlingTime` covers `10%` of passed points, `40%` of failed points, and `50%` of uncovered points. The coverage details are displayed in the Coverage Summary assistant. For more information, see Viewing the Coverage Summary.

The individual coverage report for the requirement is displayed in the *Coverage* column in the following colored bars:

- Indicates the pass percentage
- Indicates the fail percentage
- Indicates the percentage of points that are not covered in the simulation run.

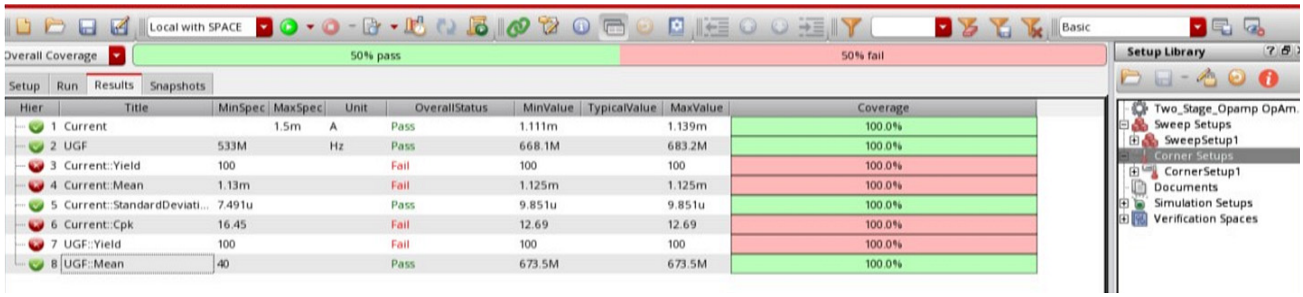
If the coverage report is not `100%`, indicating that it includes the uncovered points, recreate the ADE assembler setup using the project-specific setup from Setup Library assistant and then rerun the simulation. You can also run the simulations in ADE Verifier for the project-specific setup. For more information, see Running Simulations with Verification Spaces.

© 2023 All Rights Reserved.

Calculation of Coverage Percentage for Statistical Outputs

The statistical outputs from a Monte Carlo simulation in an implementation can be mapped to requirements. This allows a comprehensive and accurate calculation of coverage for the implementation.

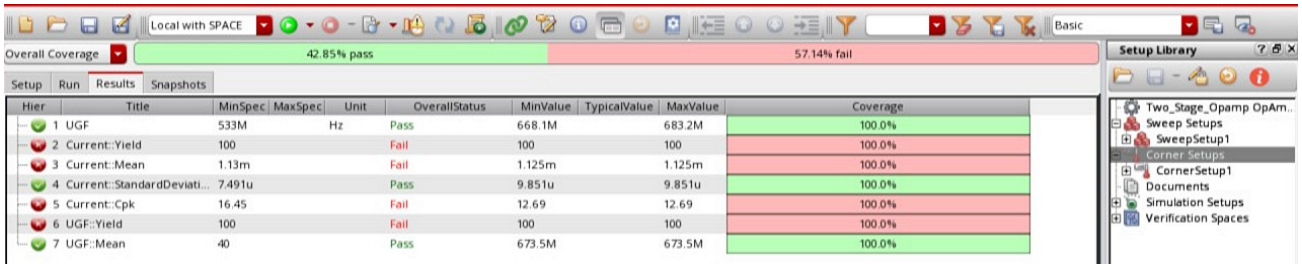
The following figure shows a setup with the nominal and statistical output of `Current` mapped to requirements.



Hier	Title	MinSpec	MaxSpec	Unit	OverallStatus	MinValue	TypicalValue	MaxValue	Coverage
1	Current		1.5m	A	Pass	1.111m		1.139m	100.0%
2	UGF	533M		Hz	Pass	668.1M		683.2M	100.0%
3	Current::Yield	100			Fail	100		100	100.0%
4	Current::Mean	1.13m			Fail	1.125m		1.125m	100.0%
5	Current::StandardDeviati...	7.491u			Pass	9.851u		9.851u	100.0%
6	Current::Cpk	16.45			Fail	12.69		12.69	100.0%
7	UGF::Yield	100			Fail	100		100	100.0%
8	UGF::Mean	40			Pass	673.5M		673.5M	100.0%

When the same verification space is specified for the requirements mapped to all outputs, the coverage of statistical outputs is the same as the coverage of the nominal outputs. This means that if the coverage for `Current(Nominal)` is `50%` and the same verification space has been set for requirements mapped to the `Current` outputs, then all statistical outputs for `Current`, such as `Current::StandardDeviation` or `Current::Yield` in the implementation also report `50%` coverage. The calculation of the coverage percentage for nominal outputs is based on the corner, sweep, or simulation setups specified in the Setup Library assistant and is applied to the related statistical outputs. Therefore, if the requirement `Current` is reported as `20%` pass, `30%` fail and `50%` uncovered, the requirement `Current::StandardDeviation` will be `50%`, where pass and fail statuses depend on the specification for the requirement, and `50%` uncovered. If your setup does not have nominal outputs, the *Overall Coverage* is based on the individual coverage of the statistical outputs.

The following figure shows a setup where the nominal output `Current` is not mapped to a requirement and therefore, not included in the results, but the coverage has been calculated for the statistical outputs.



Hier	Title	MinSpec	MaxSpec	Unit	OverallStatus	MinValue	TypicalValue	MaxValue	Coverage
1	UGF	533M		Hz	Pass	668.1M		683.2M	100.0%
2	Current::Yield	100			Fail	100		100	100.0%
3	Current::Mean	1.13m			Fail	1.125m		1.125m	100.0%
4	Current::StandardDeviati...	7.491u			Pass	9.851u		9.851u	100.0%
5	Current::Cpk	16.45			Fail	12.69		12.69	100.0%
6	UGF::Yield	100			Fail	100		100	100.0%
7	UGF::Mean	40			Pass	673.5M		673.5M	100.0%

© 2023
All Rights Reserved.

The *Coverage* column shows the color of the coverage percentage of requirements mapped to statistical outputs based on the *Overall Status*, where the passed requirements are shown in green and failed requirements are displayed in red. The overall status for the same kind of requirements is set after comparing its specifications with the simulation results of its statistical outputs.

Viewing the Coverage Summary

To view the coverage summary report to identify the setup values that have not been run for a particular block, choose one of these:

- Right-click the requirement in the *Results* tab and choose *View Coverage Summary*.

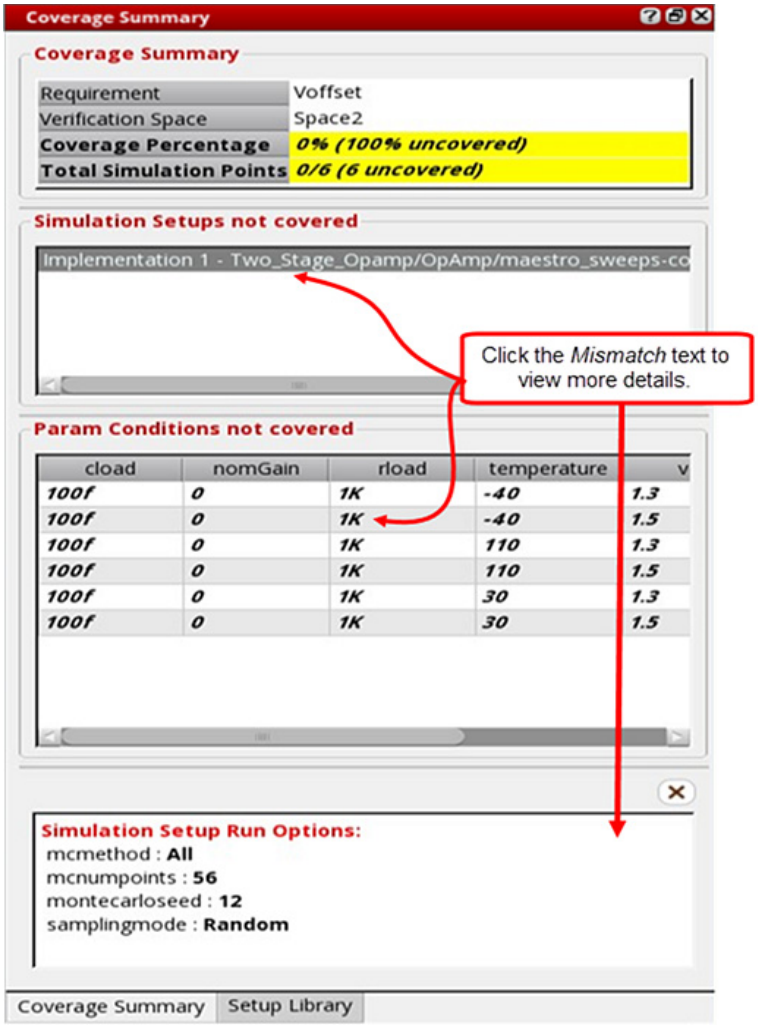
  This command is only displayed for requirements that have a verification space attached with them.

- Click *Open Coverage Summary Assistant* () on the toolbar.

  The Coverage Summary assistant appears. The following figure shows the coverage summary report for the requirement `Voffset`. The Coverage Summary assistant

© 2023 All Rights Reserved.

displays the coverage percentage for the selected implementation and also helps identify the project-specific setup values that were not run for the selected design block.



- The *Coverage Summary* section displays the following information:
  - *Requirement* – The name of the requirement for which you want to generate the coverage summary.
  - *Verification Space* – The name of the verification space that has been specified for the selected requirement.
  - *Coverage Percentage* – The total coverage percentage of the selected requirement against the project-specific setup. Coverage percentage is calculated on the basis of corners, sweeps, and simulation settings. The block-level setup must have the same values for simulation points as the Setup Library assistant. Otherwise, it reports the coverage percentage as `0%`, regardless of corners or

© 2023
All Rights Reserved.

sweeps. For example, the Coverage Summary displays the following information:
`Coverage Percentage: 0% (100% uncovered)`
`Total Simulation Points: 0/6 (6 uncovered)`

- *Total Simulation Points* – The total simulation points that are run, covered, and uncovered against the project-specific setup. The value of the simulation setup variable `mcnumpoints` (number of points) in the Setup Library assistant can be less than or equal to the block-level setup. However, the simulation setup is considered to have no mismatches only when all variable values defined in the setup library simulation setup are also less than or equal to the block-level setup values. If any variable does not pass this check, the simulation settings mismatch is calculated as `0%`. For example, the Coverage Summary displays the following information:
`Total Simulation Points: 0/6 (6 uncovered)`

- The *Param Conditions not covered* section displays a table of all the setup values given in the Setup Library assistant. The setup values that have not been run are displayed in bold. In addition, the table includes a *Model Files* column to indicate whether the model file defined in the corner setup of Setup Library is found for the specified requirement. If the model file is not found, a clickable word, `Mismatch`, is displayed that you can click to view more details.
- The *Simulation Setups not covered* section displays a list of only those implementations whose simulation settings do not match with the simulation settings of the verification space. The values that are a mismatch are displayed in bold. In addition, you can click the text in bold to view more details on the simulation setup variables.

When the Coverage Summary assistant is open, you can select the requirements on the *Results* tab and the coverage summary is updated in the assistant.

The coverage summary helps you compare the setup that is run at the block-level with the setup specified in the Setup Library assistant and ensure that all the specified values are run.

Exporting the Coverage Report

To export the coverage report in an Excel sheet or a CSV file,

➥ Choose *File – Export – CSV* or *Excel*.

The Export Requirements to CSV File or Export Requirements to Excel File form appears in which you can specify the filename. Ensure that the *Coverage* check box in the *What to Save* section is enabled.

Once the report is exported, you can add it in the *Documents* section of the Setup Library assistant.

© 2023 All Rights Reserved.

To add documents to the Setup Library cellview:

➡ Expand the *Documents* tree and click the `Click to add document` text.

The Choose document to be added form appears in which you can browse and select a document you want to add.



Once the CSV file is added in the *Documents* section, right-click the filename and choose one of the following options:

- *Open*–Select this option to open the CSV file.
- *Delete*–Select this option to delete the CSV file.
- *Save As*–Select this option to save a copy of the CSV file.

© 2023 All Rights Reserved.

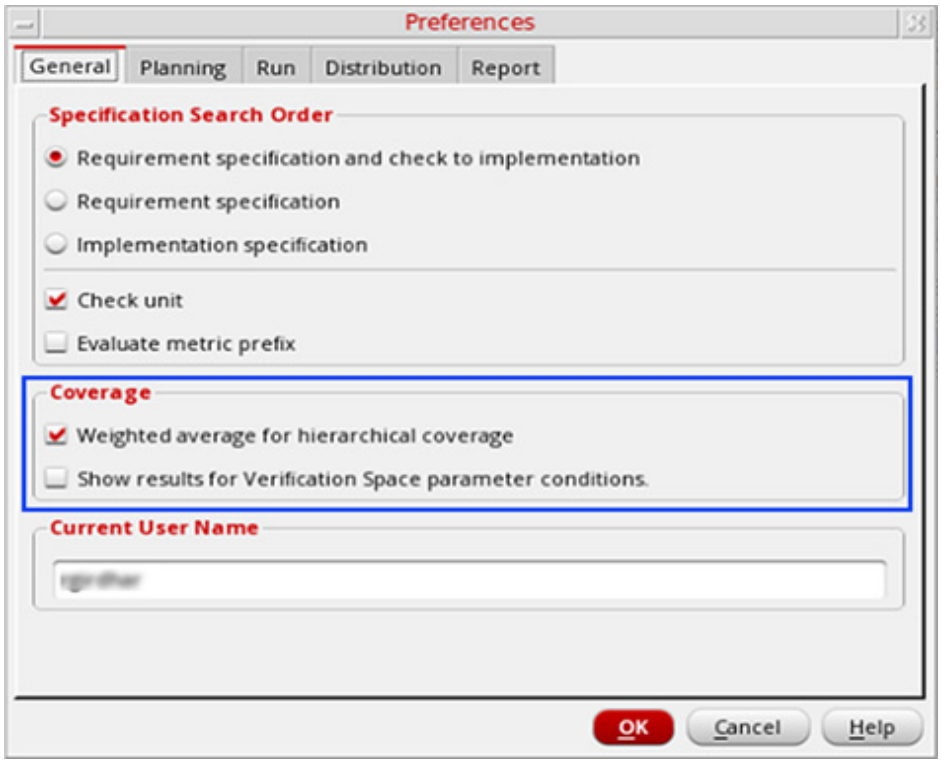
- *Refresh*–Select this option to refresh the CSV file to show the latest data.

Setting ADE Verifier Preferences for Coverage

To set the preferences to calculate and display the coverage:

➥ Choose *Edit – Preferences*.

The Preferences form appears.



In this form, in the *Coverage* section,

- Enable or disable the *Show 'Verification Space' on Setup tab and 'Coverage' column on Results tab by default* option if you want to control the default display settings of the *Verification Space* column on the *Setup* tab and the *Coverage* column on the *Results* tab. This option is selected by default.
- Enable or disable *Weighted average for hierarchical coverage* option to determine which calculation method is to be used to calculate the coverage–the weighted average of individual coverage in implementations, or the general average. This option is selected by default.

© 2023 All Rights Reserved.

8

Snapshots in Virtuoso ADE Verifier

In ADE Verifier, a snapshot contains the following information:

- Archive of the current requirements, mappings, and results as shown in the *Results* tab and stored in `snapshots.json` file

  For every implemention, the following details are saved:

  - Assembler setup database: `implementations/<name>/historyData.sdb`
  - Test states setup: `implementations /<name>/ historyData.state`
  - Extract of the results database information: `implementations /<name>/ historyData.rdb`

- Copy of the Verifier setup information (`settings.v3`)
- Backup of the loaded run summary data files in the `results` directory

  When the run summary data file exists in the `results` directory of the current cellview and the project directory, Verifier creates a backup of the last modified run summary data file. Verifier uses this backup file when restore a snapshot in the current cellview. For more information, see Location of Run Summary Data.

- Archive of the current implementations, where the overall information is saved in the `implementations.json` file, when the includeImplementations environment variable is set to `t`. For every implementation, a backup containing the following details is also saved in the `implementations/<impLib_impCell_impView_impHistory>` directory:

  - Assembler setup database file: `historyData.sdb`
  - XML state file that contains the details of all the tests: `historyData.state`
  - Extract of the results database information which includes design blocks and simulation files: `historyData.rdb`

Verifier saves snapshots in a zipped format in the `snapshots` directory of a Verifier cellview. You can track and analyze the verification progress of your project by using snapshots that

© 2023 All Rights Reserved.

can be created, modified or accessed using the Snapshots workspace and the *Snapshots* tab. Comparing the progress at required intervals helps you choose the best approach to fulfill the project goals and requirements. Additionally, snapshots help you backup and restore your project and Verifier environment information.

These snapshots of your verification progress contain status information on a date and time. You can track the progress over these timestamps and compare results at different stages in the verification cycle. Snapshots also serve as a time-stamped backups of your setup. Use these backups to restore your active setup to a previous state. By taking a snapshot of your verification project at early stages, you can ensure that all requirements are the same and prevent unintended changes. You can also verify that results, before and after major software, environment or PDK changes, are the same.

Related Topics

Snapshots User Interface in ADE Verifier

Snapshot Management in ADE Verifier

Comparison of Design and Simulation Setup Differences between Snapshots

Exporting Snapshot Comparison Differences

© 2023 All Rights Reserved.

Snapshots User Interface in ADE Verifier

The Snapshots user interface contains the following key elements:

- Snapshots Toolbar
- Snapshots Editor
- Snapshots Workspace
- Snapshots Tab

Snapshots Toolbar

The Snapshots toolbar is available to the left in the Verifier window. Use this toolbar to manage your snapshots.

The following table describes the main controls of the Snapshots toolbar.

Command	Tooltip	Description
	*Enabled*	Enables creating snapshots in the Verifier cellview. By default, it is deselected.
	*Create Snapshot*	Creates a snapshot of the current setup and results. By default, it is disabled. Selecting the *Enabled* check box enables this command.
	*Open / Close Snapshots Editor*	Opens or closes the Snapshots Editor.

Accessing the Snapshots Toolbar

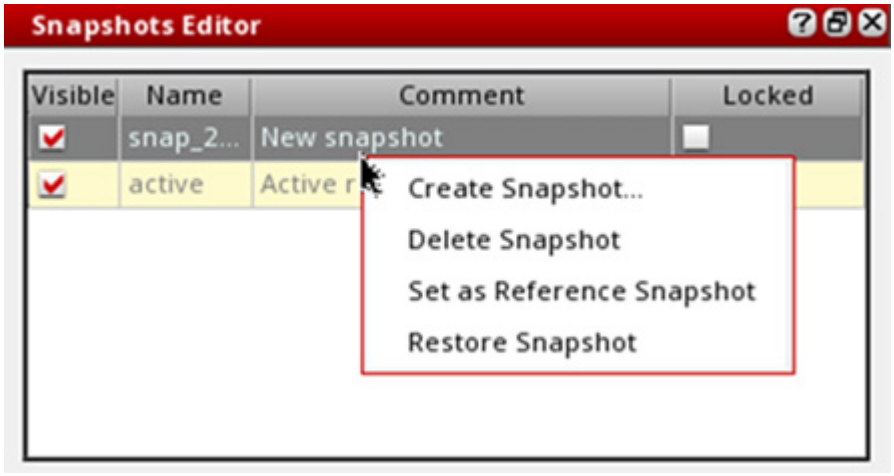
To access the *Snapshots* toolbar, do one of the following:

- Choose *Window – Toolbars – Snapshots*.
- Right-click anywhere on the Verifier toolbar and choose *Toolbars – Snapshots*.
- Choose *Window – Workspaces – Snapshots*.
- Choose *Snapshots* in the workspace list on the Verifier toolbar.

© 2023 All Rights Reserved.

Snapshots Editor

The Snapshots Editor lets you edit information for the snapshots that you create.



The following columns are available in the Snapshot Editor assistant:

Column	Description
Visible	Makes the snapshot visible in the *Snapshots* tab for comparison. The *active* snapshot is always visible.
Name	Shows the name of the snapshot. The default name is `snap_<yyyymmdd>_<hhmmss>`. Click the name of the snapshot to edit it.
Comment	Shows the comments that you add for a snapshot.
Locked	Locks the snapshot to prevent unattended removal. Deselect this check box to unlock the snapshot. To delete a locked snapshot, you need to unlock the snapshot first.

When you add a new snapshot, the Snapshots Editor displays the oldest snapshot in the top row and moves the active snapshot to the bottom of the snapshots list. Newly added snapshots are added above the active snapshot in the order of their creation time.

The following shortcut menu commands are available in the Snapshots Editor to let you manage snapshots in a Verifier cellview:

Command	Description
Create Snapshot	Creates a new snapshot and saves it in the `snapshot` directory.

© 2023 All Rights Reserved.

Command	Description
Delete Snapshot	Deletes the specified snapshot. You can delete only unlocked snapshots.
Set as Reference Snapshot	Sets the selected snapshot as the reference snapshot and highlights it in yellow. By default, a snapshot of the *active* setup is the reference snapshot.
Restore Snapshot	Restores the active setup to the specified snapshot.

Accessing the Snapshots Editor Assistant

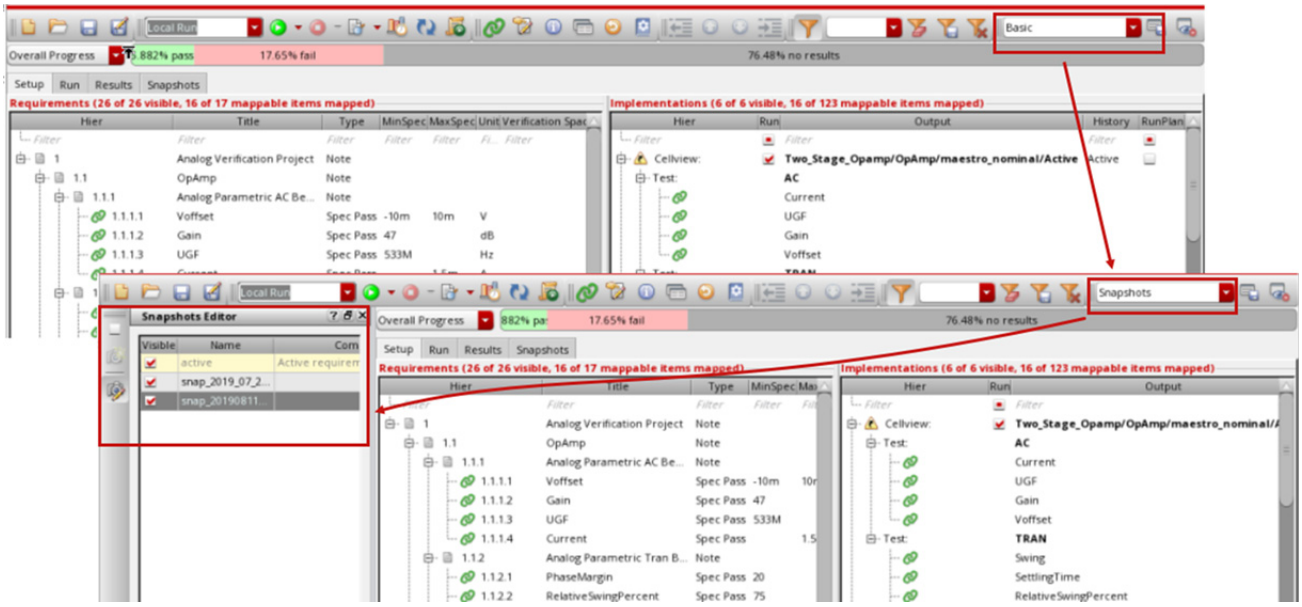
To access the Snapshots Editor assistant, do one of the following:

- Choose *Window – Assistant – Snapshots*.
- Right-click anywhere on the Verifier toolbar and choose *Assistant – Snapshots*.
- Choose *Window – Workspaces – Snapshots*.
- Choose *Snapshots* in the workspace list on the Verifier toolbar.

© 2023 All Rights Reserved.

Snapshots Workspace

The Snapshots workspace makes the Verifier environment ready to edit and manage snapshots by opening the Snapshots toolbar and the Snapshots Editor.



Accessing the Snapshots Workspace

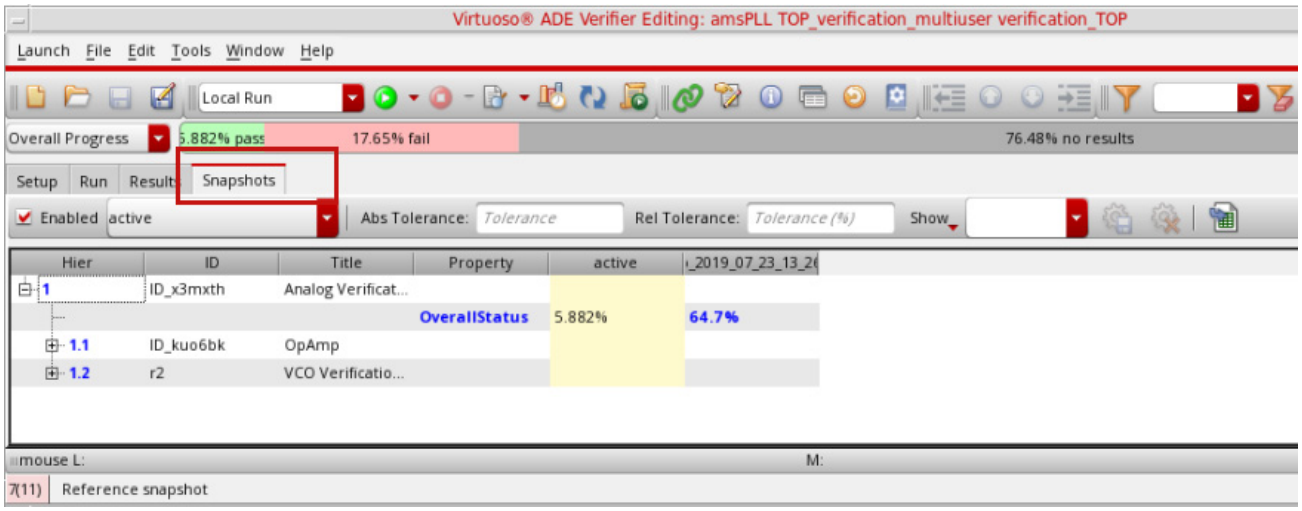
To open the *Snapshots* workspace, do one of the following:

- Choose *Window – Workspaces – Snapshots*.
- Choose *Snapshots* in the workspace list on the Verifier toolbar.

© 2023
All Rights Reserved.

Snapshots Tab

The following figure shows the *Snapshots* tab:



The *Snapshots* tab contains the following columns:

Column	Description
Hier	Shows the unique hierarchical number that Verifier assigns to a requirement to identify its placement in the requirements hierarchy.
ID	Shows the unique ID of the requirement.
Title	Shows the title of the requirement.
Property	Shows the property name of the requirement for which values are to be compared.
active	Shows the snapshot of the active setup. By default, this column is used as the reference snapshot.
snapshot<date_time>	Shows all snapshots that you set as *Visible* for comparison.

Additional Information

- You can remove a snapshot from the Snapshots tab by deselecting the *Visible* check box in the *Snapshots Editor* assistant.
- You can drag and drop the columns in the *Snapshots* tab to change their location. The *Hier* column is frozen to the left.

© 2023
All Rights Reserved.

- You can use the following shortcut menu commands in the *Snapshots* tab to expand or collapse the rows:
  - *Expand*: Expands all the requirements. When you select a specific requirement and click *Expand*, only the selected requirement expands.
  - *Collapse*: Collapses all the requirements. When you select a specific requirement and click *Collapse*, only the selected requirement collapses.

© 2023 All Rights Reserved.

Snapshot Management in ADE Verifier

You can perform the following operations with snapshots in the Verifier environment:

After creating multiple snapshots in Verifier, you can compare the requirement properties using the *Snapshots* tab. Based on the comparison, you can make changes to your verification setup.

- Adding or Removing a Snapshot
- Renaming a Snapshot
- Setting a Reference Snapshot
- Adding Comments to a Snapshot
- Changing the Order of Snapshots
- Restoring a Snapshot

Adding or Removing a Snapshot

To add a new snapshot:

➡ On the Snapshot toolbar, click *Create Snapshot*.

Observe that a new snapshot appears in the Snapshot Editor. The name of the snapshot is in the format `<snap_yyyymmdd_hhmmss>` where `yyyymmdd` shows the date and `hhmmss` shows the time. This ensures that each snapshot name is unique. For example, if you create a snapshot `snap_20190727_103806` and create another snapshot immediately, the next snapshot is saved as `snap_20190727_103807`. The new snapshot becomes available for comparison in the Snapshots tab only after you select the *Visible* check box for this snapshot. To create snapshots automatically using SKILL, see verifCreateSnapshot.

To remove a snapshot:

➡ On the Snapshot Editor, right-click a snapshot and choose *Delete Snapshot*.

Verifier deletes the selected snapshot.

© 2023 All Rights Reserved.

Renaming a Snapshot

To rename a snapshot:

1. On the Snapshot Editor, double-click the snapshot name.

   The snapshot name appears highlighted.

2. Enter a new name for the snapshot or edit the default name.
3. Press *Enter*.

   The new snapshot name appears in the Snapshot Editor.

Setting a Reference Snapshot

To set a snapshot as a reference:

➡ On the Snapshot Editor, right-click a snapshot and choose *Set as Reference Snapshot*.

Verifier sets the specified snapshot as the reference snapshot and highlights it with a yellow background. This is typically the snapshot that contains the correct specifications. Comparing values with the reference snapshot can help you to evaluate the changes made during the verification cycle.

Adding Comments to a Snapshot

To add comments to a snapshot:

1. On the Snapshot Editor, select a snapshot.
2. Double-click the *Comments* field and type in the comments.
3. Press *Enter*.

Changing the Order of Snapshots

When there are multiple snapshots in the ADE Verifier setup, you can change the display order of snapshots in the *Snapshots Editor* assistant. By default, when you create a new snapshot, it appears to the right of existing snapshots in the Snapshots tab and is set as *Visible*.

© 2023 All Rights Reserved.

To change the order of a snapshot:

1. Select a snapshot header.
2. Drag and drop it to the required column.

   The snapshot appears at the specified location.

Alternatively, deselect the *Visible* check box for all snapshots in the *Snapshots Editor*. This removes all snapshots from the *Snapshots* tab. Select the *Visible* check box for the snapshot that you want to appear first, and so on. The order in which the snapshots are displayed is determined by the order of selecting the *Visible* check box.

Restoring a Snapshot

To restore a snapshot:

➡ On the Snapshot Editor, right-click the snapshot from which you want to restore the setup and choose *Restore Snapshot*.

Verifier restores the setup from the specified snapshot. This overwrites the current setup. Using *Restore Snapshot* creates a snapshot with the suffix `pre_restore` as a precautionary measure. This is helpful when you have unsaved data for the date that you overwrite the active setup, or if you have accidentally chosen *Restore Snapshot*.

The *Restore Snapshot* command restores only the Verifier setup from `settings.v3` and run summary data files from the results directory. The schematic designs and ADE Assembler views are not saved as a part of the snapshot.

Exporting Snapshots

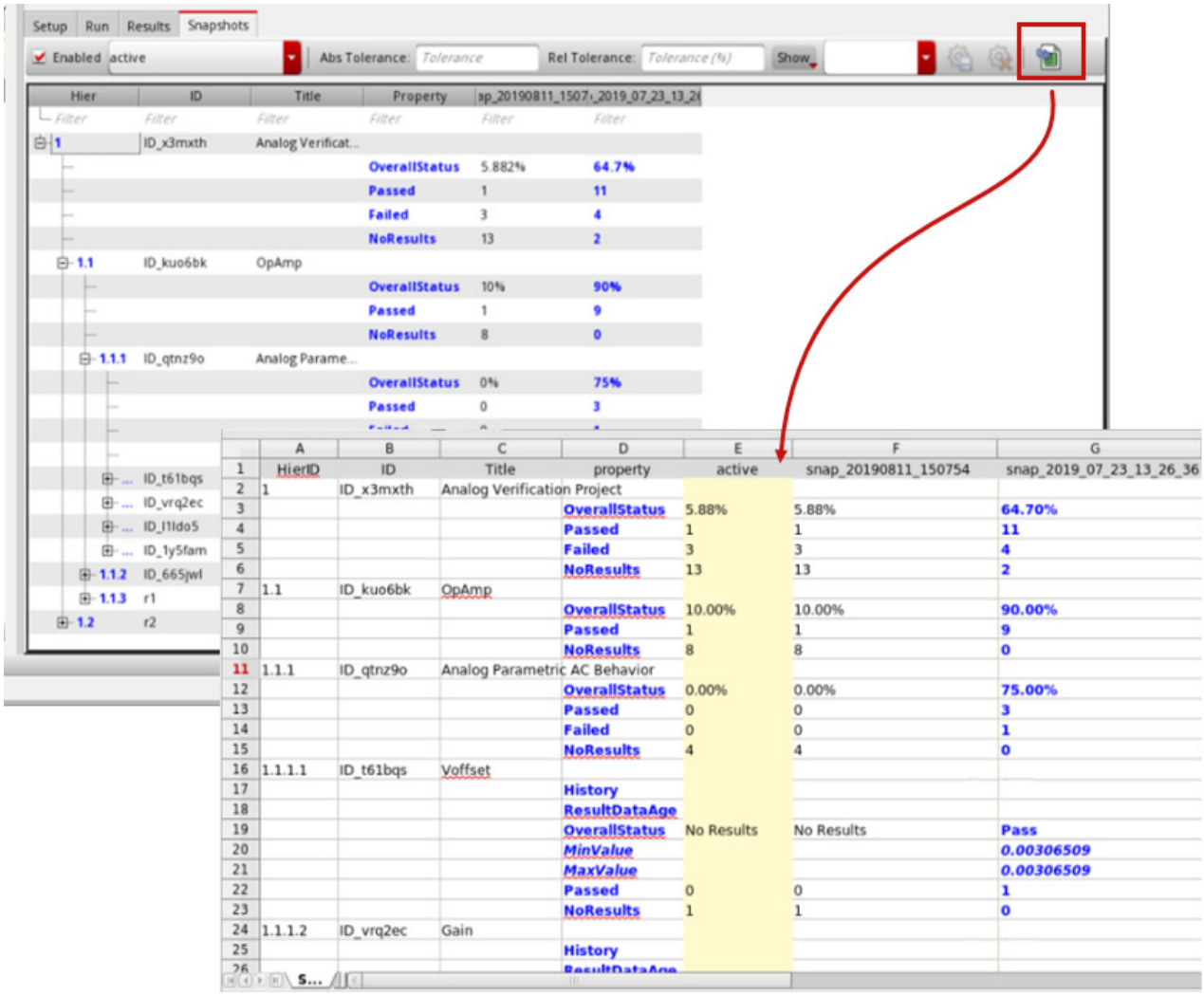
To export a snapshot:

➡ On the *Snapshots* tab, click the *Export Snapshot to Excel* command.

This command exports the snapshot comparison table to a Microsoft Excel file. The exported file contains only the snapshots that are set to *Visible*.

© 2023 All Rights Reserved.

The format of the data is identical in the *Snapshots* tab and the exported file. You can use the Excel file to analyze the snapshots data.



Related Topics

Adding or Removing a Snapshot

Renaming a Snapshot

Setting a Reference Snapshot

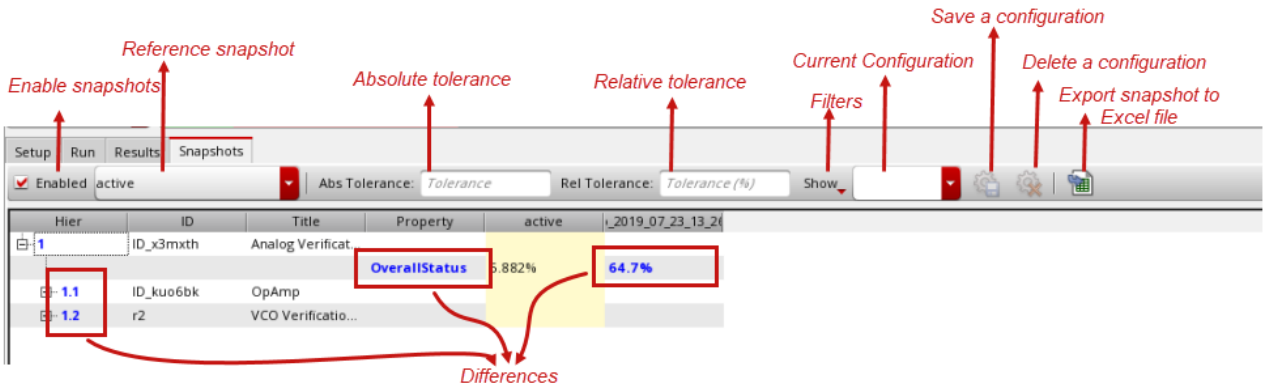
Adding Comments to a Snapshot

Changing the Order of Snapshots

© 2023
All Rights Reserved.

Snapshot Comparison

The rows in the *Snapshots* tab show the property values set for each requirement in the hierarchy. While comparing the snapshots, you can use the following features of this tab:



The following table describes the features that help with snapshot comparison:

Feature	Description
Enabled	Enables snapshots in the current Verifier session. If this check box is not selected, the Snapshots tab does not display any data.
Reference snapshot	Shows the name of the snapshot that is currently set as the reference. Change the reference snapshot by selecting any other snapshot from the list box. Alternatively, right-click a snapshot in the *Snapshots Editor* assistant and choose *Set as Reference Snapshot*.
Abs Tolerance	Allows you to specify the absolute tolerance for comparison. The Snapshots tab shows the property values based on the absolute tolerance.
Rel Tolerance	Allows you to specify the relative tolerance for comparison. The Snapshots tab shows the property values based on the relative tolerance.
Show	Allows you to specify property filters as criteria to show specified properties and hide other properties. Differences between property values and the reference are shown with blue highlight across the comparison table.
Table Configuration Name	Shows the name of the selected table configuration. Allows you to specify a new name for saving the current configuration.

© 2023
All Rights Reserved.

Feature	Description
Save Table Configuration	Saves the current table configuration with a name that you specify.
Delete Table Configuration	Deletes the selected table configuration.
Export Snapshots to Excel	Exports a snapshot of the verification setup to a Microsoft Excel file.

When comparing snapshots, Verifier uses only the data stored in a `.json` file. For more information on snapshots, see Snapshots in Virtuoso ADE Verifier.

© 2023 All Rights Reserved.

Property Filters in Snapshots

The Snapshots tab lets you apply multiple filters to make a comprehensive comparison. You can use the *Show* filter to select the properties for comparison.

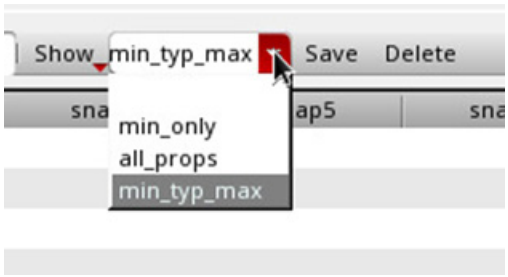


You can undock the *Show* list from the *Snapshots* tab and move it to a required location. This is particularly useful if you want to make multiple selections simultaneously in the list.

The list box to the right of the *Show* list contains the default configurations *Differences*, *Extended Results*, *Results*, *Setup Details*, and *Specs*. These configurations are installed in `/cdssetup/dfII/verifier/snapshotsConfigs/*.json`. Additionally, you can save the selected filters that you specify in the *Show* list box as a new configuration. You can

© 2023 All Rights Reserved.

save this configuration by specifying a name in the list box to the right of the *Show* list. The saved configuration contains the *Show* list settings as well as the tolerance settings. The new and updated configurations are saved in the `.cadence` directory of the current directory. If you delete a configuration, it is removed from the current session in memory and the `.cadence` directory of the current directory. If you save the configuration to a different location, it is shown in the list box the next time you open the cellview.



The available filters in the *Show* list are:

- *Differences*: Shows only those property values for requirements that have a difference or have a child requirement with a difference.
- *Compare Visible Properties*: Uses only the visible properties of requirements for comparison. If you deselect this filter, it is possible that the comparison in the *Snapshots* tab shows differences in some requirements but the required property is not visible.
- *Hide Empty Rows*: Hides the rows that do not have property values. All empty rows are hidden by default.
- *Show all Properties*: Shows all properties for requirements.
- *Hide all Properties*: Hides all properties for requirements.
- *Property List*: Shows the rows with the selected property. You can select multiple properties.

You can easily set up and save different configurations for the *Snapshots* tab. You can quickly generate configuration setups for different comparison scenarios using the different filters and properties.

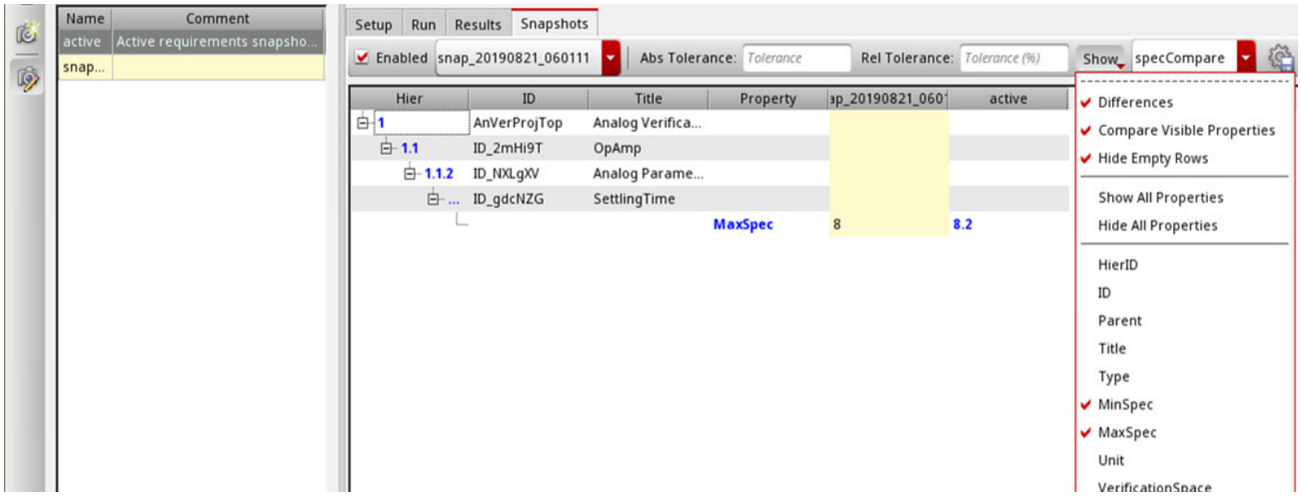
Case 1: You have a Verifier cellview that contains an ideal set of specifications. You create a snapshot and define it as a reference snapshot. You might then run simulations and create new snapshots. To ensure that the required specifications are unmodified during the project run time, you compare all snapshots by selecting the following configuration on the *Snapshots* tab:

- *Show – Differences*

© 2023 All Rights Reserved.

- *Show – Compare Visible Properties*
- *Show – Hide Empty Rows*
- *Show – MinSpec*
- *Show – MaxSpec*

The following figure shows an example of a comparison using the above configuration.



The *Snapshots* tab displays no results if the specifications remain the same. You only see results when a property specification is modified. Here, *Compare Visible Properties* is useful because you want to compare only specifications.

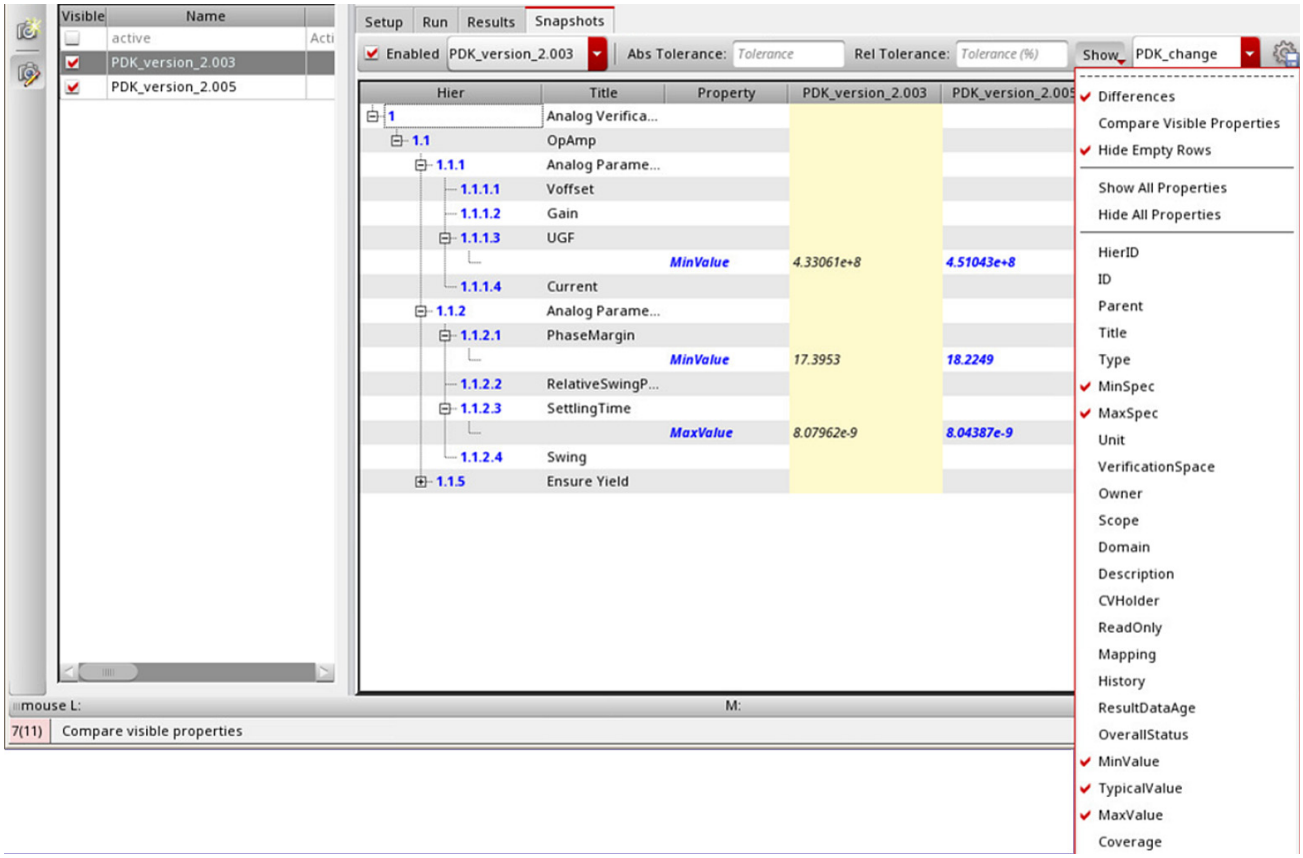
Case 2: You have a PDK upgrade scheduled and you want to ensure that there are no changes in result values before and after this upgrade. You run simulations and take a snapshot before the PDK upgrade. After the upgrade, you run simulations again and take another snapshot.

To check for any change in simulation results before and after the upgrade, you select the following configuration on the *Snapshots* tab:

- *Show – Differences*
- *Show – Hide Empty Rows*
- *Show – MinValue*
- *Show – TypicalValue*
- *Show – MaxValue*

© 2023
All Rights Reserved.

The following illustration shows an example of a comparison of simulation results that are saved before and after PDK upgrade using the above configuration.

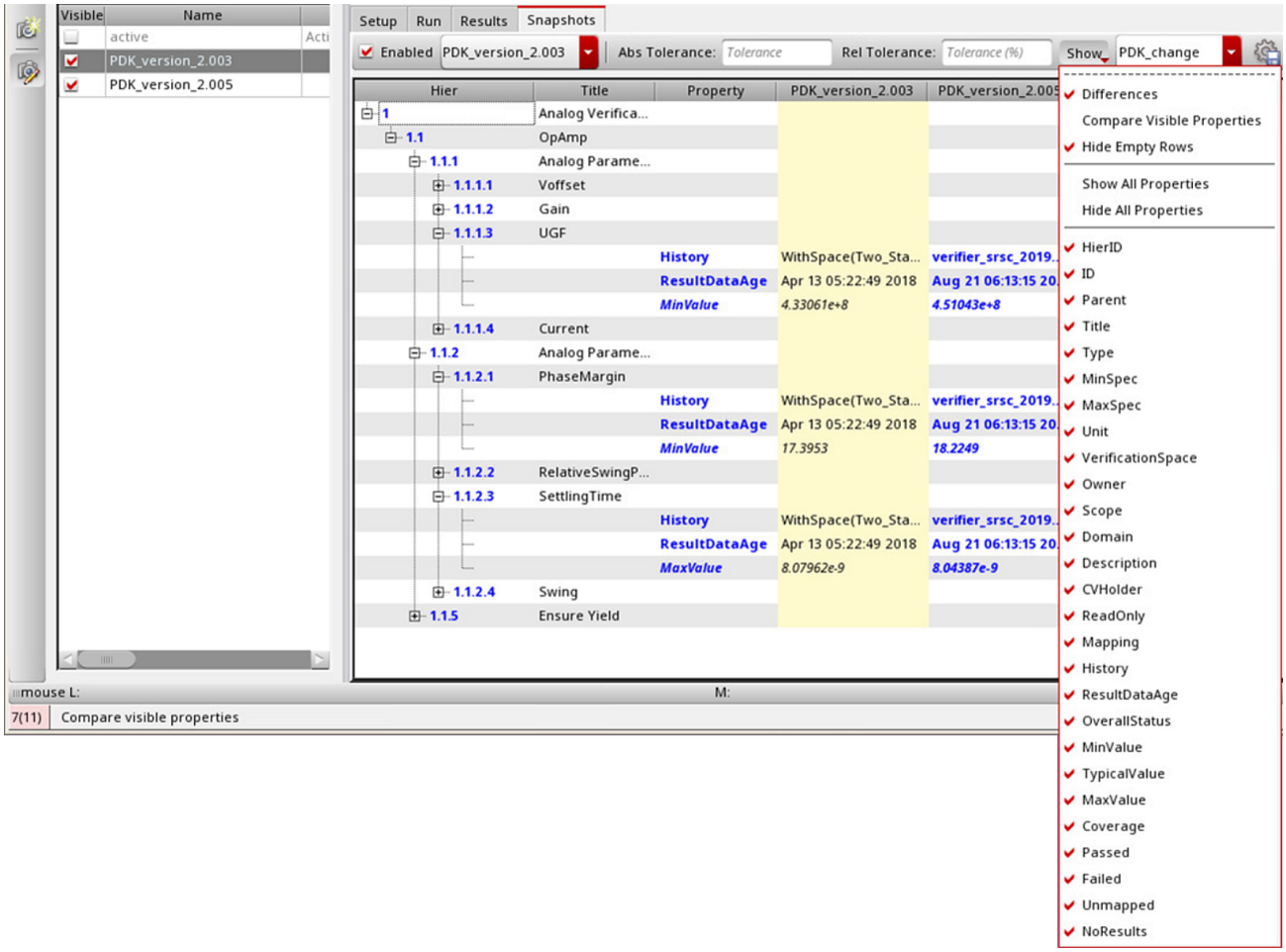


Observe that *Compare Visible Properties* is not selected. This means that the comparison is done for visible and non-visible properties, and highlights differences for even those properties which are not selected in the current configuration.

To make other comparisons in property values, select the property in the *Show* list. The *Snapshots* tab dynamically updates with the additional property values. To analyze changes in all property values, choose *Show – Show All Properties*.

© 2023
All Rights Reserved.

The following figure shows a comparison of simulations results that are saved before and after PDK upgrade, for all properties.



The above illustration shows all properties as selected but highlights only the property values for *History* and *ResultDataAge* as differences. This implies that only these two properties have differences after the PDK upgrade.

Additionally, you might want to use tolerance settings to filter out minor changes in the result values.

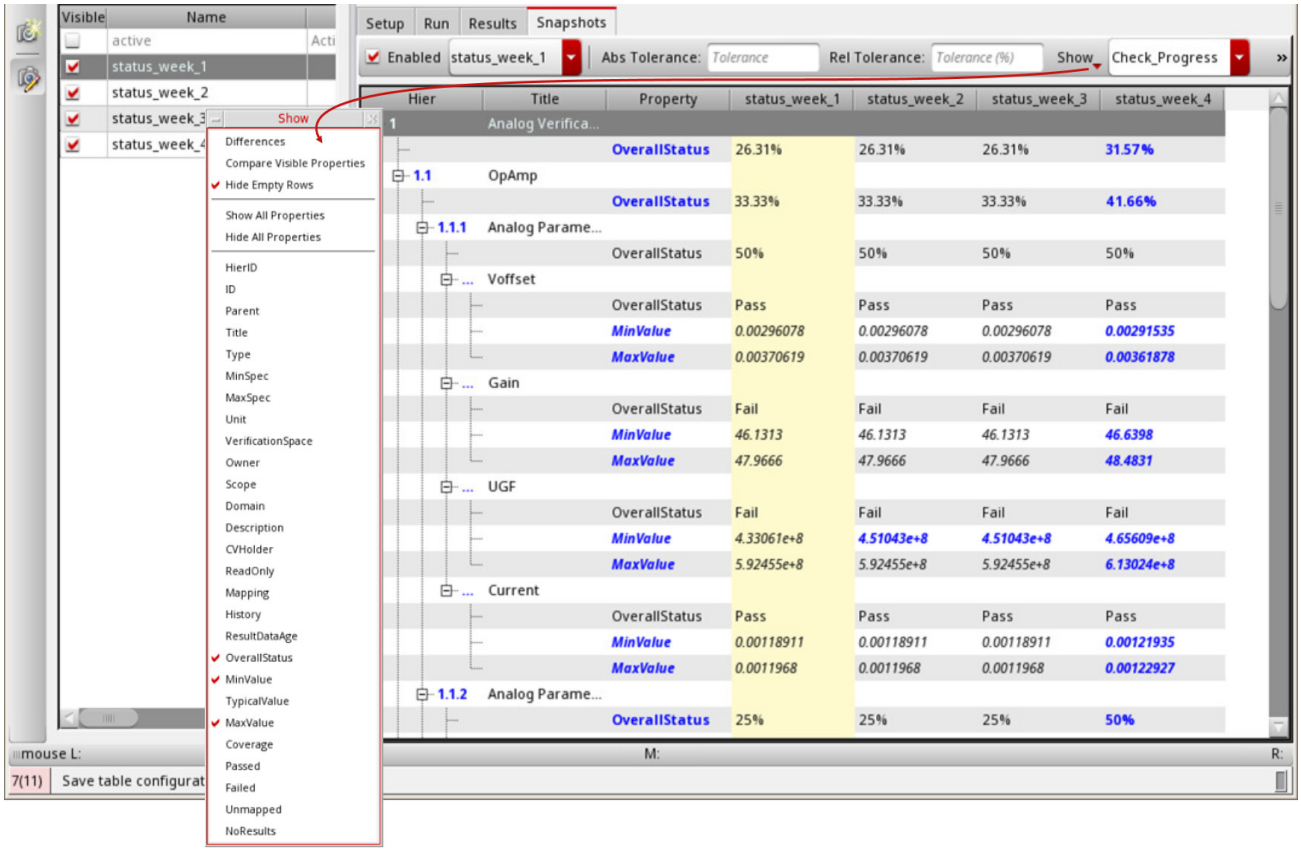
Case 3: You want to monitor the progress of your project over different time intervals, and you take a snapshot every Friday with the following configuration:

- *Show – Hide Empty Rows*
- *Show – OverallStatus*
- *Show – MinValue*

© 2023 All Rights Reserved.

- *Show – MaxValue*

The following figure shows an example of a comparison of specifications and results with the above configuration:

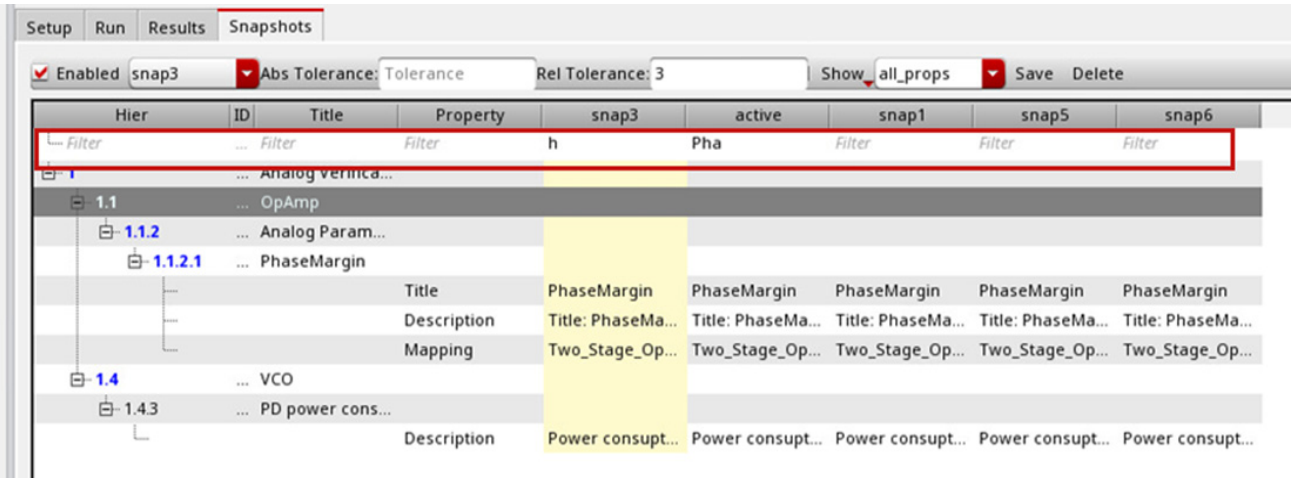


Observe that this configuration does not include *Differences* and *Compare Visible Differences* because you want to see all the values for all available snapshots. This configuration shows the values of the selected properties even if they have not changed in comparison to the reference snapshot.

You can use *Export Snapshots to Excel* on the *Snapshots* tab to visualize the progress graphically.

© 2023
All Rights Reserved.

In addition to the above filters, use the *Filter Rows* command from the Verifier toolbar to display requirements or values based on the specified filter text.

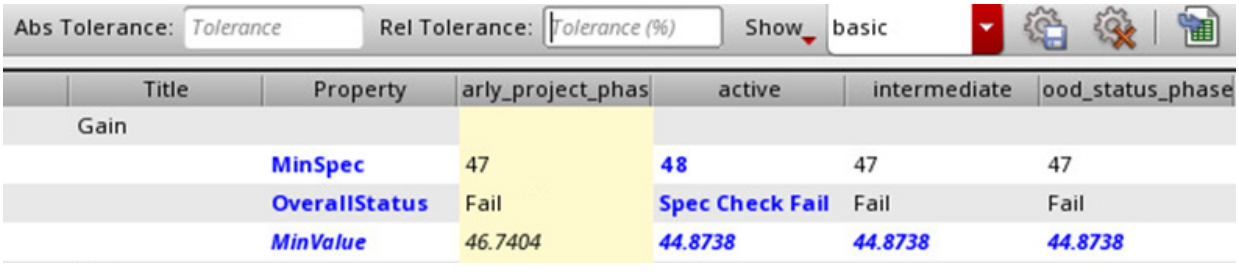


Application of Tolerance Values in Snapshots

When you specify tolerances, the *Snapshots* tab shows the differences in property values of available snapshots after considering the specified tolerance. When you do not specify a tolerance value, the *Snapshots* tab highlights the specification value differences.

The following figures show the changes in the Snapshots tab when you apply tolerances:

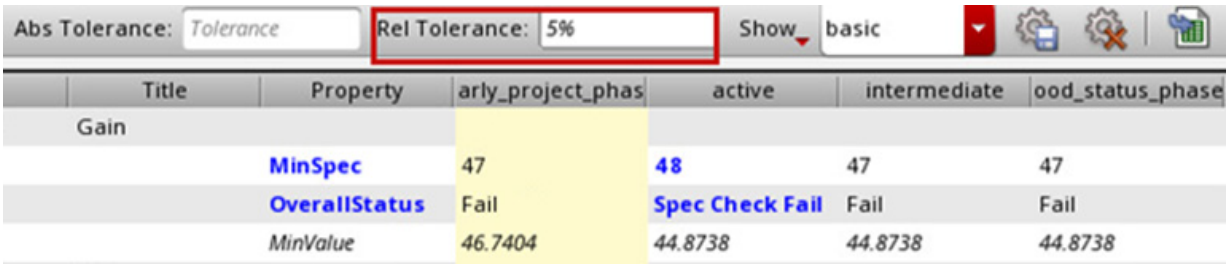
- No tolerance:



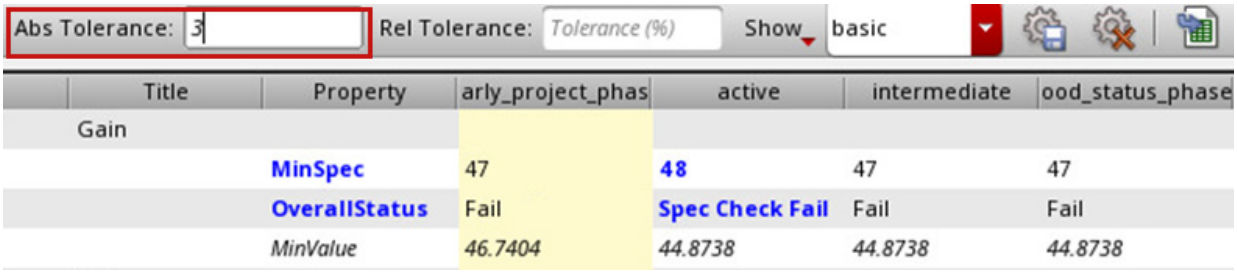
  Differences in specification values are always highlighted.

© 2023
All Rights Reserved.

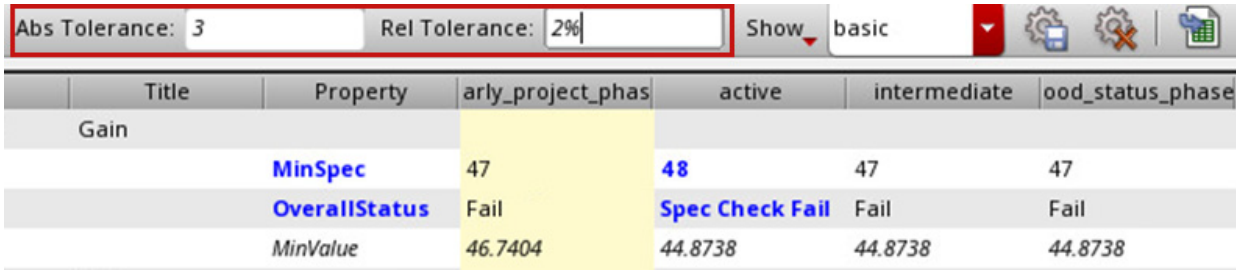
- Relative tolerance:



- Absolute tolerance:

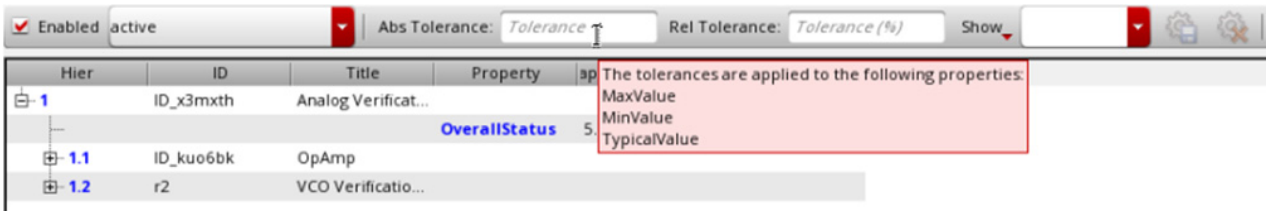


- Relative and absolute tolerance:



Note: Using *Save table configuration* ( )also saves the specified tolerances.

The *Abs Tolerance* and *Rel Tolerance* fields on the *Snapshots* tab and the rows in the *Snapshots* table with tolerance values applied appear in italics. The tolerance fields also show tooltips listing the properties to which the tolerances are applied.

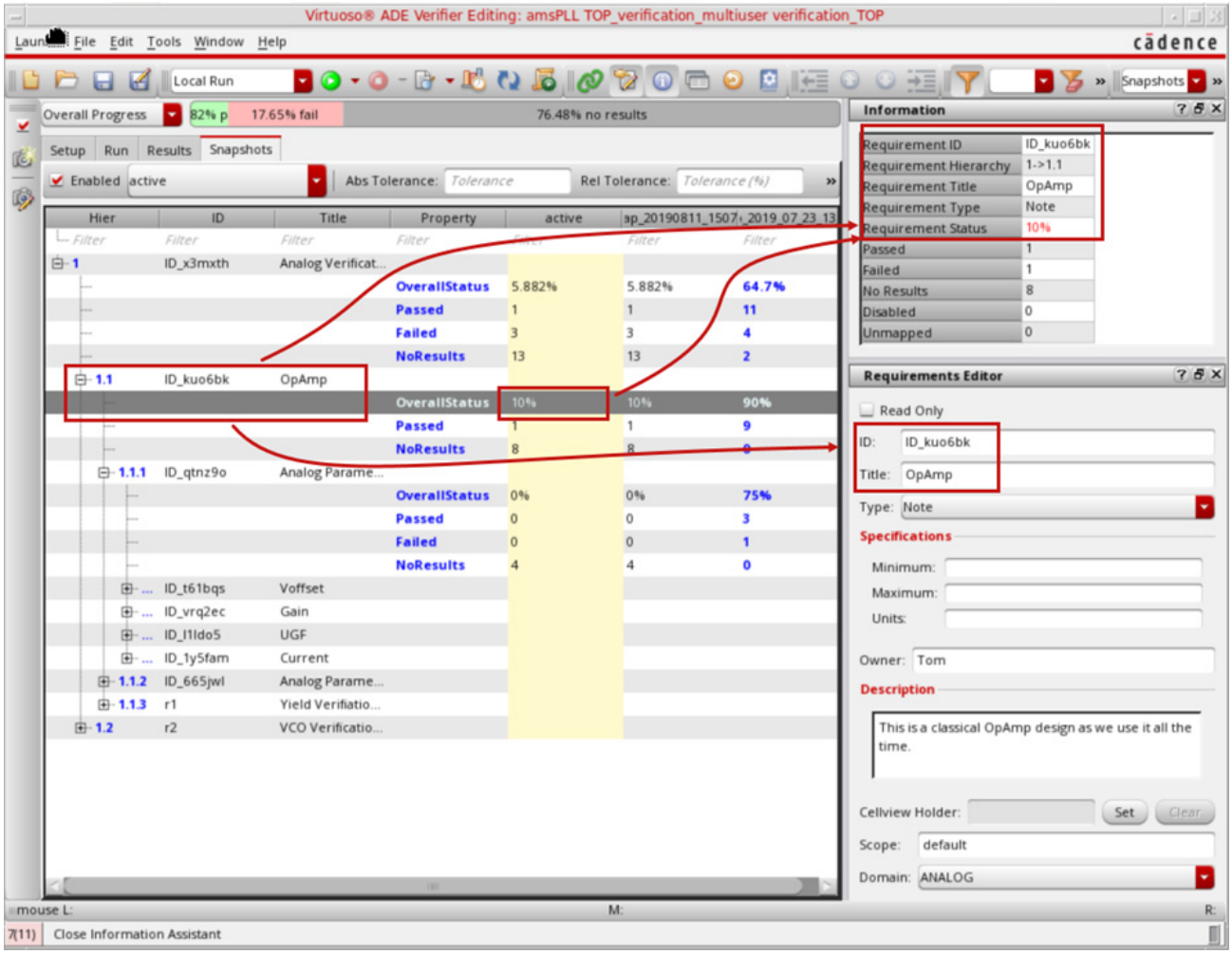


© 2023 All Rights Reserved.

By default, tolerances are applied only to `MinValue`, `TypicalValue`, and `MaxValue`. Other properties, such as the `MinSpec` and `MaxSpec` values, are compared without applying tolerances. To make changes to the properties that have tolerances applied, use the environment variable toleranceProperties.

Modification of the Active Setup in the Snapshots Workspace

Snapshots are specially useful when you can modify your active setup based on the snapshot comparison results. When you select a row on the *Snapshots* tab, the *Information* and *Requirements Editor* assistants show the active setup details corresponding to the selected row. The *Information* assistant shows read-only details of the selection. The *Requirements Editor* lets you modify the active setup values for the selection in the *Snapshots* tab.



© 2023
All Rights Reserved.

Comparison of Design and Simulation Setup Differences between Snapshots

Using snapshots, you can track the verification status of a project over a period. But the information in snapshots can change because of several factors, such as a change in the design, maestro setup, model files, verification plan, or mapping.

The *Compare Implementations from Snapshots* feature in ADE Verifier lets you save the implementations into snapshots and then lets you compare the implementations from these snapshots.

Prerequisites for Snapshot Comparison

The comparison of snapshots is available with the following conditions:

- Additional design and ADE Assembler setup data for implementations are saved within snapshots.

  You can enable this by setting the `includeImplementations` environment variable to `t`.

- Separate history management is enabled in the maestro simulation setup.

  You can enable this by setting the `useSeparateHistoryFileManagement` environment variable to `all`.

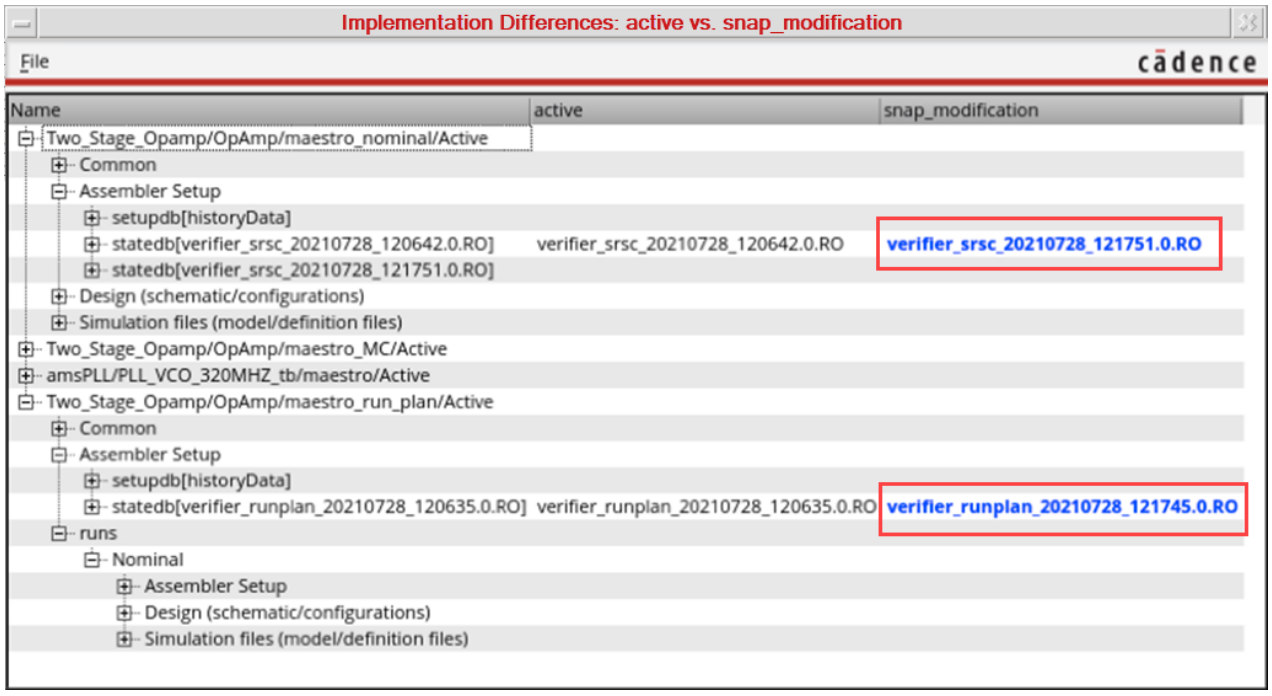
Comparing Design and Simulation Setup Differences

To compare a snapshot against the reference snapshot:

➥ Right-click a value in the snapshot that you want to compare and select *Compare Implementations from Snapshots*.

© 2023 All Rights Reserved.

The Implementation Differences window opens.



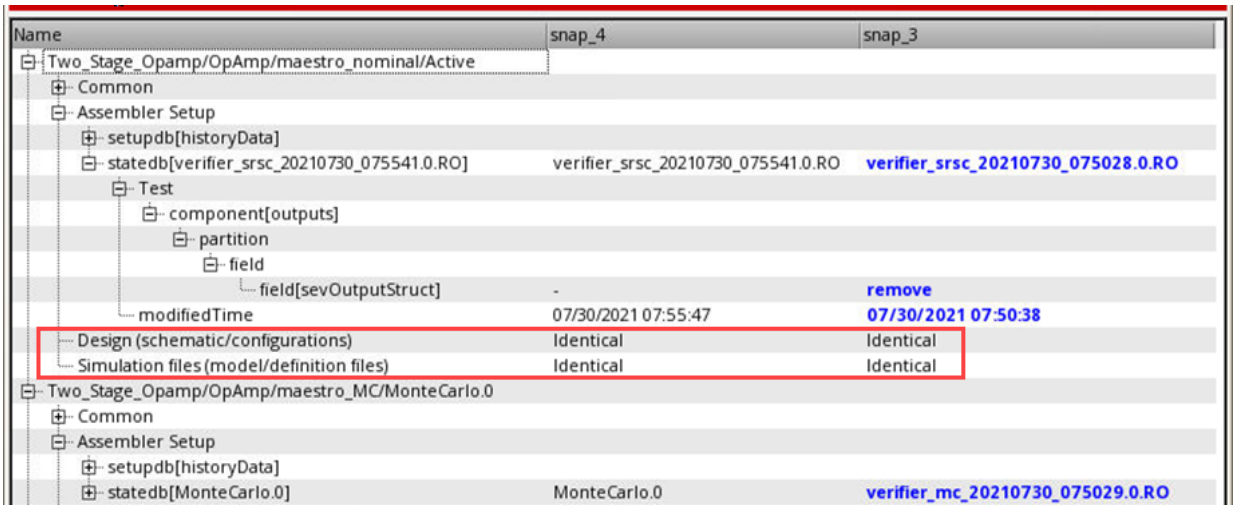
The *Compare Implementations to Snapshots* command is disabled when you right-click anywhere on the reference snapshot column.

The window displays a detailed and categorized comparison of the information that has been saved in the snapshot under the following sections:

- `Common`: Includes the basic information which is common to both snapshots. No comparison is done for this section.
- `Design`: Includes the schematic and configuration details

© 2023
All Rights Reserved.

- `Simulation Files`: Includes the model files and definition files



The differences that are found in these sections are displayed in blue text. If there are no differences in a section, the window displays *Identical* in both the snapshot columns. Additionally, the simulations files are compared using an MD5 hash, and the design files are compared using the OA change counter.

The first column displays the name of the implementation, which is followed by a hierachical categorization of the compared information. While the *Name* column is automatically resized to fit contents, the other columns can be resized manually, as required.

Ignoring Elements or their Order when Comparing Implementations from Snapshots

When comparing implementations within snapshots, there may be some elements that you want to exclude from the comparison because they are not important for the verification status.

You can choose to ignore these elements from your design or simulation setup by doing the following:

- To control the sensitivity of the order of differences during comparison, use the impElementOrderSensitivity environment variable.
- To specify that disabled elements are to be ignored during implementation comparison, use the ignoreDisabledElements environment variable.
- To specify the elements to be ignored during the comparison of *.sdb files, use the ignoreImpSetupElements environment variable.

© 2023 All Rights Reserved.

- To specify the elements to be ignored during the comparison of *.state files, use the ignoreImpStateElements environment variable.

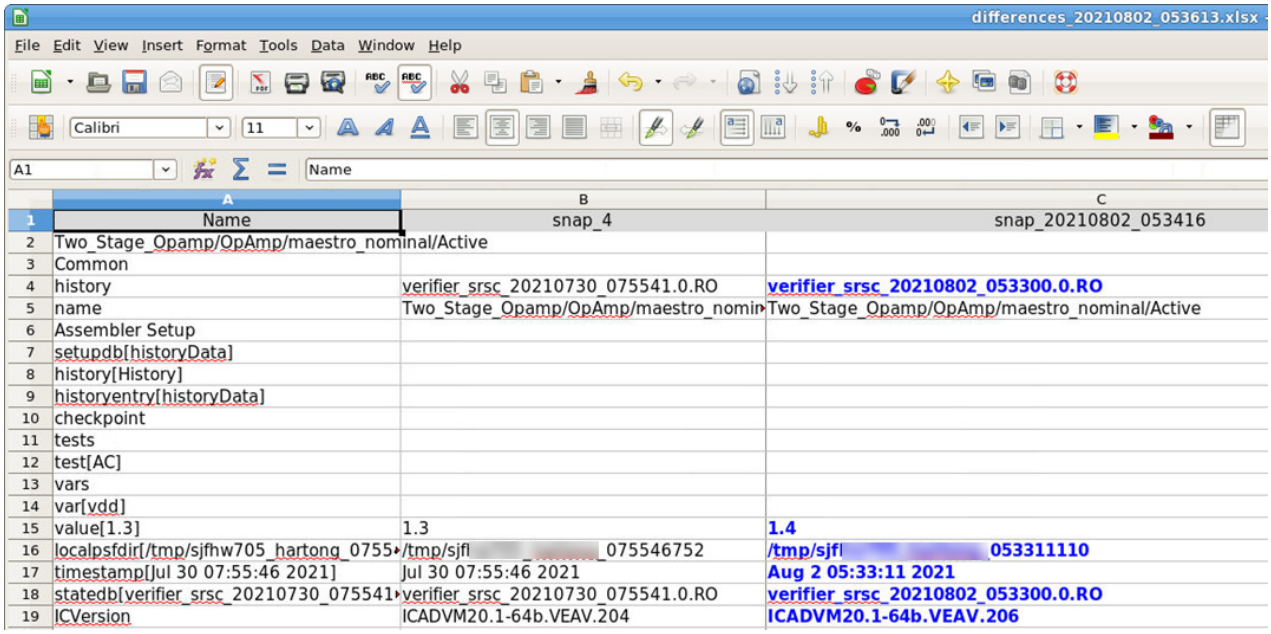
Unique or fundamentally different elements are ignored by default. For example, `uuid` or `history_name`.

Exporting Snapshot Comparison Differences

The differences found during snapshot comparison can be export to a Microsoft Excel file. To do this:

➥ In the Implementation Differences window, choose *File – Export Differences to Excel*.

The implementation differences are exported to a Microsoft Excel file. By default, the name of the exported file is `differences_<date>_<time>.xlsx`. It retains the same display format as the Implementation Differences window, and all differences are highlighted in blue text.



	A	B	C
1	Name	snap_4	snap_20210802_053416
2	Two_Stage_Opamp/OpAmp/maestro_nominal/Active		
3	Common		
4	history	verifier_srsc_20210730_075541.0.RO	verifier_srsc_20210802_053300.0.RO
5	name	Two_Stage_Opamp/OpAmp/maestro_nomin	Two_Stage_Opamp/OpAmp/maestro_nominal/Active
6	Assembler Setup		
7	setupdb[historyData]		
8	history[History]		
9	historyentry[historyData]		
10	checkpoint		
11	tests		
12	test[AC]		
13	vars		
14	var[vdd]		
15	value[1.3]	1.3	1.4
16	localpsfdir[/tmp/sjfhw705_hartong_0755	/tmp/sjfl _ _075546752	/tmp/sjfl 053311110
17	timestamp[Jul 30 07:55:46 2021]	Jul 30 07:55:46 2021	Aug 2 05:33:11 2021
18	statedb[verifier_srsc_20210730_075541	verifier_srsc_20210730_075541.0.RO	verifier_srsc_20210802_053300.0.RO
19	ICVersion	ICADVM20.1-64b.VEAV.204	ICADVM20.1-64b.VEAV.206

© 2023 All Rights Reserved.

© 2023 All Rights Reserved.

9

External References in Virtuoso ADE Verifier

Verifier lets you set up an analog design verification project with multiple cellviews. You can import existing Verifier cellviews as external references, and maintain your master cellview.

The master cellview maintains the verification plan, also referred to as the requirements hierarchy, by creating references to external cellviews. In their local cellviews, designers add implementations and map them to their requirements. This methodology offers flexibility to the requirement owners, and provides up-to-date details to the verification project manager.

This topic provides information on how you can manage a design verification project using Virtuoso ADE Verifier (Verifier) that supports multiple Verifier cellviews through references.

- Flow for Referencing External Cellviews
- Importing or Accessing Cellviews as References
- Hierarchy for Referenced Cellviews
- Editing or Updating Broken External References
- Multiple Hierarchy Levels in Multi-User Verification Projects
- Synchronization Between the Master and Referenced Cellviews
- Simulation Runs on Referenced Cellviews

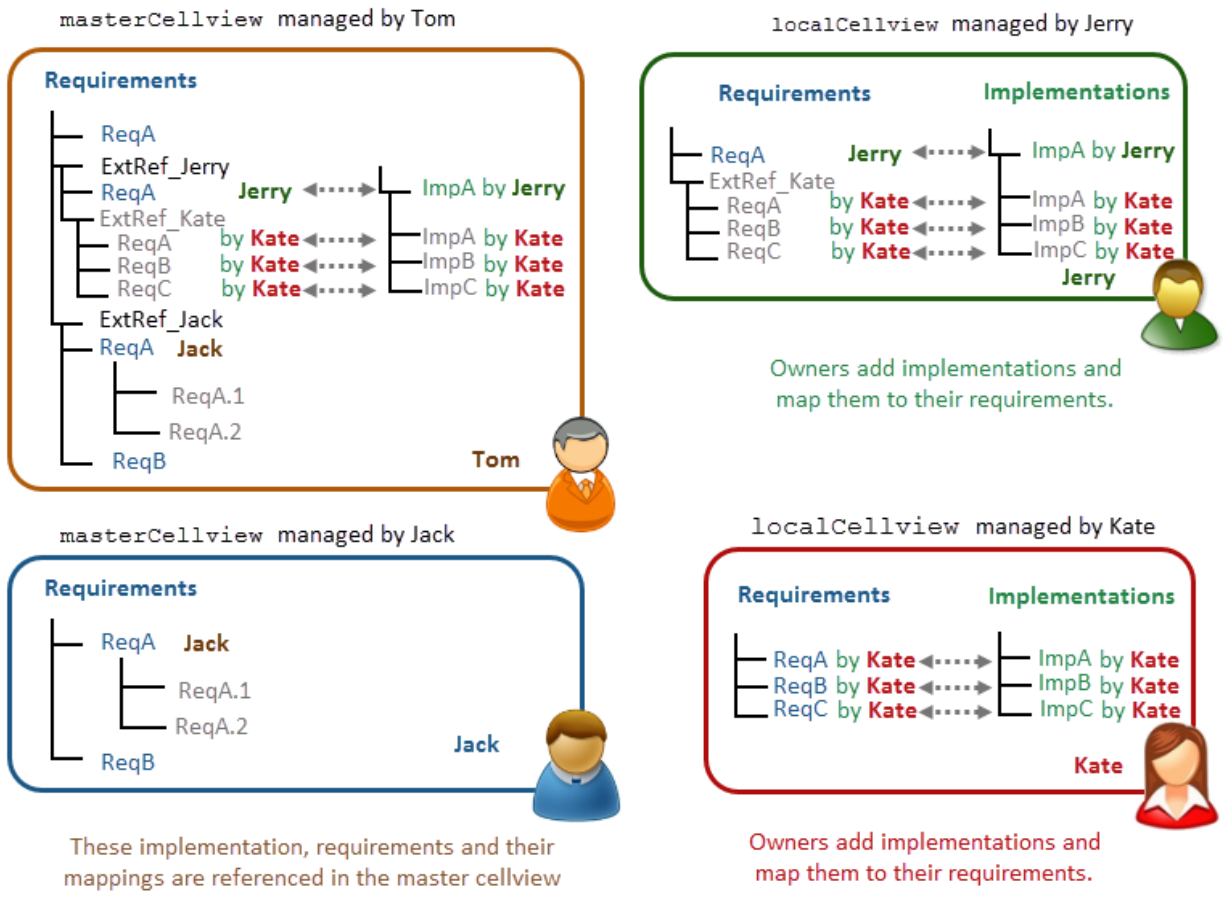
© 2023 All Rights Reserved.

Flow for Referencing External Cellviews

As a verification project manager, you can reference cellviews containing requirements and implementations when multiple designers are involved in the project. The designers have their local cellviews containing their own set of requirements and implementations. You can then import these specific cellviews from the designers, and maintain your master cellview.

The designers add implementations and map them with their requirements in their local cellviews. These requirements and implementations become available to the master cellview through references. The verification project manager can view the latest verification status of requirements and their implementations from each designer.

The following figure shows how you can manage a design verification project with multiple references to other cellviews.



Multi-level hierarchy in a design verification project

© 2023
All Rights Reserved.

The following steps highlight the flow for managing a new analog design verification project with multiple references to external cellviews.

1. The master cellview references the external cellviews for capturing the implementations and mappings that the designers add.

   See Importing or Accessing Cellviews as References and Adding Requirements.

2. The verification manager moves the external reference up or down in the hierarchy based on project specifications. See Hierarchy for Referenced Cellviews.

3. The external references in the master cellview maintain the mappings between the requirements and implementations. Any changes in the owner cellview break the reference in the master cellview, making it invalid, and the mappings between the requirements and implementations are lost.

   See Editing or Updating Broken External References.

4. The requirement owners add implementations in their respective local cellviews and map them to their requirements. Owners can also run simulations and monitor the verification status of the requirements they own. The project manager monitors the progress of the verification project using the master cellview.

   See Synchronization Between the Master and Referenced Cellviews.

5. The project manager can run simulations on referenced cellviews and manage the verification project.

   See Simulation Runs on Referenced Cellviews.

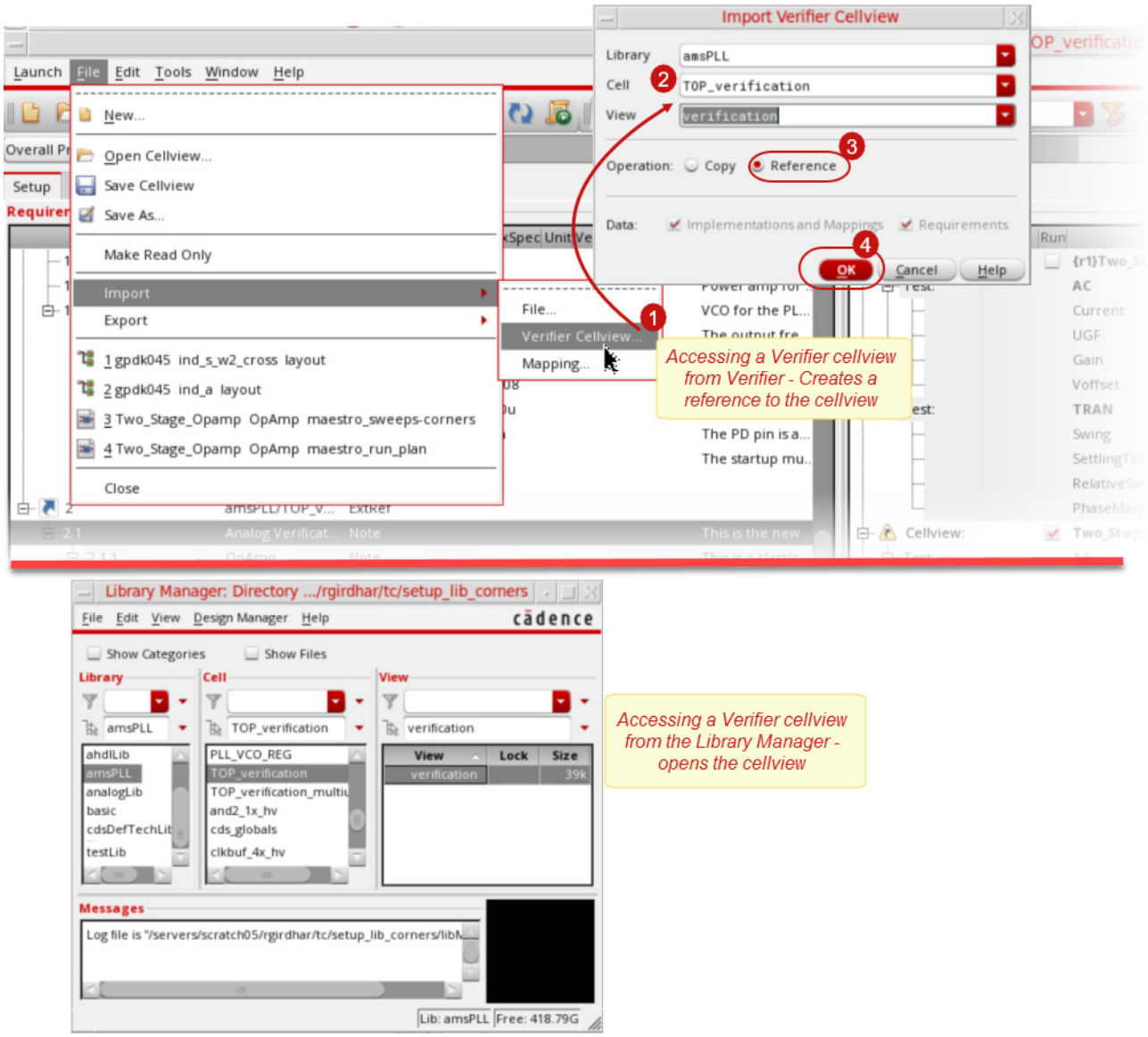
Importing or Accessing Cellviews as References

The project manager can successfully analyze the verification setup when requirements and implementation details are available from all designers involved in the project. To ensure that the overall verification status of the project can be analyzed accurately, the project manager references the cellviews of the designers in the master verification setup.

The verification project manager typically references the external cellviews directly from the master cellview. Designers typically access their local cellviews from Library Manager.

© 2023 All Rights Reserved.

The following figure illustrates how you can access external cellviews.



To access a cellview as an external reference from the master cellview opened in Verifier, do one of the following:

- Choose *File – Import – Verifier Cellview*
  - Choose the *Reference* operation.
  - Click *OK*.
- Right-click anywhere in the *Requirements* pane, and choose *Import Cellview*.

© 2023
All Rights Reserved.

- Choose the *Reference* operation.
- Click *OK*.

The referenced cellview gets added to the current Verifier session with the type *ExtRef*.

The shortcut menu options for external references are as follows:

Option	Description
Edit Reference	To change the specified Verifier cellview.
Open Reference	To open the source Verifier cellview.
Update Reference	To reload the external cellview details and replace the existing referenced requirements, mappings and implementations with updates from the specified Verifier cellview.
Delete Requirements	To delete the external reference and the requirements, mappings and implementations imported with the reference.

Additional Information

- You can only import a cellview as a reference when the current session is editable.
- You can specify the default filters for referenced cellviews using the environment variable filterOptions. These options show as selected when you right-click in the *Requirements* pane and choose *Show*.

For more information on External References, see Import Requirements from External Sources.

Related Topics

Import Requirements from External Sources

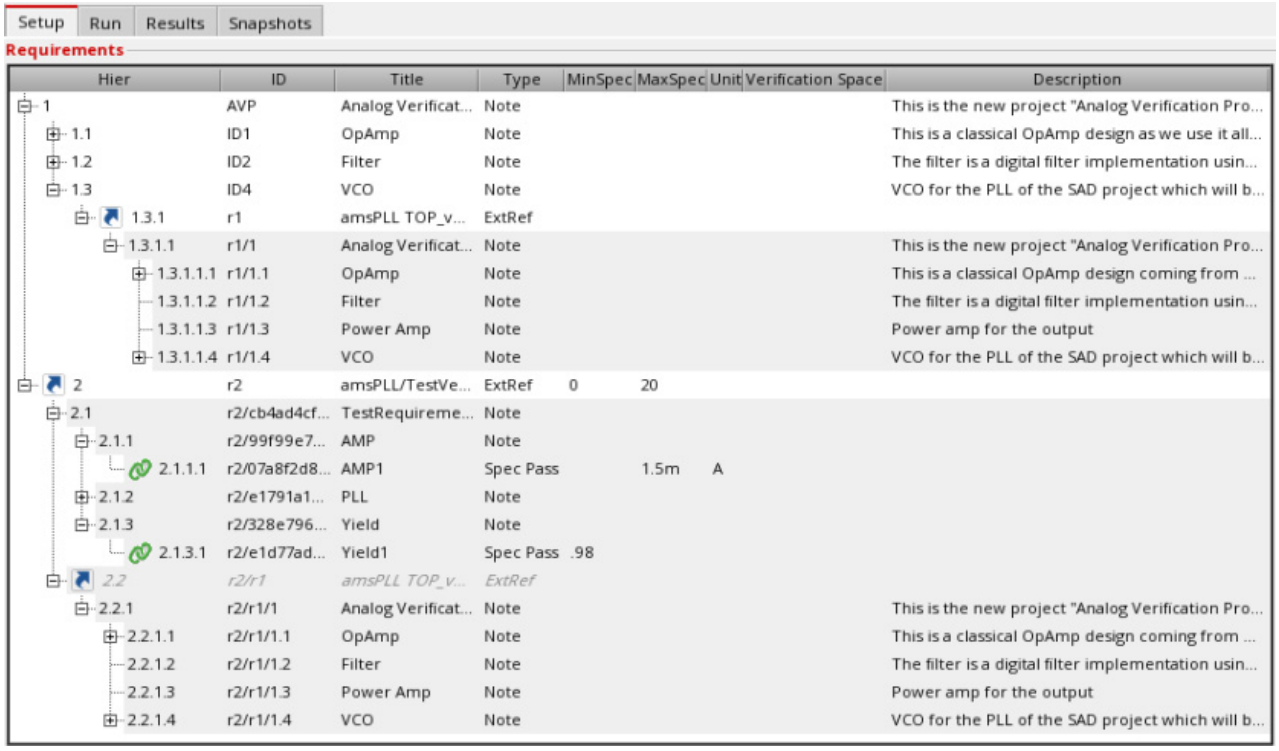
Adding Requirements

© 2023 All Rights Reserved.

Hierarchy for Referenced Cellviews

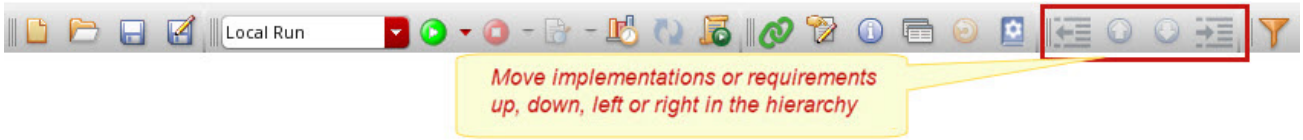
The project manager can get the overall project status by referencing other cellviews that contain requirements and implementations. See Multi-level hierarchy in a design verification project.

The following figure shows a hierarchy of referenced cellviews:



Hier	ID	Title	Type	MinSpec	MaxSpec	Unit	Verification Space	Description
1	AVP	Analog Verificat...	Note					This is the new project "Analog Verification Pro...
1.1	ID1	OpAmp	Note					This is a classical OpAmp design as we use it all...
1.2	ID2	Filter	Note					The filter is a digital filter implementation usin...
1.3	ID4	VCO	Note					VCO for the PLL of the SAD project which will b...
1.3.1	r1	amsPLL TOP_v...	ExtRef					
1.3.1.1	r1/1	Analog Verificat...	Note					This is the new project "Analog Verification Pro...
1.3.1.1.1	r1/1.1	OpAmp	Note					This is a classical OpAmp design coming from ...
1.3.1.1.2	r1/1.2	Filter	Note					The filter is a digital filter implementation usin...
1.3.1.1.3	r1/1.3	Power Amp	Note					Power amp for the output
1.3.1.1.4	r1/1.4	VCO	Note					VCO for the PLL of the SAD project which will b...
2	r2	amsPLL/TestVe...	ExtRef	0	20			
2.1	r2/cb4ad4cf...	TestRequireme...	Note					
2.1.1	r2/99f99e7...	AMP	Note					
2.1.1.1	r2/07a8f2d8...	AMP1	Spec Pass		1.5m	A		
2.1.2	r2/e1791a1...	PLL	Note					
2.1.3	r2/328e796...	Yield	Note					
2.1.3.1	r2/e1d77ad...	Yield1	Spec Pass	.98				
2.2	*r2/r1*	*amsPLL TOP_v...*	*ExtRef*					
2.2.1	r2/r1/1	Analog Verificat...	Note					This is the new project "Analog Verification Pro...
2.2.1.1	r2/r1/1.1	OpAmp	Note					This is a classical OpAmp design coming from ...
2.2.1.2	r2/r1/1.2	Filter	Note					The filter is a digital filter implementation usin...
2.2.1.3	r2/r1/1.3	Power Amp	Note					Power amp for the output
2.2.1.4	r2/r1/1.4	VCO	Note					VCO for the PLL of the SAD project which will b...

The IDs for referenced requirements are constructed hierarchically in the format (`r1`, `r2`…) to be easily readable. You can change the hierarchy levels by using the toolbar buttons shown in the following figure:



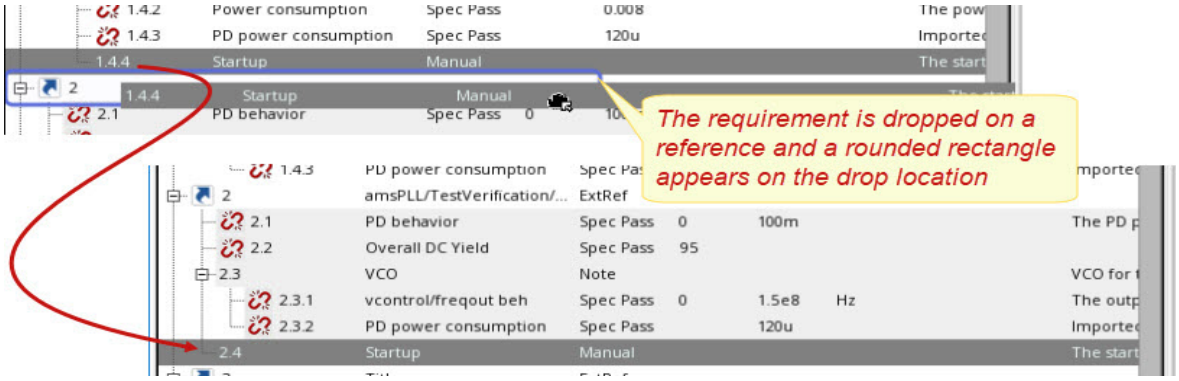
© 2023
All Rights Reserved.

Reassignment of Requirements

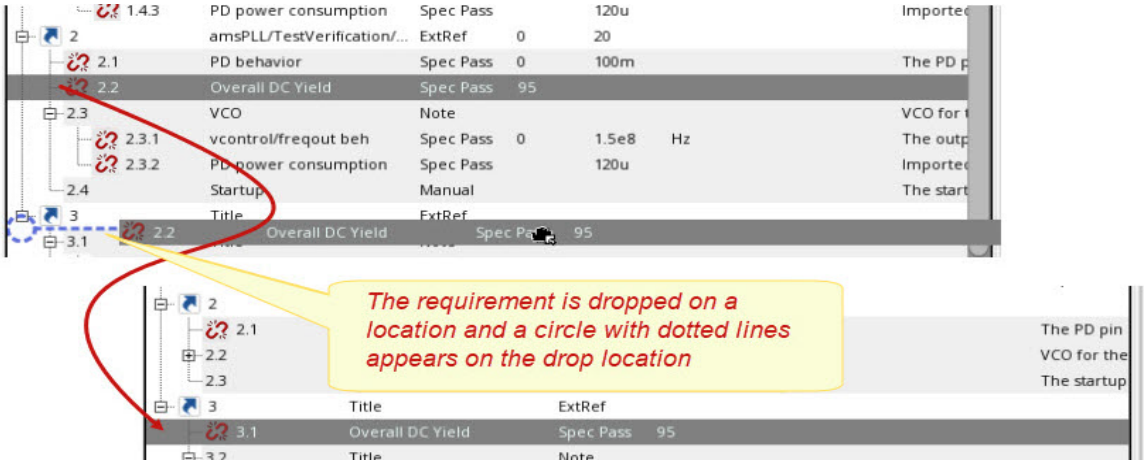
You can drag requirements from the master set and drop them to a reference. Here, the requirements are added as a child of the reference. Similarly, you can also drag a requirement from the referenced cellview and drop it to the master set of requirements. Additionally, you can drag a requirement from a reference and drop it to a different reference. Any of these drop operations results in the requirements being deleted from the original location in the hierarchy.These changes are automatically reflected in the local copy of the reference. This functionality is useful when you, as a project manager need to reassign any requirement or set of requirements to designers in the verification setup.

Additional Information

- When you drop a requirement on another requirement or reference, it is added as a child requirement and moved down to the bottom of existing requirements in the hierarchy.



- When you drop a requirement to a specific position in the hierarchy, it is added as a child requirement at the specified location.



- Requirements that are dragged are deleted from the original position in the hierarchy.

© 2023 All Rights Reserved.

Editing or Updating Broken External References

When you import cellviews as references, Verifier displays them as read-only. You cannot add requirements or implementations to the referenced node in the requirements hierarchy. The mappings between the requirements and implementation are maintained in the *ExtRef* node in the requirement hierarchy.

When the requirement designer makes a change to any requirement in his local cellview, the reference breaks in the master cellview and becomes invalid. This results in the mapping between the requirement and the implementation information to break. The following figure shows how Verifier displays an icon against the requirement in the master cellview for any changes in the referenced cellview:



It is possible that a referenced cellview was renamed or moved. In this case, when you open the Verifier cellview that contains the reference, Verifier prompts you to specify the new location and name, or remove the reference. When you specify the new location or name of the same cellview, the mapping information is retained.

To edit an external reference:

➡ Right-click anywhere on a requirement of *ExtRef* type, and choose *Edit Reference*.

The Change Referenced Cellview Form appears.

It is also possible that the referenced cellview was modified. This is indicated by the icon in the Implementations pane.

To synchronize the master cellview with new modifications:

➡ Right-click anywhere on a requirement of *ExtRef* type, and choose *Update Reference*.

Multiple Hierarchy Levels in Multi-User Verification Projects

In a verification project, multiple designers and the project manager work on their assigned verification plans. It becomes imperative that the requirements and implementations be used in multiple setups. These setups can be the local setups or the master setup which is managed by the project manager.

© 2023 All Rights Reserved.

In a typical bottom-up approach, as a project manager you can reference multiple cellviews from different designers or choose to reference cellviews that contain referenced cellviews. This ensures complete visibility across the project.

Consider an example where Tom, who is the project manager for a verification setup, wants to analyze the verification plan from a designer, Jerry, and imports Jerry's `verifier` cellview. Tom can now view and analyze Jerry's cellview. Tom also references two other cellviews from Jack and Hyde. Tom now has three external references in the master setup. Kate is another requirement designer on this project. Jerry references Kate's cellview. Tom updates the references in the master setup and can see the verification plan from Kate in Jerry's verification details. In this case, a multi-level owner hierarchy is set up. Kate, Jack and Hyde cannot view any other cellview other than their local cellview, but Jerry can view Kate's cellview, and Tom can view cellviews from Jack, Hyde, Jerry and Kate.

Alternatively, you can export requirements from your verification setup and a requirement owner can import these requirements to a local Verifier cellview. The requirement owner can then export a set of requirements, and a new owner can import these requirements in another local cellview. In this case, an hierarchy with multiple levels of owners is introduced. You can choose to add more levels to the owner hierarchy. The project manager can reference cellviews from all these owners to view the master verification plan. It may be possible that multiple project managers reference a cellview from a requirement owner.

Taking the same example above, Tom now assigns a set of requirements to Jerry, and exports these requirements to a CSV or Microsoft Excel file for Jerry to reuse. Jerry imports these requirements and adds implementations and mappings in a local cellview. Tom can now reference Jerry's cellview. Jerry now repeats the same steps as Tom, to export a set of requirements in a file. Kate, another requirement owner, can now import these requirements in another Verifier setup. Jerry references Kate's cellview. In this case, a multi-level owner hierarchy is set up. Kate cannot view any other cellview other than their owned cellview, Jerry can view Kate's cellview, and Tom can view cellviews from Jerry and Kate.

Synchronization Between the Master and Referenced Cellviews

When you import owner cellviews using external references, Verifier makes the following changes:

- The master cellview references external cellviews to capture the implementations and mappings that requirement owners add in their owner cellviews. The project manager cannot make any additions, deletions or modifications in the referenced cellviews.
- The verification setup has unidirectional visibility. This means that the project manager can view the referenced cellviews in the master cellview, but the requirements and

© 2023 All Rights Reserved.

implementations in the master cellview are visible only to the project manager. The external cellview owners can only view the requirements and implementations in their local cellview.

- The requirement designers can make any changes to the requirement hierarchy as required in their local setup. The referenced requirements, imported from these local cellviews, are displayed in the master cellview. Any additions to the requirement hierarchy in the local cellview remain invisible to the project manager.
- Verifier watches the referenced cellview for any changes. However, the project manager may choose the shortcut menu action *Update Reference* to synchronize the master cellview with the referenced cellview.

The synchronization between the master and owner cellviews ensure that the following information is synchronized in the master cellview of the verification plan:

- Requirements hierarchy
- Implementations and their mapping with the requirements
- Manual sign-off details

Only the requirement designers can sign off their requirements that failed verification or are of the type *Manual* in their local cellviews. The designers can also edit and delete their sign off details. For details, see Requirements Signoff in Virtuoso ADE Verifier.

Important

You cannot sign-off a referenced requirement on the *Results* tab.

The synchronization between the master and owner cellviews is based on the UNIX file system visibility. The changes in a cellview become available to the cellview that references it only when those changes are saved. If you are using a data management system with Virtuoso, the changes become available only when the updated cellview is checked in and the cellview that references the updated cellview is refreshed.

In a Verifier setup with multiple designers and users, by default, the results are stored in the `results` directories of the current cellview. To know more, see Location of Run Summary Data.

© 2023 All Rights Reserved.

Updating Referenced Cellviews

To update a referenced cellview:

➡ Right-click anywhere in the Requirements pane, and choose *Update Reference*.

To update all referenced cellviews do one of the following:

➡ Click the *Update Implementations and References*  icon anywhere in the Requirements pane, and choose *Update Reference*.

➡ Choose *Tools – Update Implementations and References*.

Simulation Runs on Referenced Cellviews

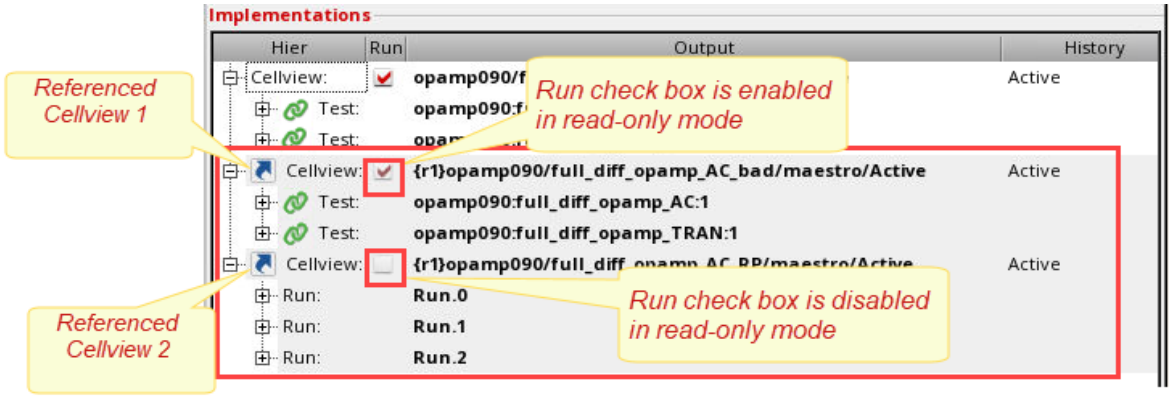
A typical multi-user verification plan in Verifier consists of the top-level cellview and the referenced cellviews. You can run simulations on these cellviews using *Run*() from the toolbar. You can open single or multiple referenced cellview in parallel with the top-level cellview and view the status in the progress bar in these cellviews.

Additional Information

- To enable simulation of a referenced cellview from the top-level cellview, ensure that the *Run* check box is selected in the referenced cellview. To select or deselect this check box, you need to open the referenced cellview. You cannot update this setting from the top-level cellview.
- If you have an implementation and a reference to the same implementation in a cellview, both update when one of them finishes simulation. The referenced implementation essentially exists in a separate cellview and its PALS results are saved in a different location. However, the implementation and the referenced implementation both indicate the same `maestro` cellview, and you need to simulate any one of them and update the PALS results of the other, irrespective of the order. For example, you have an implementation `amsPLL/PLL_VCO_320MHZ_tb/maestro_nominal` and an external reference `R1`, to `amsPLL/PLL_VCO_320MHZ_tb/maestro_nominal`. Both point to the same maestro cellview, but their results are saved in separate locations. You can run the simulation on `R1` or the implementation and update the results for the other.
- The Maximum implementations to run in parallel value that you specify in the Preferences form of the top-level cellview determines the number of simulations that can be run in parallel. The preference value specified for the referenced cellview is ignored. For example, the preference value is set to '5' in the referenced cellview, and '1' in the top-level cellview. In this case, Verifier uses the value '1' from the top-level cellview and runs the scheduled simulations only one at a time.

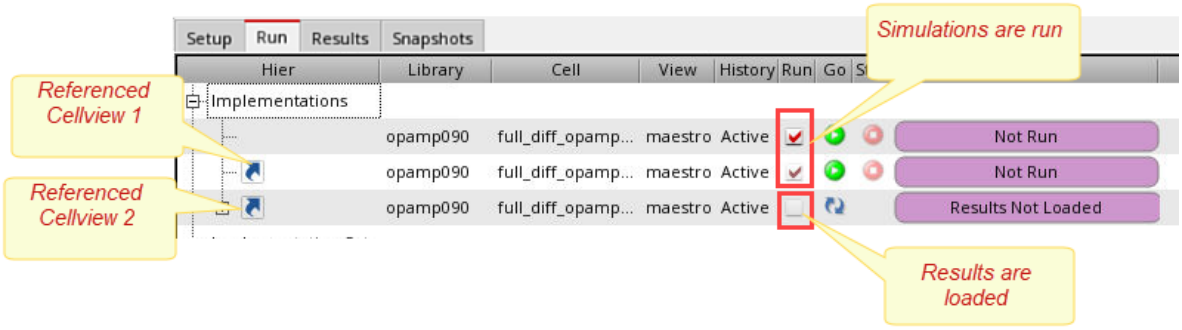
© 2023 All Rights Reserved.

The following illustration shows the *Implementations* pane with two referenced cellviews in the top-level cellview:

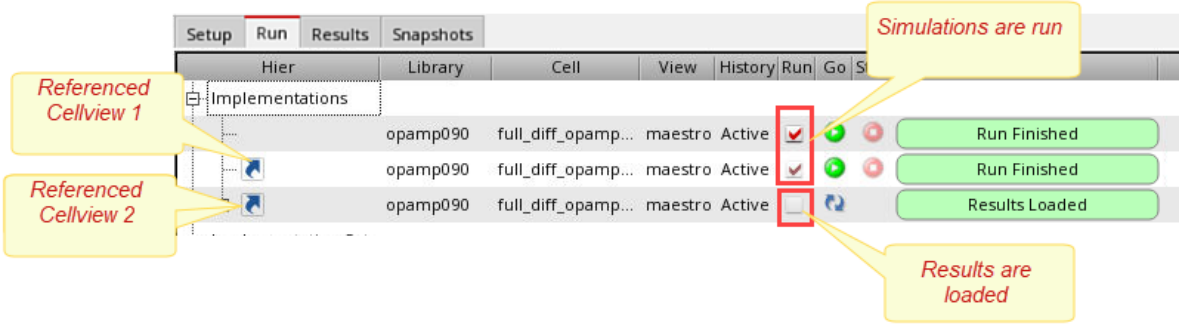


Here, the *Run* check box is enabled for just one of the referenced cellviews. However, the setting cannot be modified for both the referenced cellviews.

The *Run* tab also displays the same settings as the *Implementations* pane. Based on the setting of the *Run* check box, you can run simulations or load the results for a referenced cellview.



In the given scenario, clicking *Run* on the toolbar runs the first two simulations. The results are loaded for the second reference that does not have the *Run* check box enabled.



© 2023
All Rights Reserved.

10

Verification in ADE Verifier with Reliability

Reliability analysis is performed on `maestro` cellviews to analyze the effects of time on circuit performance drift and to predict the reliability of designs in terms of performance. In other words, reliability analysis helps to evaluate the degradation of designs caused by the passage of time or self-heating. In ADE Assembler, you can run reliability simulations to generate outputs for fresh tests (when time is zero), stress tests (to generate degradation data), and age tests (at specific intervals, such as 1 year, 3 years, or 10 years).

ADE Verifier (Verifier) provides an overarching verification plan that links all analog tests across multiple designs. With reliability outputs in implementations, you can verify if the design degradation is according to the verification plan.

To know about setting up a reliability analysis in ADE Assembler, see Performing Reliability Analysis in the *Virtuoso ADE Assembler User Guide*.


Video

For a video overview of the reliability flow, see *Reliability in Virtuoso ADE Verifier* on Cadence Online Support. Also read a blog related to this at *Virtuosity: Verification in Virtuoso ADE Verifier - The Reliability Way!*.


Important

Currently, ADE Verifier does not support MAA/MAS outputs and reliability scenarios.

Related Topics

Reliability Hierarchy in ADE Verifier

Reliability Outputs in ADE Verifier

Reliability Flow in ADE Verifier

© 2023 All Rights Reserved.

Reliability Outputs in Different Run Modes

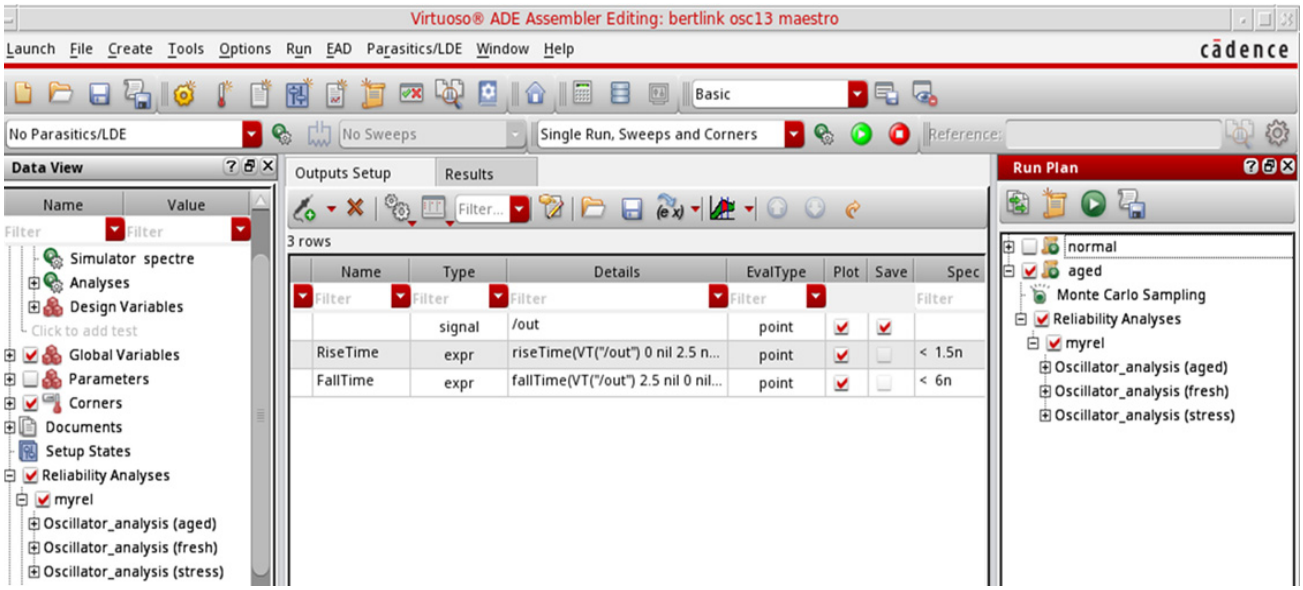
Validating Simulation Results for Reliability Outputs

Viewing Simulation Logs for Reliability Outputs

Reliability Hierarchy in ADE Verifier

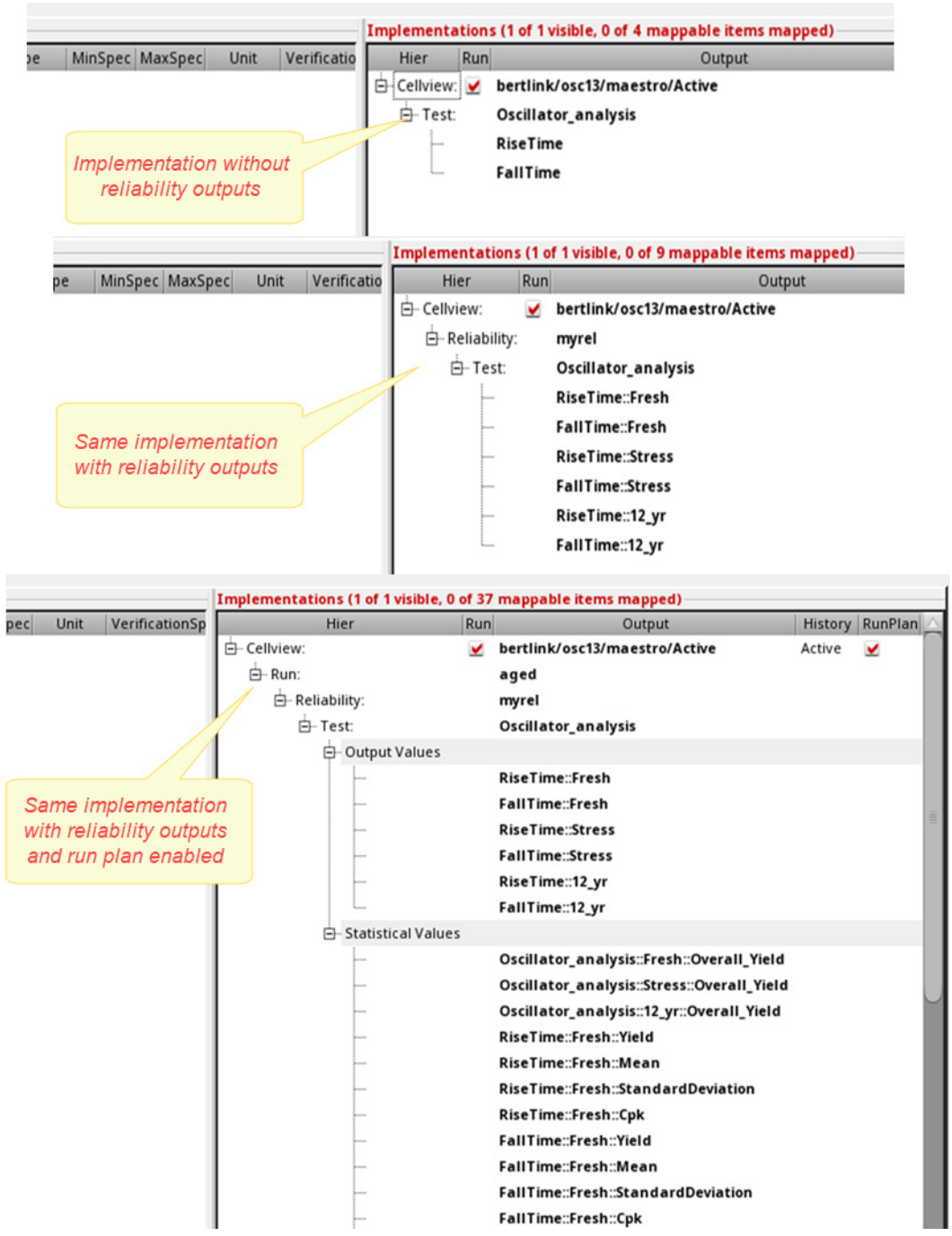
When you add a `maestro` cellview as an implementation in Verifier, the hierarchy is based on the tests in the cellview. However, when you add a `maestro` cellview with a reliability setup enabled, the hierarchy changes in Verifier and is additionally based on the reliability setups.

Consider the following figure of a `maestro` cellview where a reliability analysis and a run plan are set up and enabled.



© 2023
All Rights Reserved.

The following figure shows the differences in the Verifier hierachy in such a scenario.



Related Topics

Verification in ADE Verifier with Reliability

© 2023
All Rights Reserved.

Reliability Outputs in ADE Verifier

Reliability Flow in ADE Verifier

Reliability Outputs in Different Run Modes

Validating Simulation Results for Reliability Outputs

Viewing Simulation Logs for Reliability Outputs

Reliability Flow in ADE Verifier

A typical reliability flow in ADE Verifier includes the following steps:

1. Open a `maestro` cellview in ADE Assembler.
2. Specify a reliability analysis setup.

   See Specifying the Reliability Analysis Setup.
3. Save the `maestro` cellview.
4. In ADE Verifier, choose *Edit – Preferences – General – Reliability Outputs* and select the required settings.
5. Add the `maestro` cellview as an implementation.

   See Adding Implementations.
6. Create requirements from the implementation.

   See Creating Requirements from Implementations.
7. Run a simulation in Verifier.

   See Simulation of Implementations.
8. View the results of the simulation run.

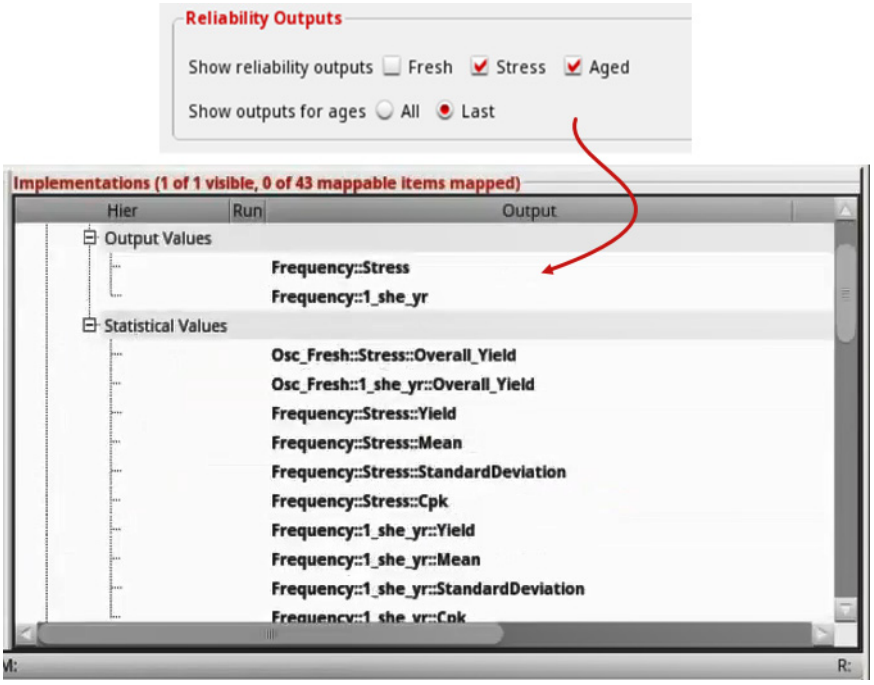
   See Verification Status of Requirements and Validating Simulation Results for Reliability Outputs.
9. View the simulation logs.

   See Viewing Simulation Logs for Reliability Outputs.

The reliability outputs are displayed in the implementation hierarchy based on the selected settings. For example, to view reliability outputs related to stress and age tests, select the

© 2023 All Rights Reserved.

Stress and *Aged* check box next to *Show reliability outputs* and deselect the *Fresh* check box. The implementation will now only display the outputs from the stress test.



Related Topics

Verification in ADE Verifier with Reliability

Reliability Hierarchy in ADE Verifier

Reliability Outputs in ADE Verifier

Reliability Outputs in Different Run Modes

Validating Simulation Results for Reliability Outputs

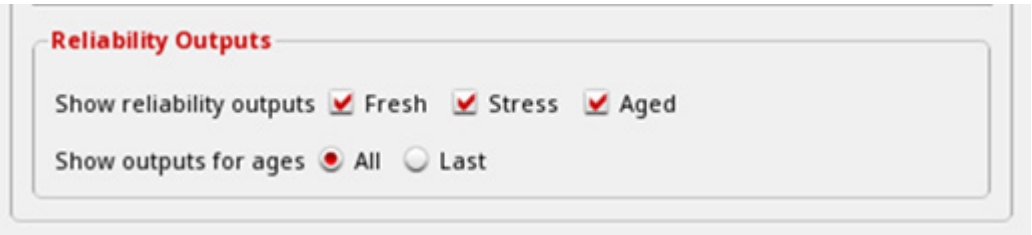
Viewing Simulation Logs for Reliability Outputs

Reliability Outputs in ADE Verifier

The reliability outputs in a Verifier implementation hierarchy might differ depending on the analysis and aging settings specified in the reliability setup in ADE Assembler. To enable

© 2023 All Rights Reserved.

reliability outputs in Verifier, you need to enable the related settings in the Preferences form as shown in the following illustration.



Using reliability analysis can expand the implementation hierarchy significantly. To be efficient in your verification, it is important that you focus only on the data that is really relevant to your final verification and disable all other options.

Alternatively, you can also use the following environment variable in the Virtuoso CIW to specify the preferences related to reliability outputs:

- displayFresh
- displayStress
- displayAge
- displayRelOutputAges

Example:The following usage of the environment variables will show the reliability outputs for fresh tests and last age only.

```
envSetVal("verifier.preferences" "displayFresh" 'boolean t)
envSetVal("verifier.preferences" "displayAge" 'boolean t)
envSetVal("verifier.preferences" "displayRelOutputAges" 'string "Last")
```

By default, all reliability outputs are selected and displayed for all ages. However, you can make changes to the selection based on the outputs that you want to display in Verifier implementations.

Related Topics

Verification in ADE Verifier with Reliability

Reliability Hierarchy in ADE Verifier

Reliability Flow in ADE Verifier

Reliability Outputs in Different Run Modes

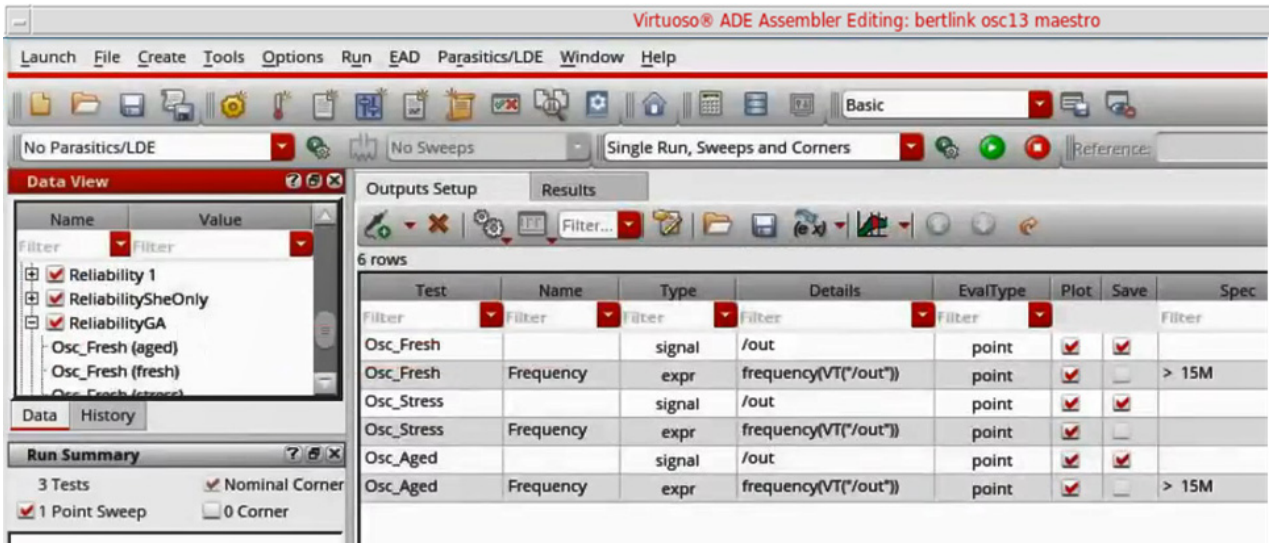
Validating Simulation Results for Reliability Outputs

© 2023 All Rights Reserved.

Viewing Simulation Logs for Reliability Outputs

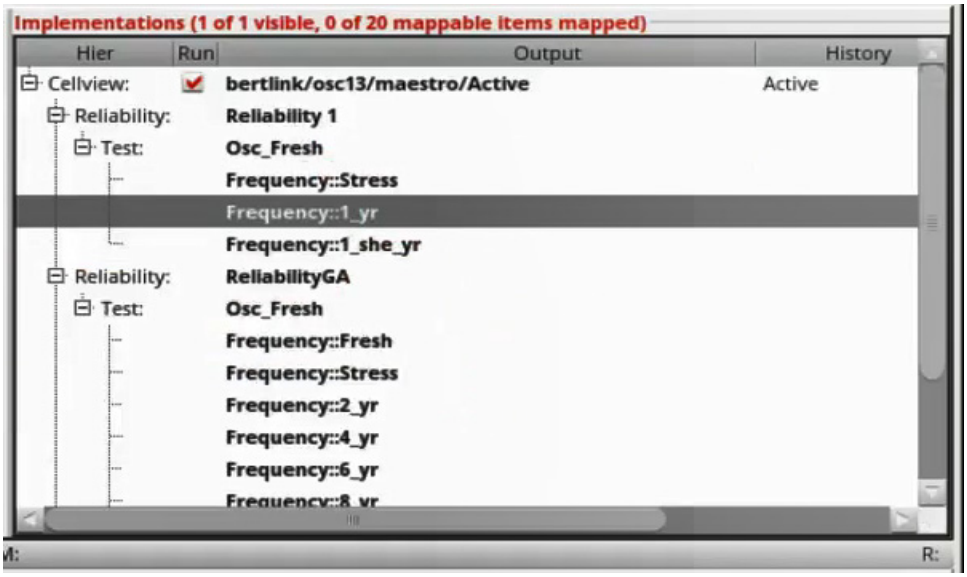
Reliability Outputs in Different Run Modes

Consider that you have the following `maestro` cellview in ADE Assembler.



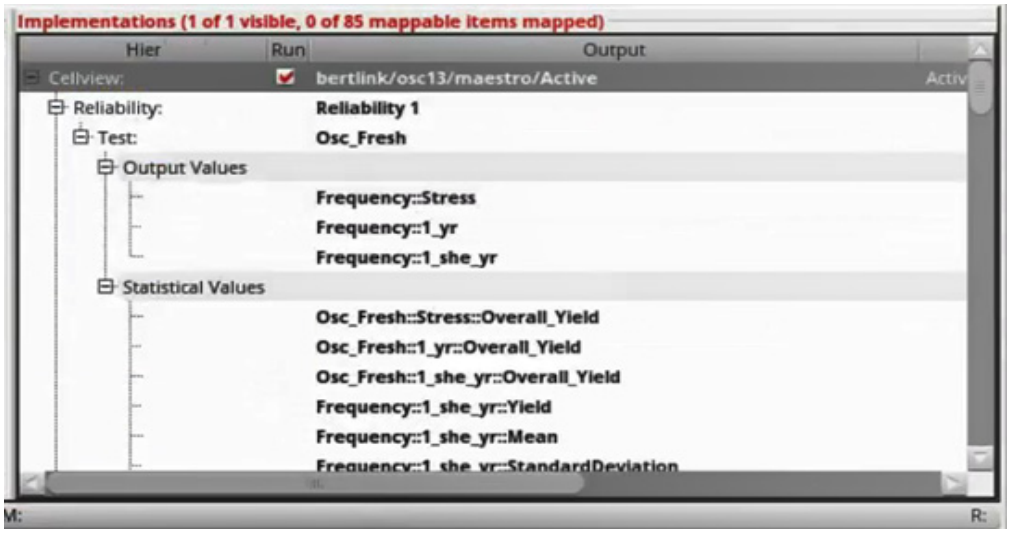
In Verifier, when you add this `maestro` cellview as an implementation, the hierarchy changes as follows depending on the selected settings.

- When run mode is set to *Single Sweep, Sweeps and Corners* in ADE Assembler, the outputs are grouped by the tests under the reliability setup.



© 2023
All Rights Reserved.

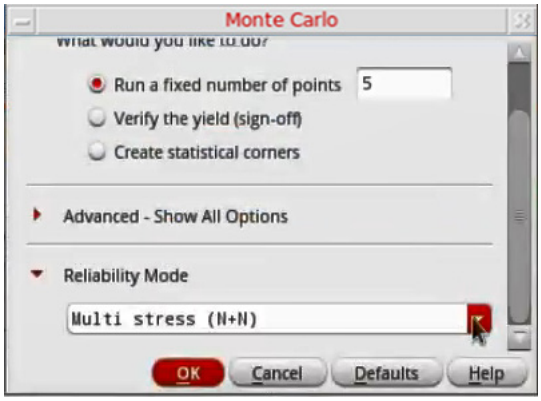
- When run mode is set to *Monte Carlo Sampling* in ADE Assembler, the implementation displays additional statistical parameters than the *Single Run, Sweeps and Corners* run mode. Statistical corners can be added accordingly. The output and statistical values are grouped separately in the *Output Values* and *Statistical Values* sections under a test, and all tests are grouped under the corresponding reliability setup.



You can run simulations in the *Monte Carlo Sampling* run mode using two reliability modes:

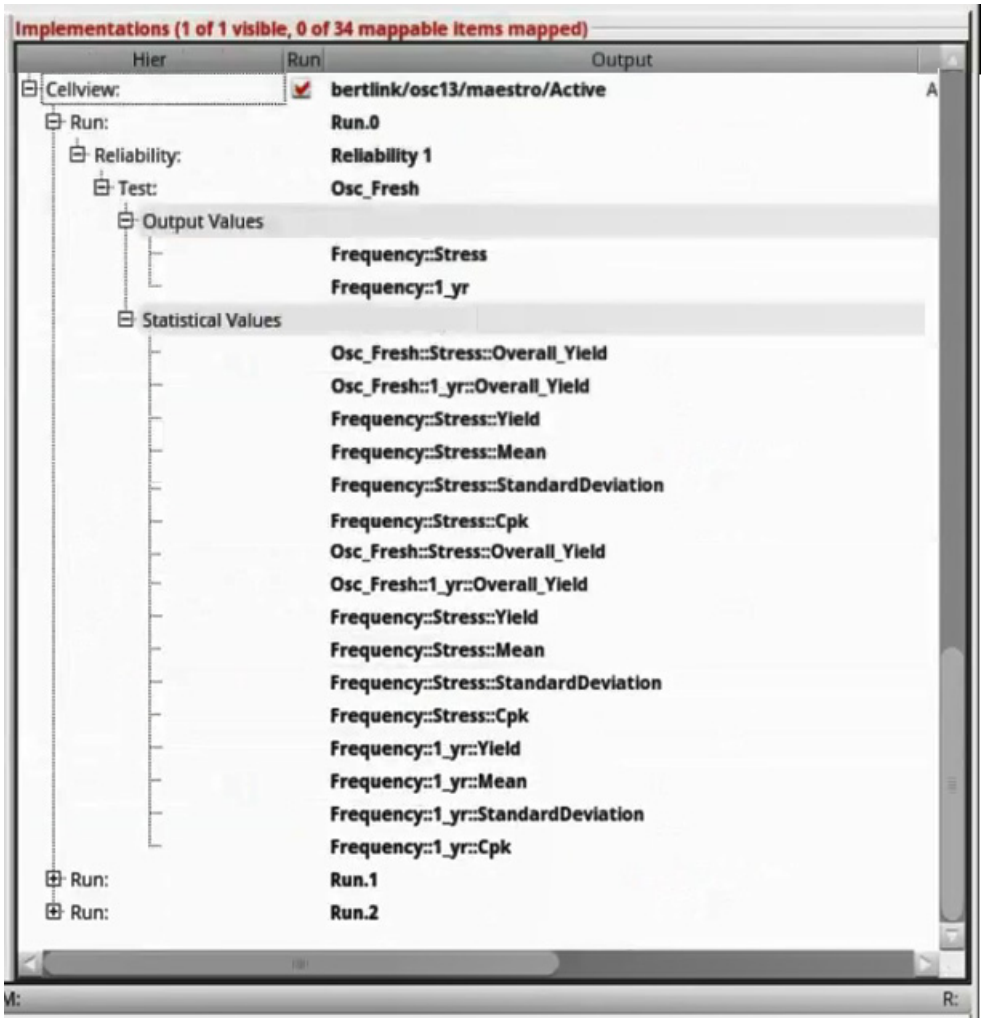
- **N+N**

  In this flow, *Monte Carlo Sampling* (process variations) is applied in both the stress and age simulations to provide more higher accurate results.



© 2023
All Rights Reserved.

The reliability outputs in the Verifier implementation show the different outputs and values for multiple stress simulation points.

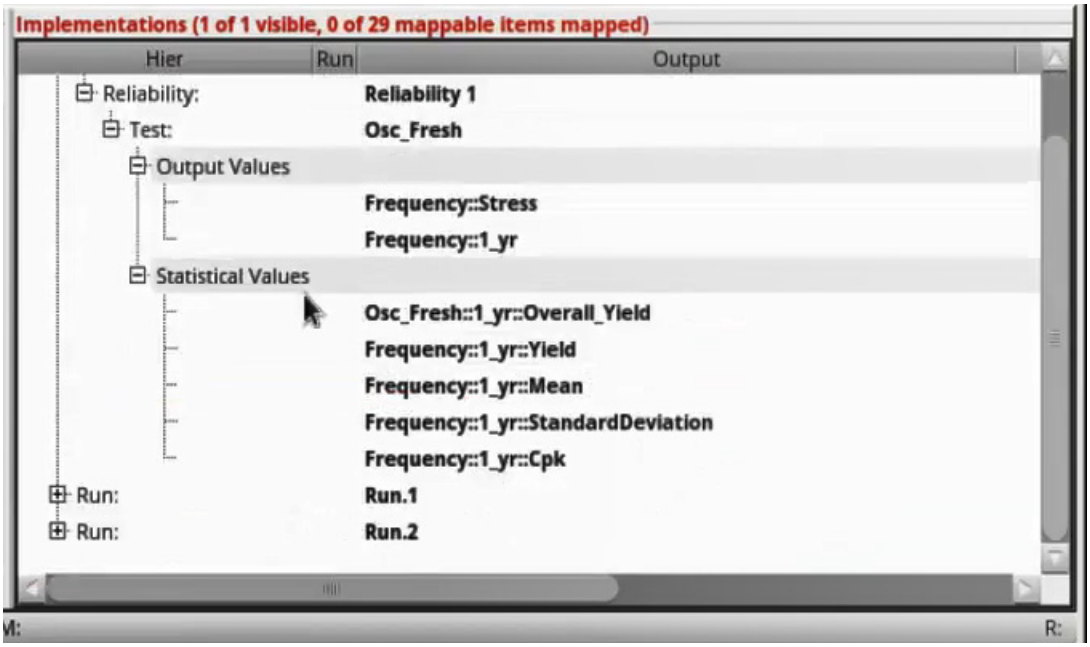


- **1+N**

  In this flow, a single stress simulation is performed in the first iteration to calculate the aging degradation. It is reused with process variations during Monte Carlo analysis combined with aging.

© 2023
All Rights Reserved.

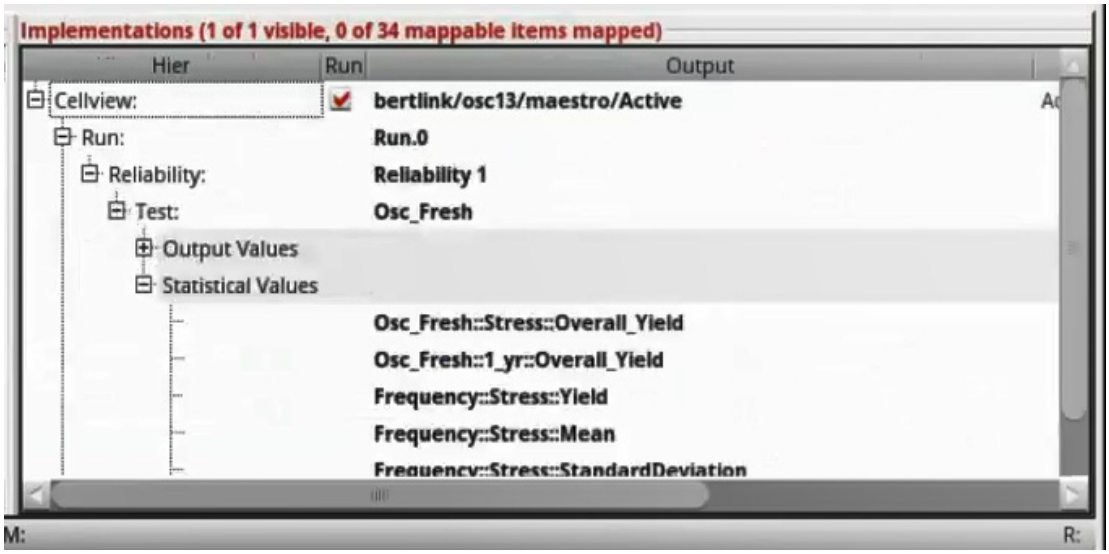
To view Monte Carlo Sampling outputs and statistical values for single-stress simulations, change the *Reliability Mode* to *Single stress* (1+N) in the Monte Carlo form.
The reliability outputs in the Verifier implementation show the different outputs and values for single-stress simulation points. Here, the stress values are removed from the statistical values.



- When *Run Plan Mode* is enabled in Verifier, the run plan is shown on the top in the implementation hierarchy. Similar to the hierarchy displayed for *Monte Carlo Sampling* run mode, the outputs and statistical values in the *Outputs* and *Statistical Values*

© 2023
All Rights Reserved.

sections are separately grouped under a test. All tests are grouped under the corresponding reliability setup. All reliability setups are again grouped under a run plan.



Related Topics

Verification in ADE Verifier with Reliability

Reliability Hierarchy in ADE Verifier

Reliability Outputs in ADE Verifier

Reliability Flow in ADE Verifier

Validating Simulation Results for Reliability Outputs

Viewing Simulation Logs for Reliability Outputs

Validating Simulation Results for Reliability Outputs

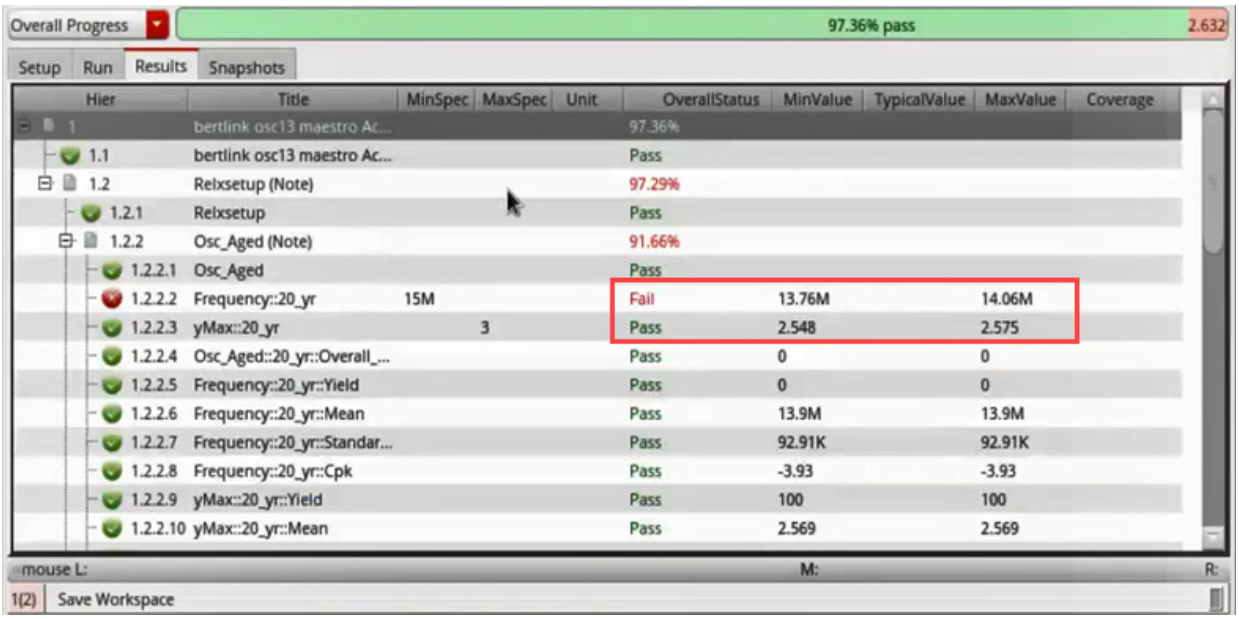
To validate simulation results for reliability outputs:

1. Open a `maestro` cellview in ADE Assembler.
2. Run a reliability simulation.
3. In Verifier, add the `maestro` cellview as an implementation.

© 2023 All Rights Reserved.

4. Right-click the implementation and choose *Update Implementations* from the shortcut menu.
5. Click the *Results* tab to view the generated results.

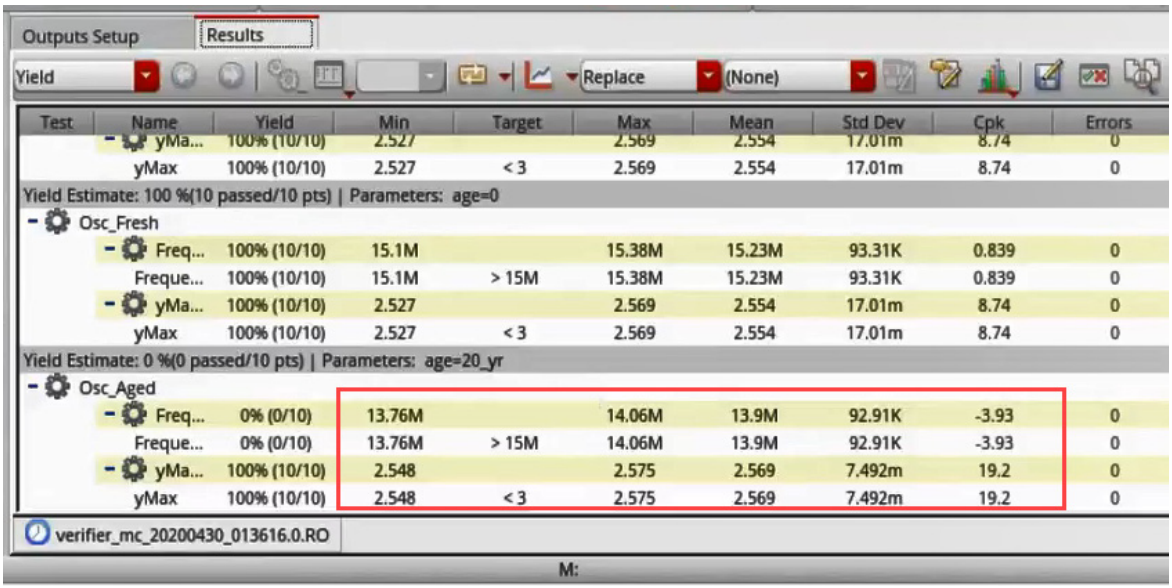
   The results are filtered for different ages. You can evaluate the results based on the age values and variable values.



6. Validate the results as follows:
   a. On the *Results* tab, select the required specification.
   b. Right-click the selected specification and choose *View Implementation Results*.

© 2023
All Rights Reserved.

The results from the `maestro` cellview are displayed.



Related Topics

Verification in ADE Verifier with Reliability

Reliability Hierarchy in ADE Verifier

Reliability Outputs in ADE Verifier

Reliability Flow in ADE Verifier

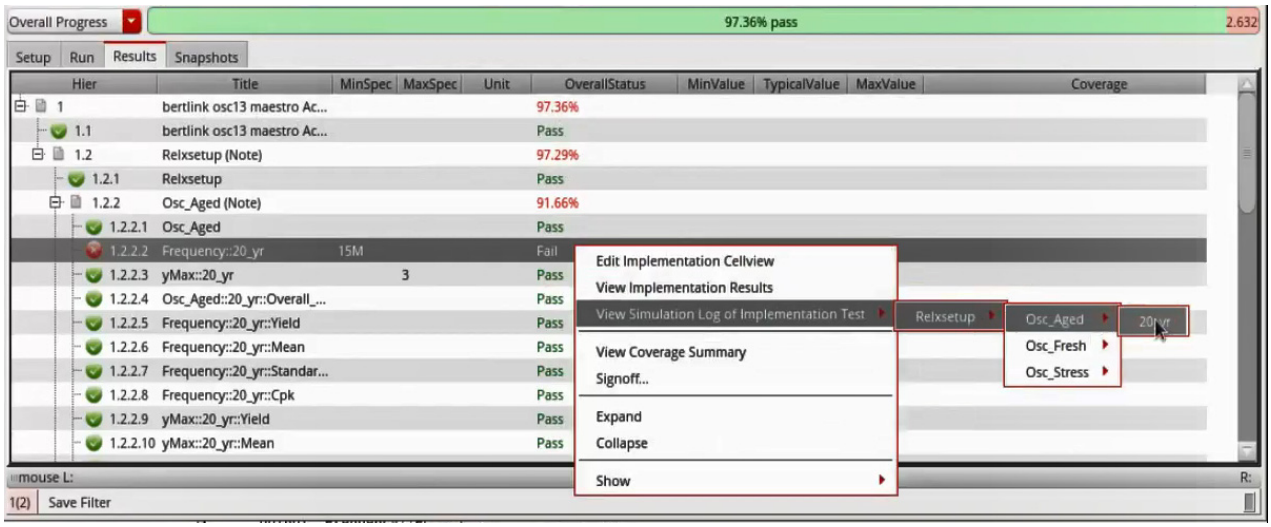
Reliability Outputs in Different Run Modes

Viewing Simulation Logs for Reliability Outputs

© 2023 All Rights Reserved.

Viewing Simulation Logs for Reliability Outputs

The simulation logs of reliability simulations are grouped on the basis of the reliability setups that include implementation tests. The tests in a reliability setup also include groups of aging result details.

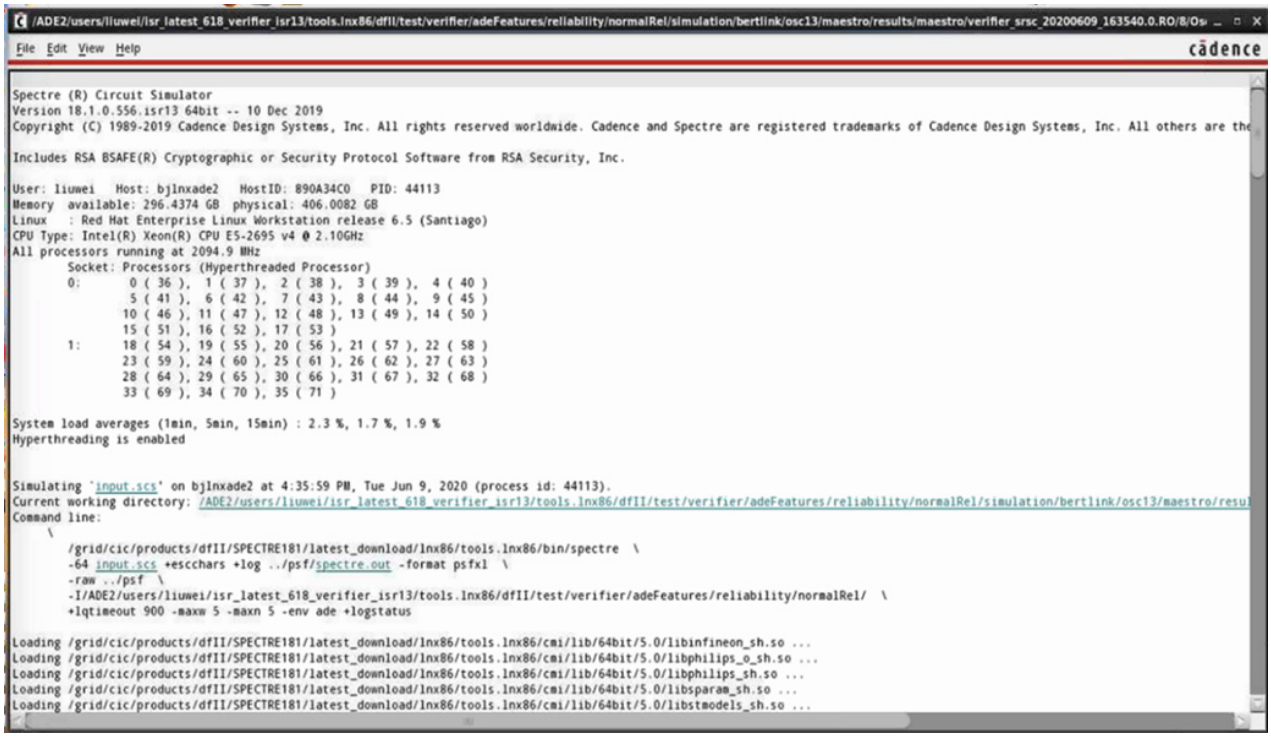


To view the output logs for reliability simulation results:

1. In the *Results* window, select the specification for which you want to view the output log.
2. Right-click the selected specification and choose *View Simulation Log of Implementation Test – <reliability setup> – <test> – <age point>*.
3. Select the required reliability setup listed in the submenu.
4. From the submenu that opens, select the required test.
5. From the submenu that opens, select the required age point of the specification.

© 2023
All Rights Reserved.

The output log displays the details of the first point corresponding to the selected age in a new window.



If the `maestro` cellview does not contain a sweep value and the reliability setup contains only one age point, the output log displays the details of that age point. If the `maestro` cellview contains a sweep value, the output log displays the details of the first point corresponding to the selected age.

Related Topics

Verification in ADE Verifier with Reliability

Reliability Hierarchy in ADE Verifier

Reliability Outputs in ADE Verifier

Reliability Flow in ADE Verifier

Reliability Outputs in Different Run Modes

Validating Simulation Results for Reliability Outputs

© 2023 All Rights Reserved.

© 2023 All Rights Reserved.

11

Mixed-Signal Design Verification

This topic provides information about the connection between Virtuoso© ADE Verifier™ (Verifier) and Cadence© vManager™ (vManager) for requirements-driven mixed-signal design verification flows.

The functional verification of mixed-signal designs is one of the top challenges in any verification project. All mixed-signal designs need analog, digital, and mixed-signal verification. For the successful functional verification of a project, some of the required aspects are metric-driven verification, top-down verification planning, and reporting or traceability of bottom-up results.

Verifier lets you set up an analog design verification project using a requirements-driven methodology. From a high level perspective vManager uses the same methodology to let you analyze and plan the verification of your digital requirements. Using Verifier and vManager, you can plan the verification of your project in a mixed-signal environment and analyze all the results together in a single cockpit.

 Video

For a video overview of the Verifier-vManager connection, see *Mixed-Signal Design Verification with ADE Verifier and vManager* on Cadence Learning and Support.

Related Topics

ADE Verifier-vManager Flow

vManager Connection Settings in ADE Verifier

List of Mappings between Verifier Requirement Fields and vManager Attributes

ADE Verifier Results in vManager

Connecting Multiple Verifier Views to a vPlan

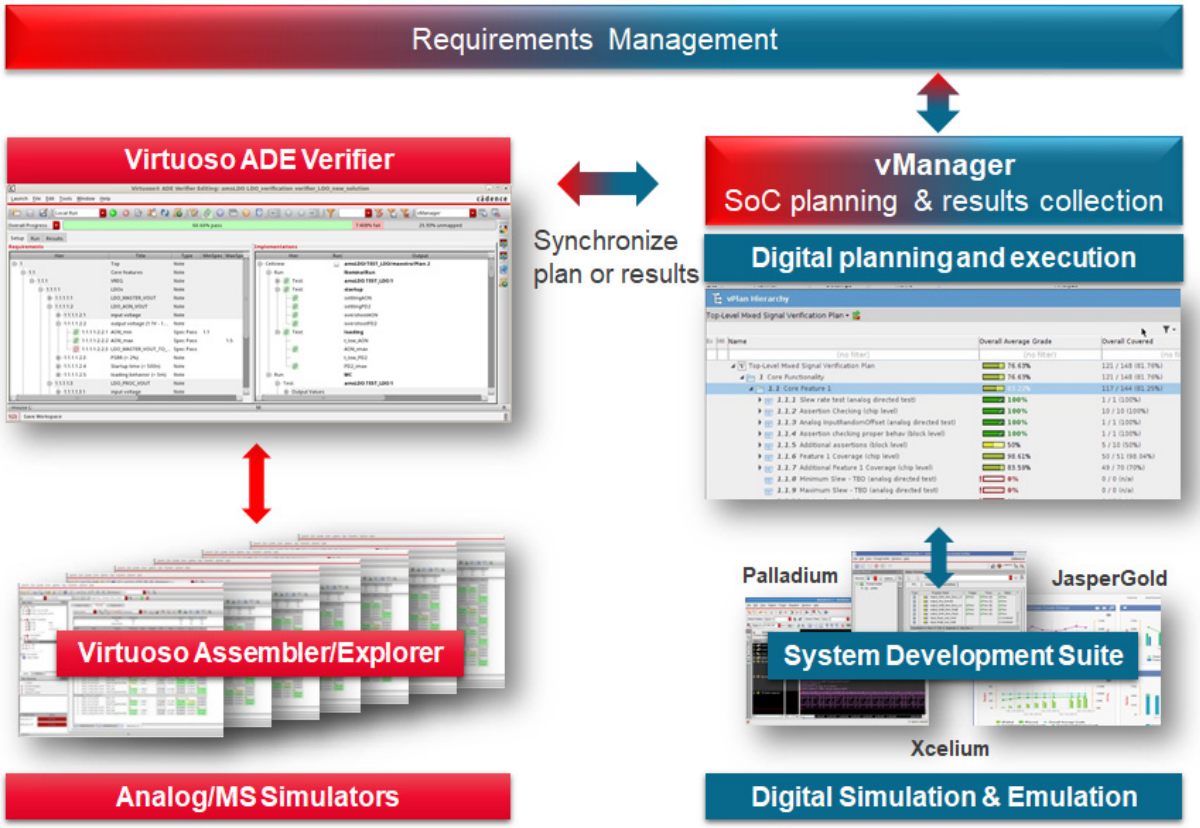
Review of the Verification Results in vManager

© 2023 All Rights Reserved.

Referenced Cellviews in an ADE Verifier-vManager Session

ADE Verifier-vManager Flow

vManager and Verifier provide a point solution using the requirements-driven methodology for the digital and the analog flow respectively.



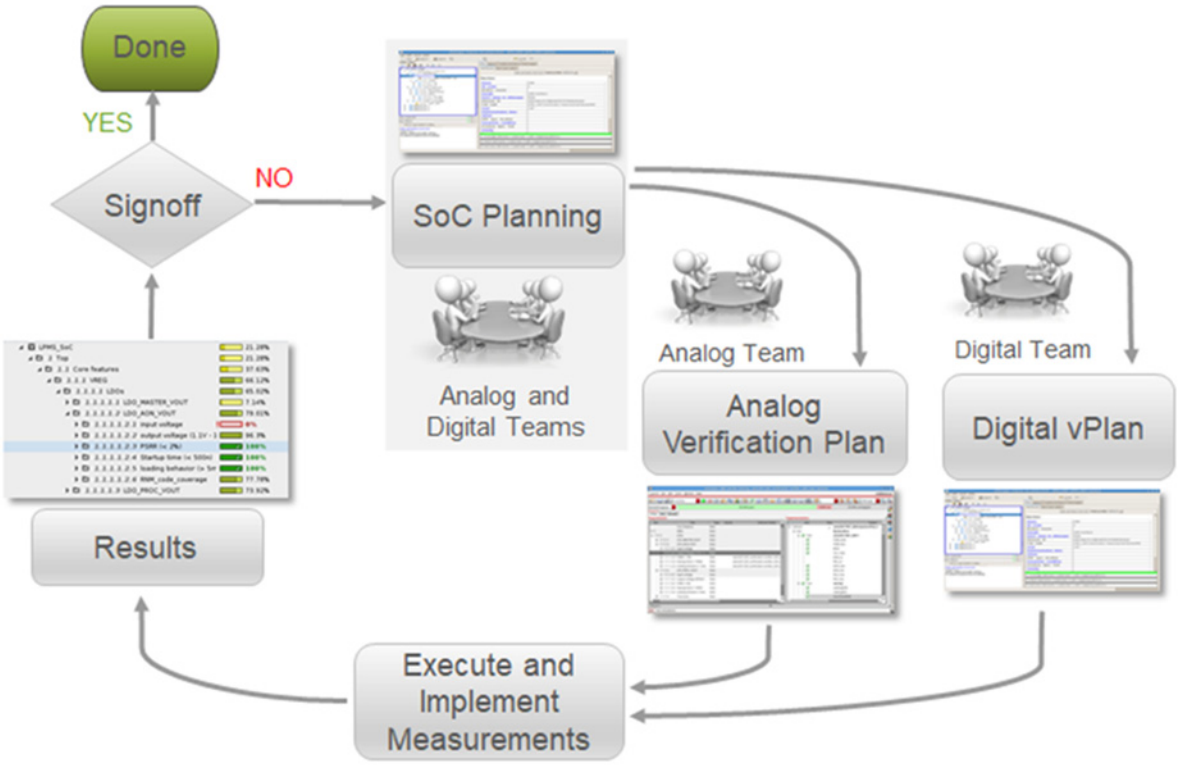
In an ideal scenario, mixed signal design verification starts with a planning session with all the engineers in the analog and digital verification teams. The planning session helps them develop a common understanding of the project. The engineers capture the high-level verification requirements and evaluate factors such as the interfaces, important performance measures, and integration of the different design blocks. The specification document is the reference for most of the high level verification items.

After the initial plan is completed, the analog and digital teams begin refining their respective parts of the plan by capturing requirements and evaluating the dependencies for their domains. These refinements include implementing checks, testbenches, measurements, running simulations, or modifying measurements in the testbenches of their respective plans.

© 2023
All Rights Reserved.

When satisfactory results have been achieved, the analog and digital plans and their results are combined. This provides a common representation of the verification status and highlights any conflict in data, connection or communication. The combined plan might change further because of other changing aspects.

The following figure displays the typical flow in a mixed signal verification project.



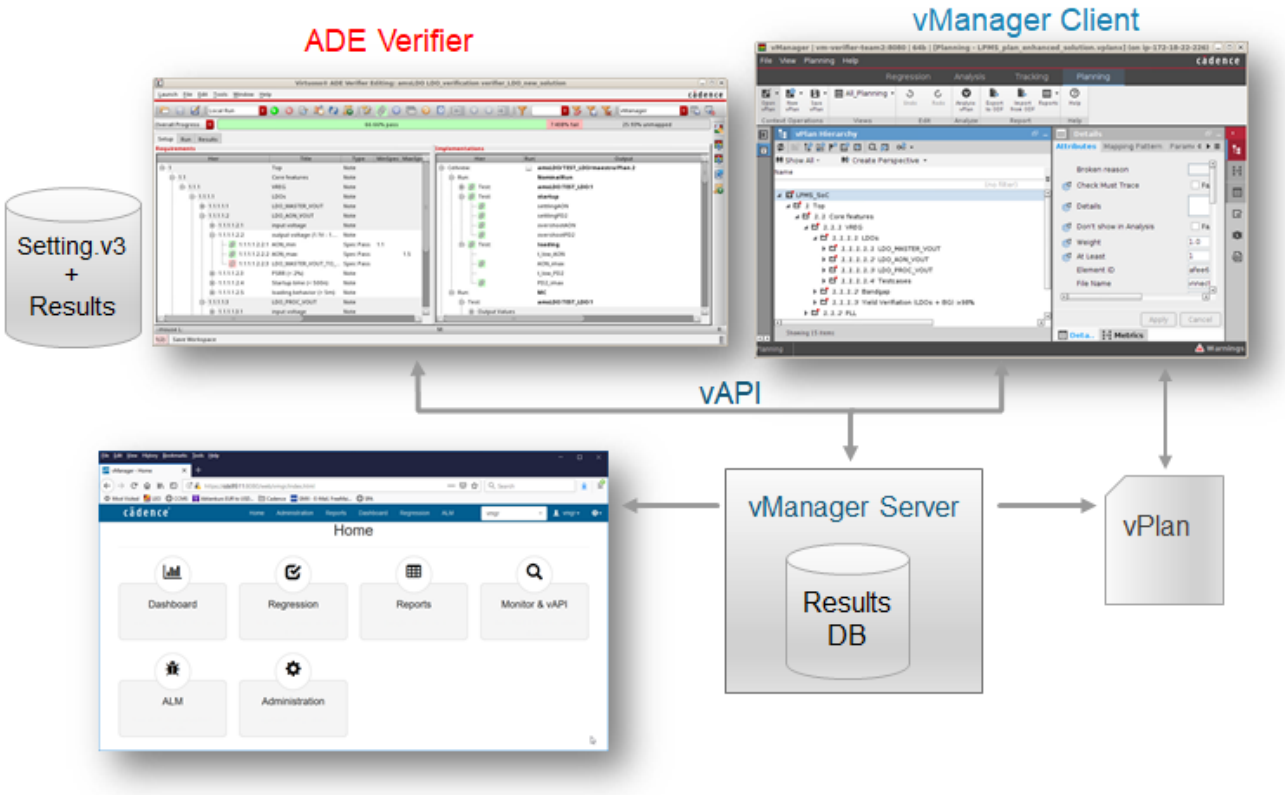
ADE Verifier Architecture

Verifier uses the vManager client-server architecture. In this architecture, Verifier is a client similar to the vManager client. The vManager server is connected to a database and can access the verification plans of clients. This allows the vManager or Verifier clients to connect to each other and fetch and send data.

The server architecture ensures synchronization between all clients. Each of the clients retains their own local data while sharing essential information with the server. You can also

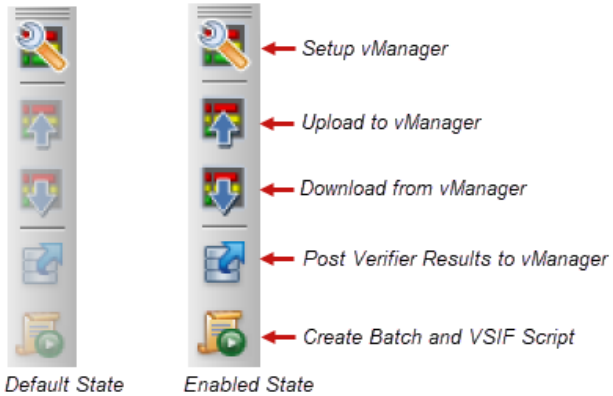
© 2023 All Rights Reserved.

connect multiple Verifier cellviews to the same vPlan to enable the verification of a large mixed-signal project. See Connecting Multiple Verifier Views to a vPlan.



vManager Connection Settings in ADE Verifier

The vManager toolbar is available to the right in the Verifier window. Use this toolbar to manage the vManager connection.



© 2023
All Rights Reserved.

This toolbar contains task-oriented commands required to synchronize the verification plans and results in Verifier and vManager. The commands are enabled after you specify the settings for the vManager connection. You can click the required command on the toolbar. Alternatively, you can choose *Tools – vManager – <required command>*.

The following tables describe the main commands on the vManager toolbar.

Command	Description
Setup vManager	Saves the vManager configuration settings. The other commands in the toolbar are enabled when valid configuration settings are provided.
Upload to vManager	Uploads the requirements of 'ANALOG' domain and their mapping information to the specified vPlan file using the vManager server.
Download from vManager	Downloads the data from the vPlan file from the vManager server to the Verifier session. A tooltip shows the last time a download was completed.
Post Verifier Results to vManager	Posts the verification results to a new vManager session.
Create Batch and VSIF Script	Creates a batch file and a VSIF file.

Accessing the vManager Toolbar

To access the vManager toolbar, do one of the following:

- Choose *vManager* in the workspace list on the Verifier toolbar.
- Choose *Window – Workspaces – vManager*.
- Choose *Window – Toolbars – vManager*.
- Right-click anywhere on the Verifier toolbar and choose *Toolbars – vManager*.

© 2023 All Rights Reserved.

Prerequisites for Setting Up a vManager Connection in ADE Verifier

- Virtuoso IC6.1.8 and ICADVM18.1 ISR3 or higher.
- vManager 19.02 AGILE or higher with the *Analog* work mode enabled in the server setup. For more information, see Project Configurations -- Settings -- Generic Configurations in *Cadence© vManager™ Web Portal User Guide*.

Additional Information

- You need to ensure that the vManager server uses the same file system as the location where the vPlan is saved.
- You need to ensure that the appropriate user or groups permissions are enabled to allow the server user (account name under which the server is running) to access the plan for read or write access. This grants the read or write permissions to the server process owner for the vPlan file.
- Connection errors are indicated by an exclamation mark on the *Setup vManager* icon (). A possible way to resolve these errors is by specifying a sufficient duration for timeout. Connection timeouts take place when the vManager server is not reachable or is not responding.
- You can only save the form with valid server settings. Deselect the *Enabled* check box to proceed without resolving the invalid server settings or connection issues.
  - If you press the *Esc* key or close the *Setup vManager* form, a message box appears asking if you want to *Save*, *Discard*, or *Cancel*.
- You can specify a new vPlan file name to create a new plan using Verifier. See vManager Setup Form.
- You need to ensure that the *Analog* work mode is enabled in the vManager server setup. For vManager to smoothly pass data back and forth, keep the *Analog* work mode enabled. For more information, see Project Configurations -- Settings -- Generic Configurations in *Cadence© vManager™ Web Portal User Guide*.

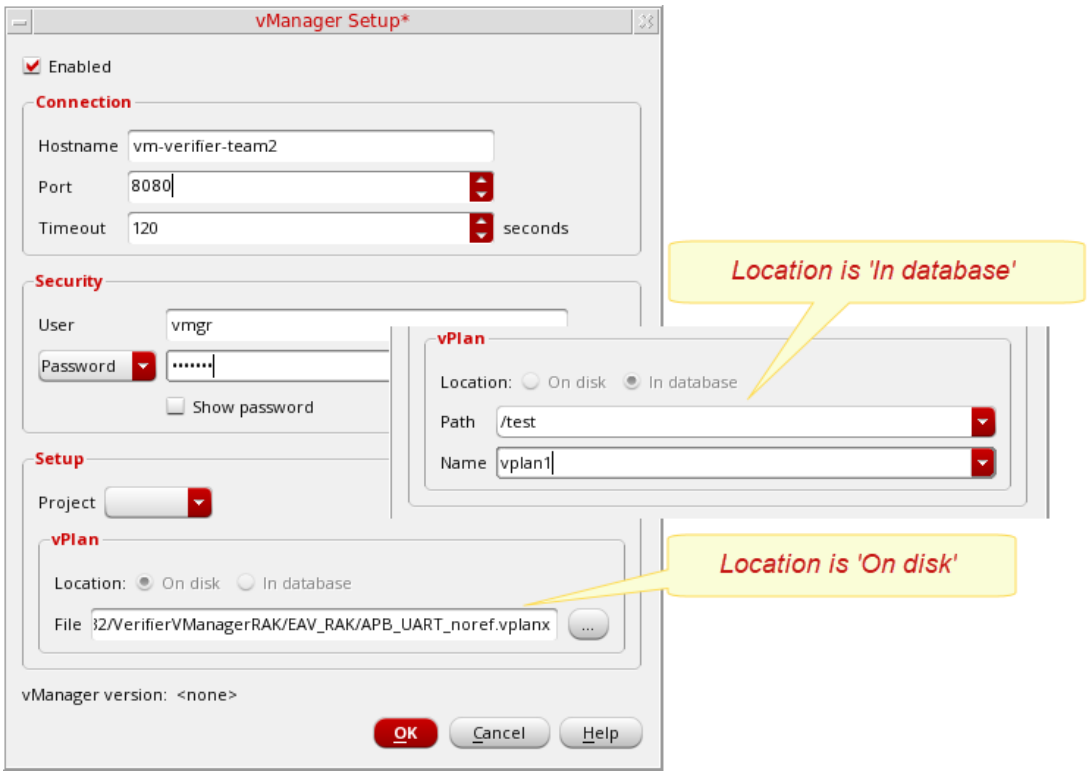
Setting Up the vManager Connection

To specify the vManager configuration settings, do one of the following:

- Click  on the vManager toolbar.
- Select *Tools – vManager – Setup vManager*.

© 2023 All Rights Reserved.

This displays the *vManager Setup* form.



To know more, see the vManager Setup Form form.

Authentication for the vManager Server

You can establish a connection between Verifier and a vManager server using the following authentication methods:

- *Password* - In the Security group box of the vManager Setup form, choose *Password* after specifying the *User*.

  Select *Password* from the list when all users share the same local user login and password to connect with the vManager server. These login credentials are saved as

© 2023
All Rights Reserved.

plain text in the `setings.v3` file in your project directory and are available for public access.



- *Token* - In the Security group box of the vManager Setup form, choose *Token.*

  Select *Token* from the list when you want to log into the vManager server with your UNIX user name. Verifier automatically populates your user name in the *User* field and uses an encrypted token instead of a password. You can get this token from your profile information saved on the vManager server. The vManager server saves this token securely in an encrypted format for every user in their `./cadence` directory. The token is not saved in the `settings.v3` file as it is only applicable to a specific user.



Download of vPlan Requirements on Initial Merge

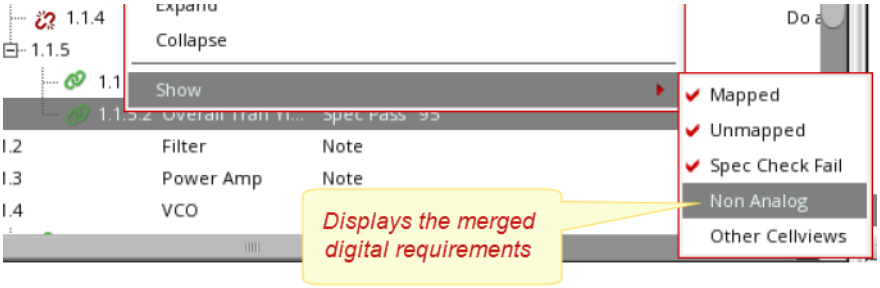
A successful connection between Verifier and vManager displays a message box. The information about the last completed download is displayed in the message box as well as the tooltip on the *Download from vManager* button on the vManager toolbar. For an existing vPlan, Verifier downloads and merges the vPlan requirements with the existing analog plan requirements. Alternatively, you can replace all the analog requirements with the digital requirements.

These vPlan requirements are not downloaded in the following conditions:

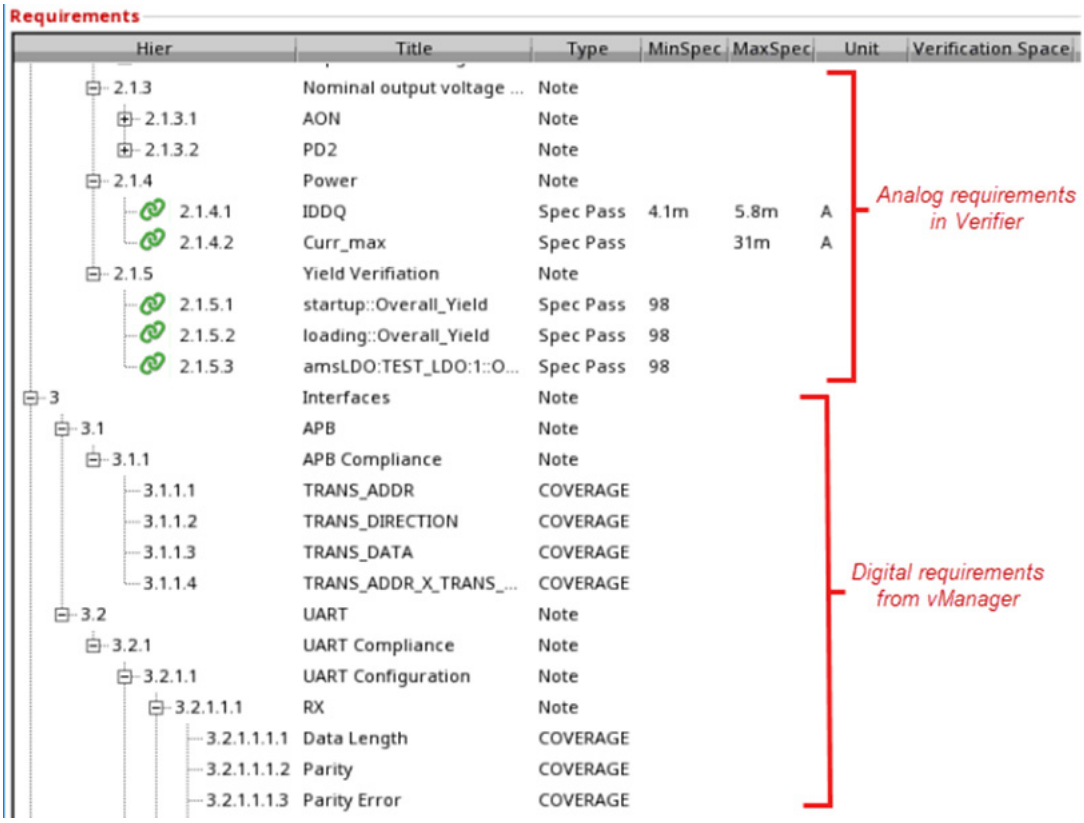
- You have selected not to save changes in the prompt that appears when you choose *File – Make Read Only*.
- You have restored a named snapshot that contains vManager setup information.

© 2023 All Rights Reserved.

To view the merged digital requirements in Verifier, right-click anywhere in the *Requirements* pane and choose *Show – Non Analog*.



This displays the digital requirements from vManager and the analog requirements from Verifier in the *Requirements* pane in Verifier.



Uploading a Verification Plan to vManager

To upload a verification plan from Verifier to the vManager server, do one of the following:

- Click  on the vManager toolbar.

© 2023 All Rights Reserved.

- Click  on the Verifier toolbar.
- Choose *Tools – vManager – Upload to vManager*.

A progress bar at the lower right side in the window displays the upload status.



While uploading, Verifier checks for the vPlan file specified in the vManager server setup. If the vPlan file does not exist on the server, Verifier creates a new vPlan file.

Open the plan in vManager and verify that it contains the changes made in Verifier.

Downloading a vPlan from vManager

To download a vPlan into Verifier, do one of the following:

- Click  on the vManager toolbar.
- Choose *Tools – vManager – Download from vManager*.
- Open a cellview that has been saved with valid vManager settings. This automatically downloads the vPlan.

When you choose to download from vManager, and there are changes in the vPlan, Verifier displays a message asking you to merge, replace or cancel the download. You can make the appropriate choice to merge or replace the analog plan in Verifier. See Download of vPlan Requirements on Initial Merge.

Editing the Verification Plan and vPlan Concurrently

When you choose to download from vManager, Verifier merges the changes from the vPlan. This makes it easy to edit both plans concurrently.

To edit the verification plan and the vPlan concurrently:

1. Open the same plan in vManager and Verifier.
2. Edit the same plan concurrently.
3. In vManager, save any changes to the vPlan.
4. In Verifier, click *Download from vManager* from the vManager toolbar.

© 2023 All Rights Reserved.

   a. Choose Merge on the dialog box that appears. See Download of vPlan Requirements on Initial Merge.

5. Choose *Upload to vManager* from the vManager toolbar to upload the merged changes to the vManager server.
   Reopen the vPlan in vManager to see the latest changes.

Additional Information

- Ensure that you download or upload in the sequence given above because all changes are merged in Verifier, and not in vManager.
- Verifier does not merge changes to the same requirement on upload, but overwrites the requirement in vManager. For example, suppose you download a vPlan to Verifier, edit a requirement within vManager, then edit that same requirement and save the changes in Verifier. This will upload the plan to vManager and overwrite the changes made in vManager. If you edit the same requirement in Verifier and upload it to vManager, all changes made in vManager are overwritten.
- Ensure that you make minimum changes to the hierarchy. Making significant changes can make the merge complex.
- Press the `Esc` key to cancel an ongoing upload or download operation.

Verification Plans in ADE Verifier and vManager Plans

In typical scenarios, Verifier downloads the vPlan and merges the changes to the verification plan. See Download of vPlan Requirements on Initial Merge.

However, a merge of the plans is not required when:

- The Verifier cellview has no requirements but vManager has requirements data.
- The Verifier cellview has requirements but vManager has no requirements data or a new vPlan file is created in vManager.

Requirement IDs in an ADE Verifier-vManager Flow

To identify requirements, Verifier uses requirement IDs, whereas vManager uses a combination of the hierarchical path and title of the requirement. When there is an overlap of requirement IDs in the downloaded requirements, the vManager data is retained and the Verifier data is discarded for the requirement with the overlapped ID.

© 2023 All Rights Reserved.

Additional Information

- Requirement IDs in Verifier must be unique. See Reassigning IDs Hierarchically.
- In case of a conflict where identical requirement IDs and requirement titles are found, requirement IDs are given preference.
- In case of a conflict, vManager data is given preference over Verifier data.

Posting Verifier Results to the vManager Server

After a simulation is run in Verifier, you can upload the simulation results to vManager by doing one of the following:

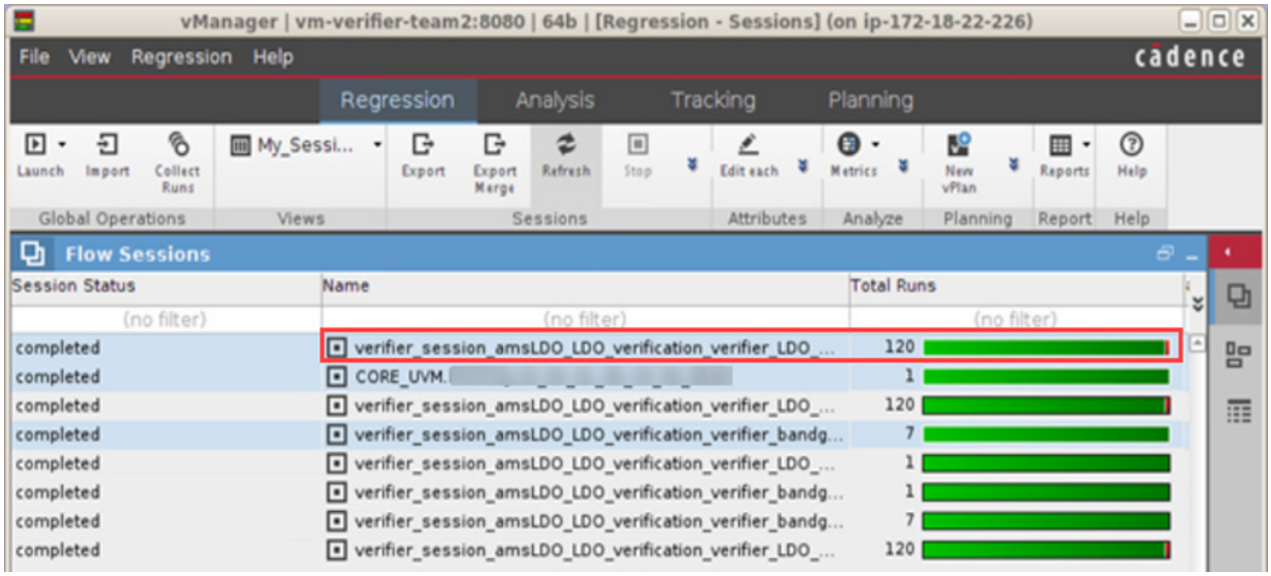
- Click  on the vManager toolbar.
- Choose *Tools – vManager – Post Results to vManager*.

A progress bar on the lower right side in the window displays the upload status.



To cancel an ongoing upload of simulation results, press the `Esc` key.

You can view the uploaded results as a new session in vManager.



© 2023 All Rights Reserved.

Creating a Batch and VSIF Script

You can create a batch and VSIF file in Verifier and use this file to start a Verifier session from vManager.

To create a batch and VSIF file:

1. In Verifier, do one of the following:
   - Click  on the vManager toolbar.
   - Choose *Tools – vManager – Create Batch and VSIF Script*.

   The *Create Batch and VSIF Script* window appears.
2. Specify a name for the script and click *Save*.
3. Open the vManager client, if not already opened.
4. In the vManager window, choose *Regression – Launch* and select the script file from the required location.

The Verifier window appears.

List of Mappings between Verifier Requirement Fields and vManager Attributes

The following table provides a list of the requirement fields in Verifier that are mapped to vManager attributes.

Verifier		vPlan
ID		Element ID
Title		Name
Type	Note	Section
	Spec Pass	Metrics Port
	Ran OK	Metrics Port
	Manual	Metrics Port
	ExtRef	Reference
MinSpec		Verifier Min Spec

© 2023 All Rights Reserved.

Verifier	vPlan
MaxSpec	Verifier Max Spec
Unit	Verifier Unit
Owner	Owner
Scope	Verification Scope
Domain	Verification Domain
Description	Details
Mapped	Mapping pattern

The *Type* field in Verifier displays the goal value of digital requirements that are downloaded from vManager and supports the CHECK, COVERAGE, and TESTCASE digital goal types, also referred to as metric ports in vManager.

For more details, see Attributes Associated with a vPlan Element in *Cadence© vManager™ User Guide*.

ADE Verifier Results in vManager

The verification results are uploaded using the *Post Results to vManager* command from Verifier and are available as run sessions in the *Regression* tab in vManager. Click a run session to view the list of runs in that session. Double-clicking a run from this list displays the corresponding run-specific details. The run is a single mappable item on the Analysis tab that include vManager attributes, errors, warnings or information. The following table describes some of the results that are uploaded and how they relate to vManager run attributes.

Run Attribute in vManager	Description
comment	Details related to the mapped implementation. Some of the details it can include are related to the test, implementation, status, simulation points, passed points, failed points, canceled points, disabled tests or outputs, history, corners, run mode, job control mode, simulation job policy, netlisting job policy, run directory, project directory, library, cell, view, and simulator specified in Verifier. These details are the same as displayed in the Information assistant in Verifier.

© 2023 All Rights Reserved.

Run Attribute in vManager	Description
engine	The value is set to `Virtuoso ADE Verifier`.
full_title	The hierarchical name of the implementation.
job_status	The value is set to `finished`.
verification_scope	The value is set to `default`.

Additionally, the List Tabs section in vManager displays the following information related to errors and warnings from the Verifier run.

Type	Description
Run Error	This section provides details on the run errors.
Name	The value is `Verifier_Fail` or `Verifier_Spec_Check_Fail`.
Description	The description contains the error message displayed for the simulation run, or spec check fail.
Severity	The value is set to `critical`.
Run Warning/Info	This section provides details on the run warnings and information messages.
Name	The value is `Verifier_Pass` , or `Verifier_Not_Mapped`. For requirements signed off in Verifier, the name could be `Verifier_Signed_Off`, indicating that the implementation is mapped to a requirement with signoff, or `Verifier_Signoff_Required`, indicating that the implementation is mapped to a manual requirement that has no signoff.
Description	The description is blank or contains the number of simulation points or the signoff information (xx points, xx passed, xx failed, xx canceled, xx disabled).
Tool	The value is set to `Virtuoso ADE Verifier`.

© 2023 All Rights Reserved.

vManager Attributes - Domain and Scope

Some of the requirements fields in Verifier which are important in the Verifier-vManager connection are as follows:

- Domain
- Scope

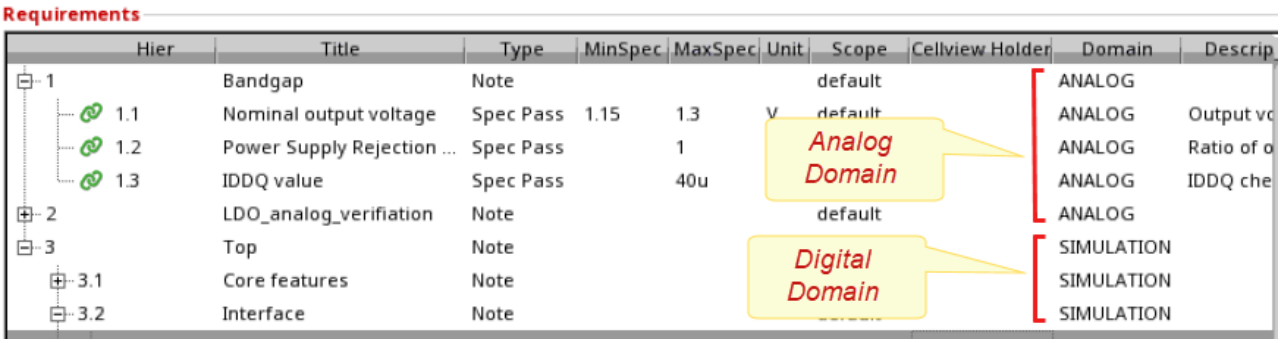
For more details, see Attributes Associated with a vPlan Element in *Cadence vManager User Guide*.

Domain

The verification domain of a requirement specifies if it is an analog or a digital requirement.

In Verifier, all digital requirements are displayed but there is no interaction with these requirements. These digital requirements serve only as information to the user. The only possible action that you can perform is to change the domain of these digital requirements to ANALOG, which makes them useful in Verifier.

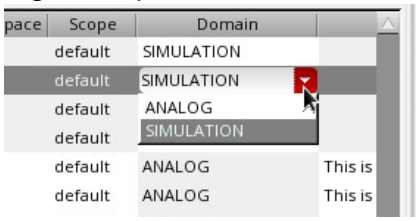
While analog requirements are editable and have the domain 'ANALOG', non-analog, that is, digital requirements are read-only and might have the domains `SIMULATION`, `EMULATION`, `FORMAL`, `NONE`, or more.



In Verifier, you can change the domain of digital requirements to ANALOG in the *Requirements* pane or the *Requirements Editor Assistant*. This indicates that you want to take 'ownership' of the requirements in your verification plan in Verifier. You cannot make

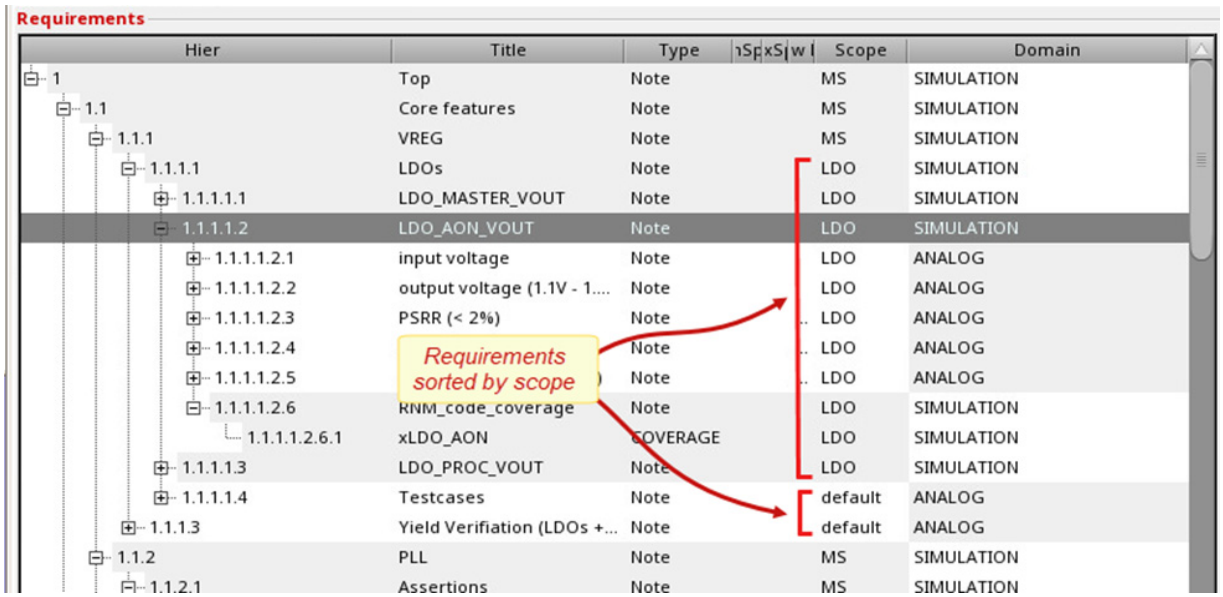
© 2023 All Rights Reserved.

any changes other than domain or type to downloaded digital requirements, but you can edit digital requirements created in Verifier and upload these to vManager.



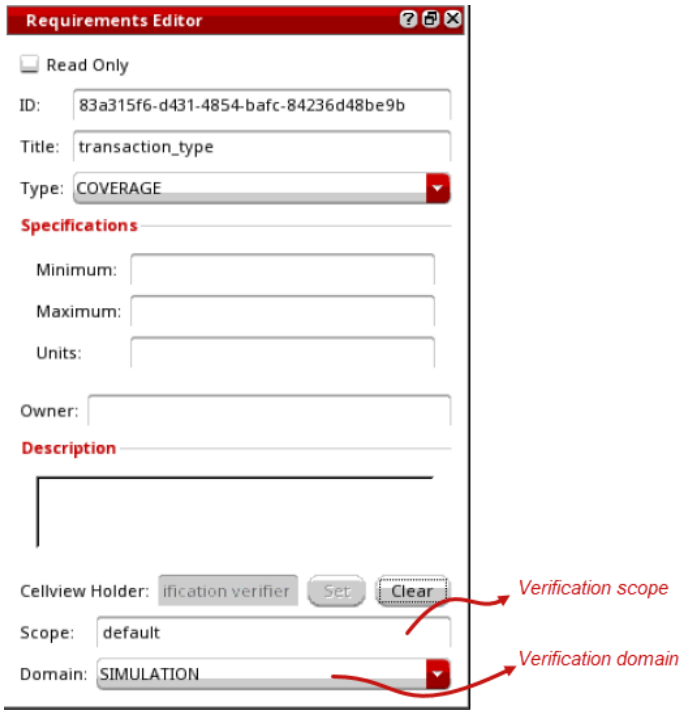
Scope

The verification scope lets you define a group of requirements that belong together. Analog and digital designers can filter the results and hierarchical plan based on the defined scope. The scope is a string field and can contain a user-defined value. The following example shows the usage of scopes `MS`, `LDO`, and `default`.



© 2023
All Rights Reserved.

The following figure shows the domain and scope of a digital requirement in the Requirements Editor assistant in Verifier.



Connecting Multiple Verifier Views to a vPlan

You can connect multiple Verifier views to the same vPlan. Using the *Cellview Holder* field in the *Requirements* pane, you can store the relationship between a cellview and the vPlan by specifying the Verifier view to which a requirement belongs. Verifier shows all requirements of the current cellview, which is the holder, and all requirements that do not have holder information. You can set or clear the holder as the current cellview.

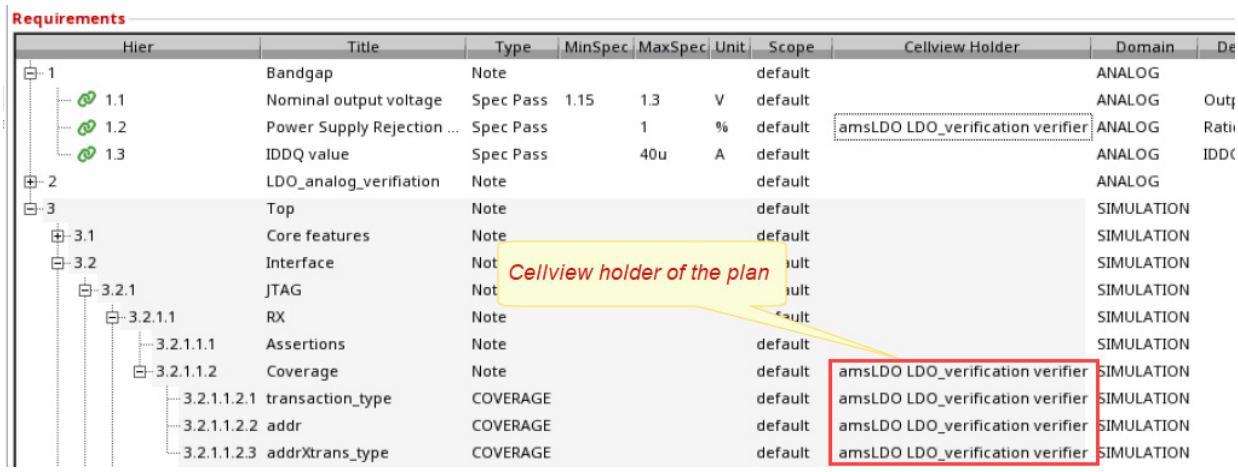
To connect a Verifier view to a vPlan using the Cellview Holder field:

1. Right-click the *Requirements* pane and choose *Set Cellview Holder* from the shortcut menu.
2. Open the *Requirements Editor* assistant and click *Set*.

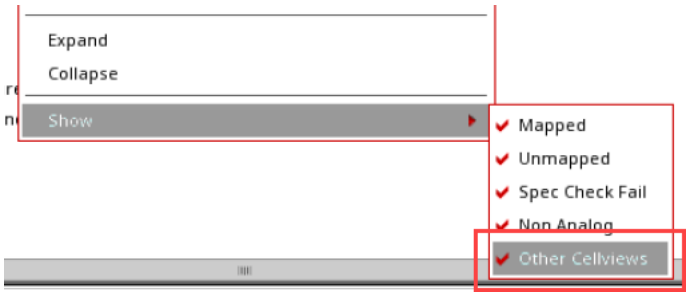
   Cellview holder is set automatically only if vManager is enabled when creating, pasting, or importing requirements. When vManager is disabled you can manually set the

© 2023 All Rights Reserved.

cellview holder. The cellview holder is loaded from the `settings.v3` file.



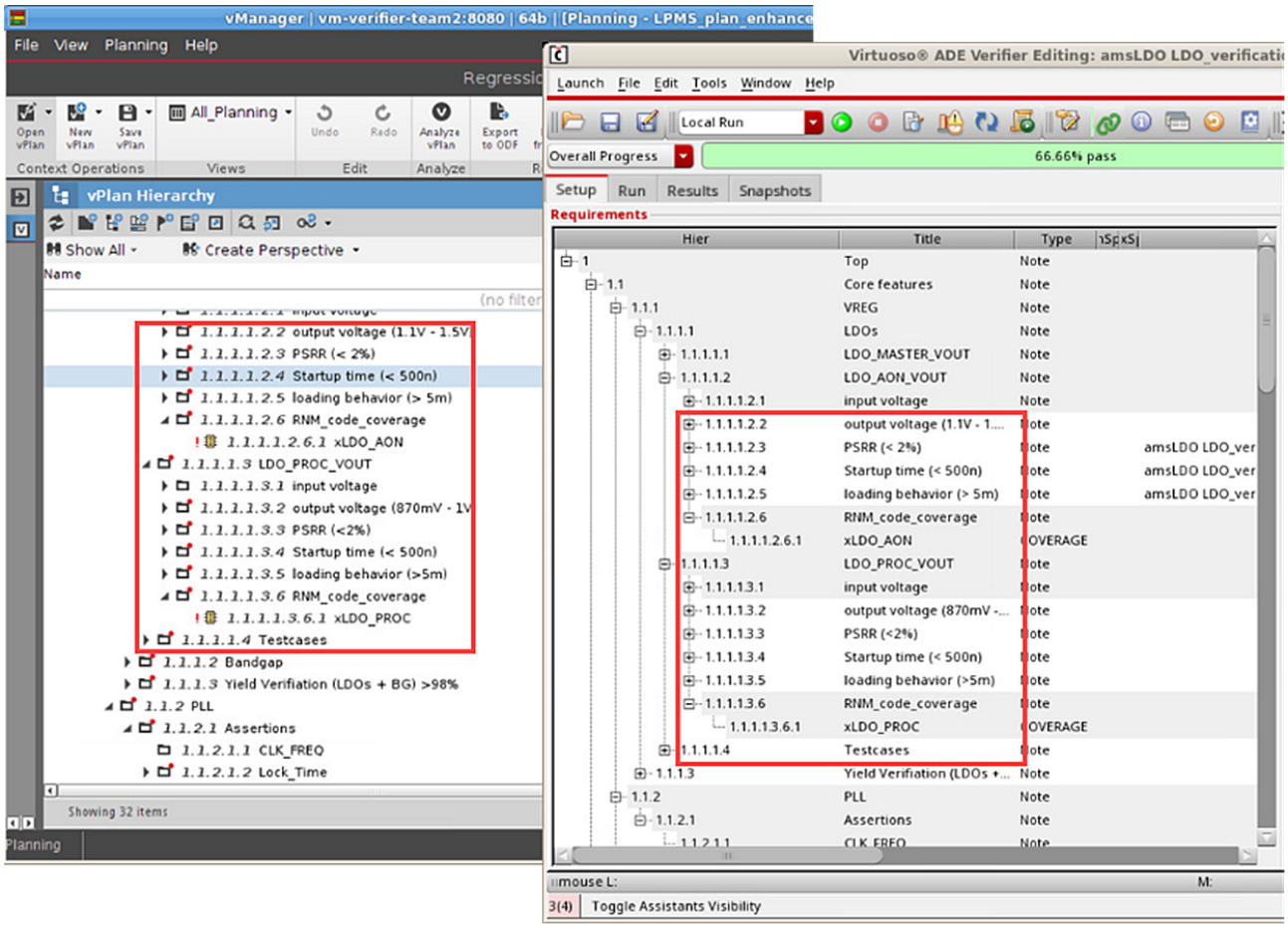
By default you can see requirements owned by other Verifier cellviews in the *Requirements* pane. You can deselect the *Other Cellviews* shortcut menu command to see only the requirements belonging to the current Verifier cellview and items without Holder assignment.



© 2023
All Rights Reserved.

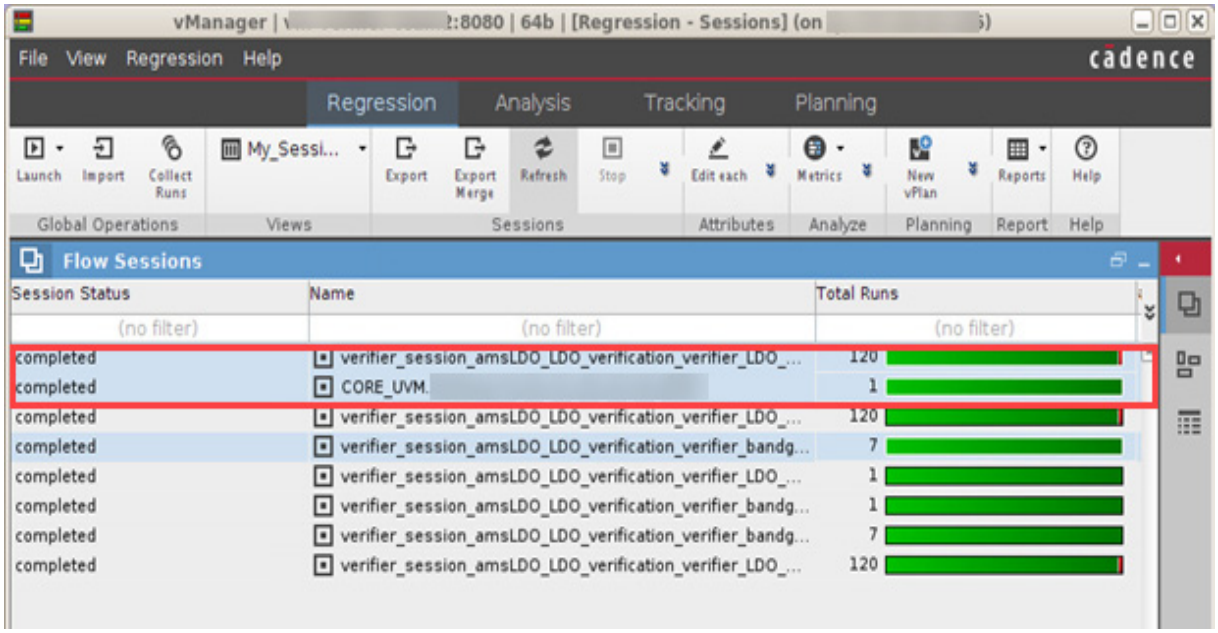
Review of the Verification Results in vManager

When you download and upload requirements, vManager and Verifier show the same hierarchy.

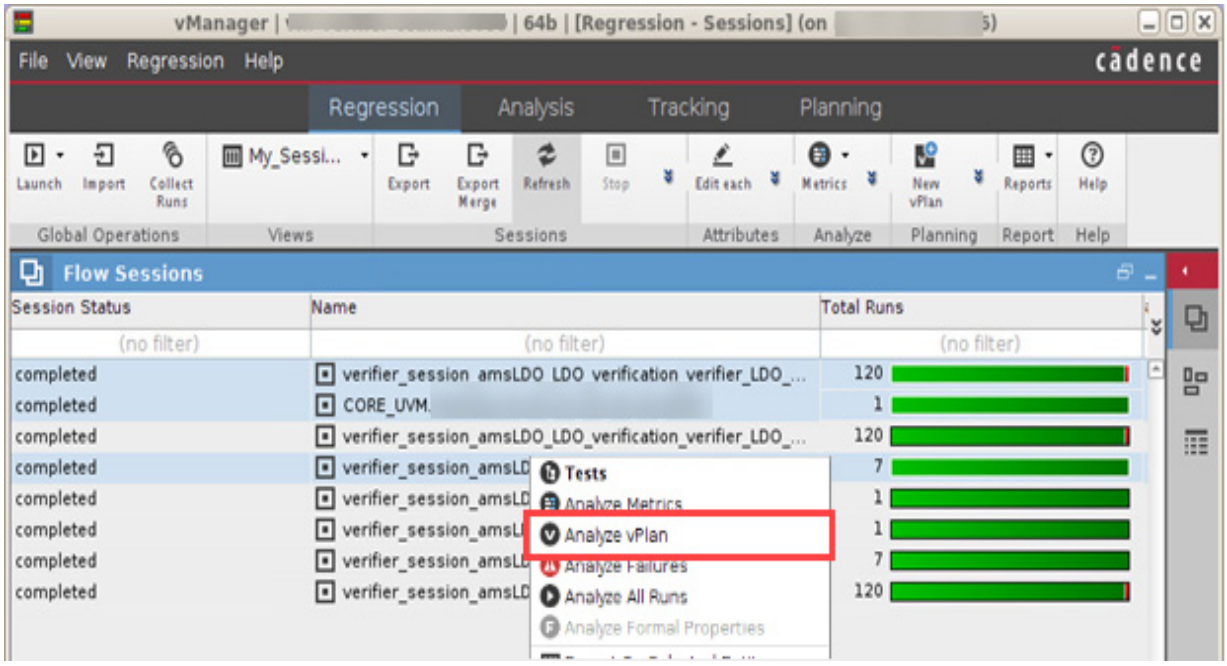


© 2023
All Rights Reserved.

You can run simulations and post the Verifier results to vManager, and the results are displayed in vManager.

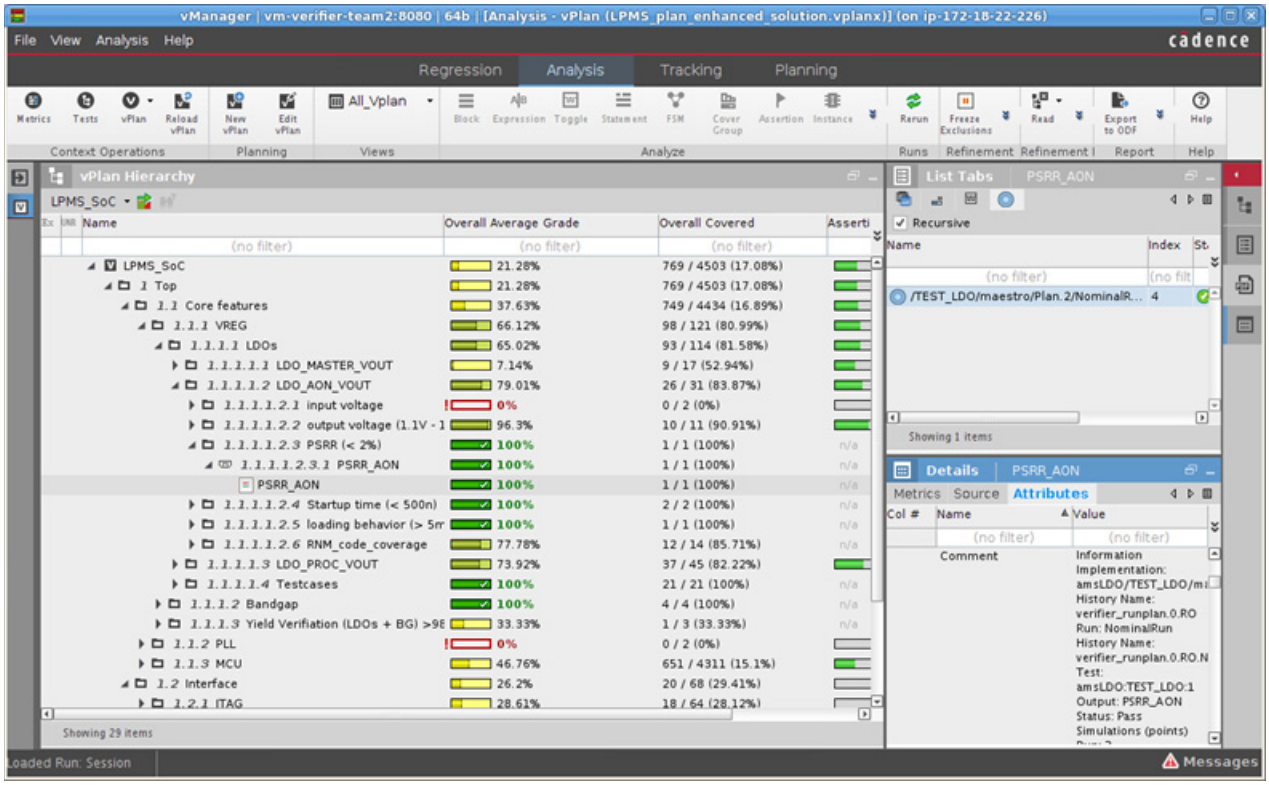


After you post the results to vManager, you can combine the uploaded results and the synchronized vPlan by using the *Analyze vPlan* command in vManager. See Analyzing Verification Plans.



© 2023 All Rights Reserved.

Choosing the Analyze vPlan command displays the analog and digital verification results together in a unified view in vManager.

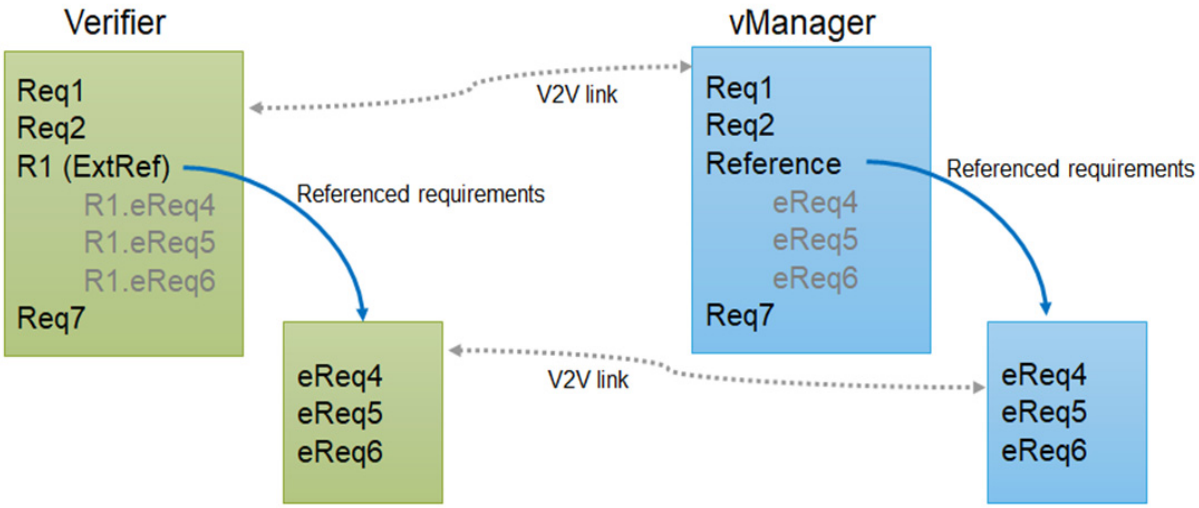


Referenced Cellviews in an ADE Verifier-vManager Session

The connection between Verifier and vManager replicates the 1:1 reference structure. Every Verifier or vManager reference builds its own Verifier-vManager link, ensuring identical structure of the plan, and therefore, avoiding discrepancies. Verifier manages the connection between the cellviews and references in a single, unidirectional connection to vManager.

© 2023
All Rights Reserved.

The following figure shows the flow when there are external references in Verifier and VManager.



Connection between an ADE Verifier Cellview and a vManager Plan with External References

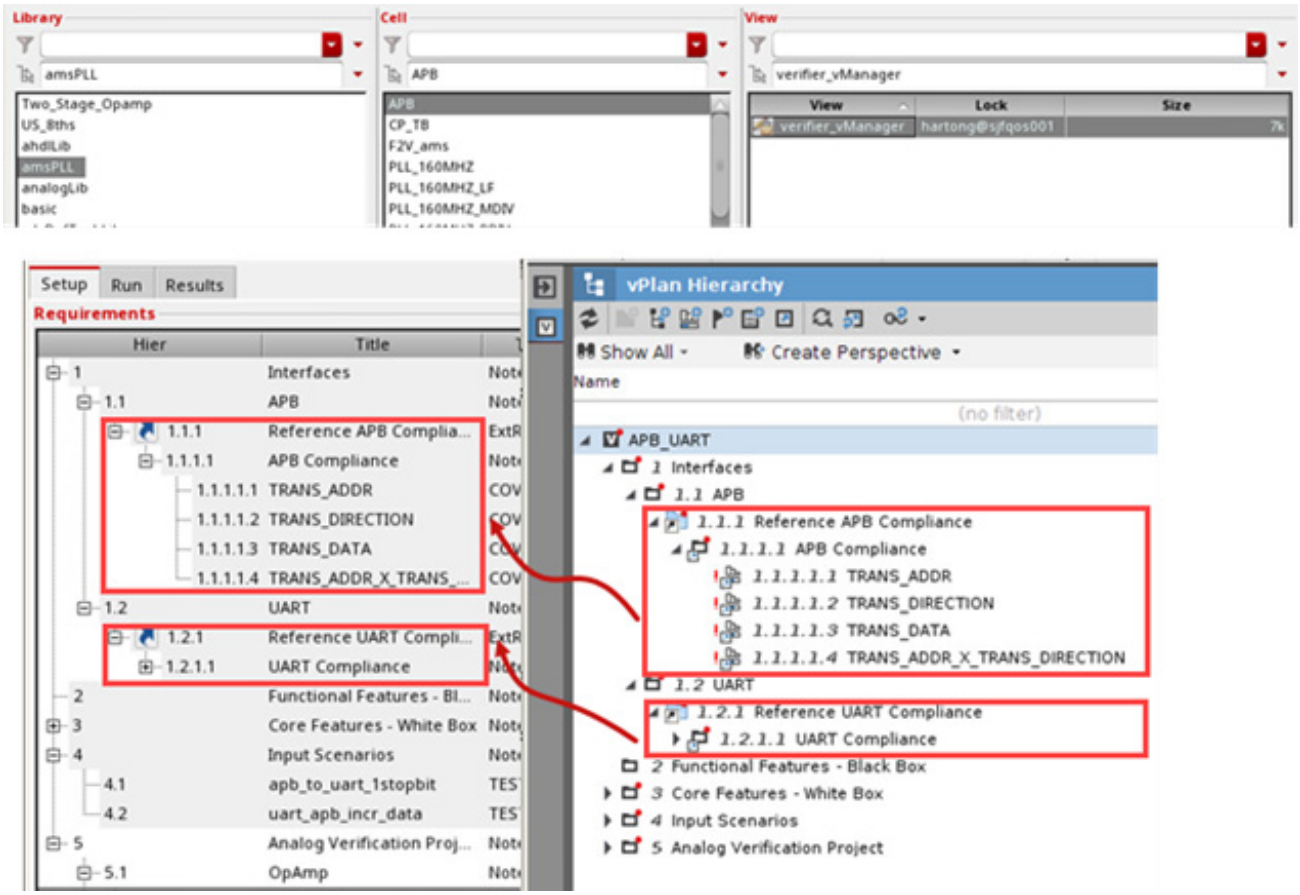
When you initiate a connection from a Verifier cellview to a vPlan that has references to other plans, a new Verifier cellview is created for each referenced vPlan. This new view is created in the same library as the current Verifier cellview. The name of the cellview is `'verifier_vManager'`, which is identical to the reference name.

When you download a reference to the current session, warning messages are shown if the reference:

- Already exists in the current Verifier session
- Contains a reference which already exists in the current Verifier session
- Contains multiple references to the same cellview

© 2023
All Rights Reserved.

The vManager settings automatically replicate the current Verifier settings. This allows showing the same references in Verifier and vManager.



Connection between an ADE Verifier Cellview with External References and a vManager Plan

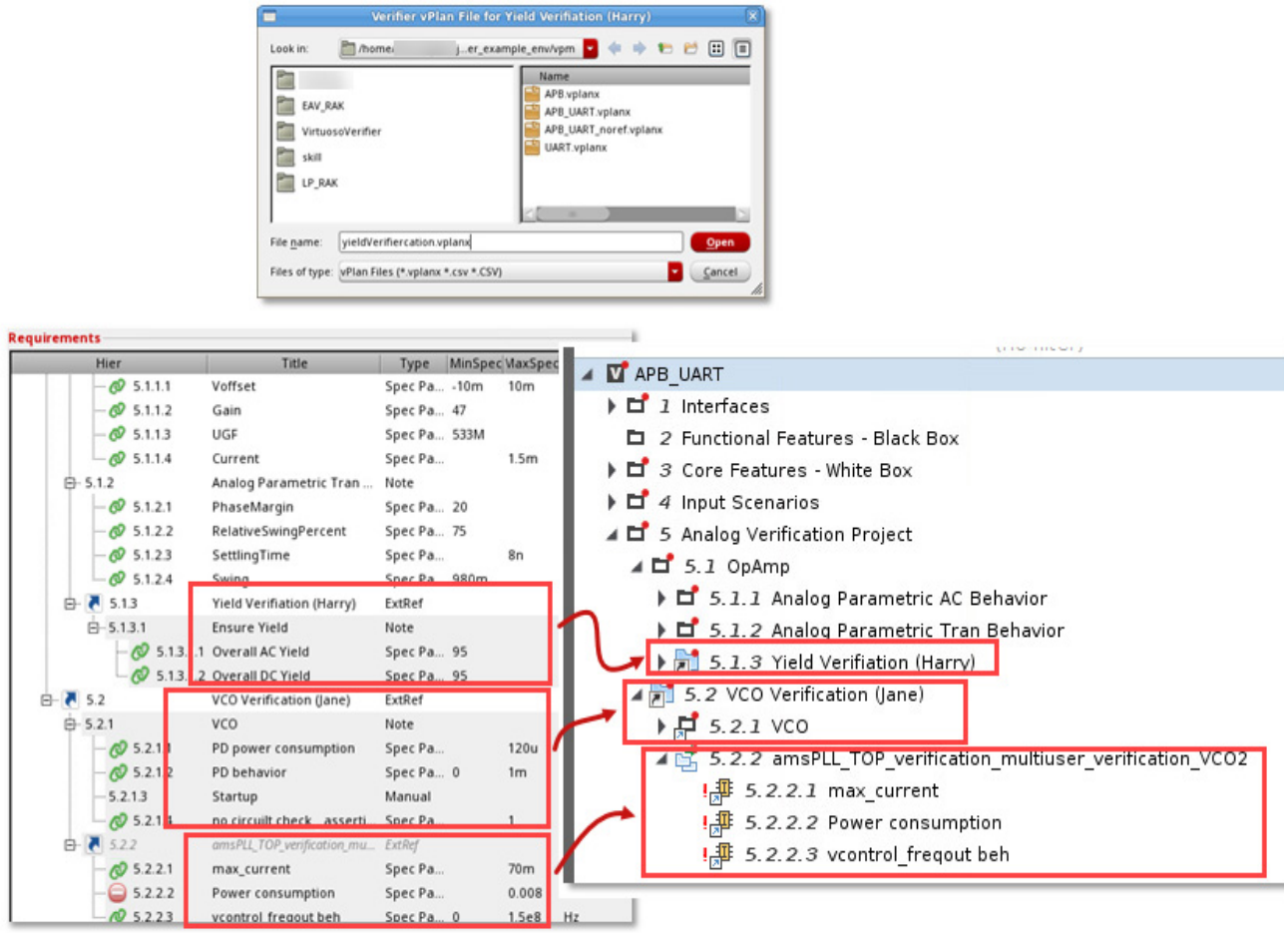
All external references in a Verifier cellview must be either on disk or in the database.

When you initiate a connection from a Verifier cellview that contains external references to a vPlan in vManager, a new vPlan is created in vManager for each cellview referenced from Verifier. When you upload from Verifier the first time, a message box is displayed to let you specify the name of the new vPlan.

If your Verifier setup contains the same `maestro` cellview as an implementation as well as an external reference, both upload the same results and create duplicate result sets in vManager. These duplicate results are mapped to the same entries, which is why vManager merges the these results and you might see two results being mapped to the same vManager entry.

© 2023 All Rights Reserved.

The vManager settings are automatically set in the referenced cellviews, and references in the vPlan are automatically created. This allows showing the same references in Verifier and vManager.



When you import external references to connect to a vPlan, the child requirements of the external reference may be hidden if the *Cellview Holder* property is not set to the current cellview. To resolve this, ensure that you specify the correct settings for *Cellview Holder* or specify the default set of filters using the environment variable filterOptions.

© 2023 All Rights Reserved.

© 2023 All Rights Reserved.

A

Virtuoso ADE Verifier Forms

This topic lists all the forms available in Virtuoso ADE Verifier (Verifier).

- Add Implementation Cellviews Form
- Add Implementation Form
- Add Implementation Set Form
- Change Referenced Cellview Form
- Choose Verifier Cellview Form
- Edit Typical Value Setup Form
- Export Mappings and Results for Requirements in Imported Files Form
- Export Mappings Form
- Export Requirements to CSV File Form
- Export Requirements to Excel File Form
- Import Form
- Import from SDB File Form
- Import Verifier Cellview Form
- Manage Imported Files Form
- Preferences Form
- Save As Form
- Set Requirement Owners Form
- Sign Off Form
- Typical Value Setup Form
- vManager Setup Form

© 2023 All Rights Reserved.

Related Topics

Graphical User Interface of ADE Verifier

© 2023 All Rights Reserved.

Add Implementation Cellviews Form

Use the Add Implementation Cellviews form to select and add multiple implementation cellviews to the current Verifier session.

The following table describes the fields of the Add Implementations Cellview form.

Field	Description
Library	Select one of the listed libraries. By default, Verifier displays all the available libraries.
Cells and Views	This section provides options to specify
Cell list with: ■ *Use* check box for each cellview in the list ■ *Cell* and *View* names ■ *History* drop-down list for each cellview in the list	Select the *Use* check box of the implementation cellviews. Also select the history you want to use with each selected implementation cellview from its The *History* drop-down list. Verifier will display the tests and outputs of the selected history in the implementations pane of the *Setup* tab, which you can map to requirements. You can click *Use All* to select all the available cellviews as the implementation cellviews. Click *Use None* to clear all the selections. You can open a cellview directly from this form. For this, select a cellview entry and click *Open*.
Create Requirements	Select *Create Requirements* to create the requirements hierarchy corresponding to the selected implementation cellviews. The auto-created requirements are also mapped to their corresponding implementations. **Note:** The default ID follows the `IDNumber` format. The title is taken from the cell, test, or output of the implementation. The default type is *Ran OK*. The default type of the requirements of the implementations that have specifications is *Spec Pass*.

Related Topics

Adding Multiple Implementations

© 2023 All Rights Reserved.

Add Implementation Form

Use the Add Implementation form to select and add an implementation cellview to the current Verifier session.

The following table describes the fields of the Add Implementation form.

Field	Description
Library	Select one of the listed libraries. By default, all the available libraries are displayed.
Cell	Select one of the listed cells. By default, all the available cells are displayed.
View	Select one of the listed views. By default, all the available views are displayed. You specify the library, cell, view of the type `maestro`, `adexl`, or `adestate`, and the cell history, to add that cellview as an implementation. If you select an `adexl` or `adestate` cellview, the cellview is first migrated as a `verifier` cellview and then added as an implementation cellview in Verifier.
History	Select one of the listed histories. By default, all the available histories are displayed.
Priority	Select the priority for the simulation run. The default is `0`.
Est Run Time	Select the estimated run time for the implementation. The default is `0`.
Run Plan Mode	Select to enable the run plan mode. By default, Verifier displays the run plan mode check as disabled.
Run	Select to enable the simulation run. By default, Verifier displays the run check as enabled.

Related Topics

Adding an Implementation Manually

© 2023 All Rights Reserved.

Add Implementation Set Form

The Add Implementation Set form lets you create an implementation set, which you can use to organize and run implementation cellviews.

Field	Description
Implementation set name	Lets you specify the name of the new implementation set.

Related Topic

Implementation Sets

© 2023 All Rights Reserved.

Change Referenced Cellview Form

Use the Choose Referenced Cellview form to change the cell view to be referenced. This form is similar to the Import Verifier Cellview Form.

The following table describes the fields of the Choose Referenced Cellview form.

Field	Description
Library	Includes the library containing the Verifier cellview.
Cell	Includes an available cell from the specified library.
View	Includes an available view from the specified library cell.

Note: The *Operation* and *Data* options in this form are disabled.

Related Topics

Updating Referenced Cellviews

© 2023 All Rights Reserved.

Choose Verifier Cellview Form

The Choose Verifier Cellview form lets you include requirements from an existing Verifier cellview or create a new Verifier cellview. You can include the implementations and mappings if your source is an existing Verifier cellview or create a cellview from scratch. This form is similar to the Import Verifier Cellview Form.

The following table describes the fields of the Choose Verifier Cellview form.

Field	Description
Source	This section provides options to specify the source settings for the cellview to be opened.
Library	Includes the library containing the Verifier cellview.
Cell	Includes an available cell from the specified library.
View	Includes available view from the specified library cell.
If you specify a *Cell* or *View* that does not exist in the specified Library, the following options are enabled:	
Create Empty View	Specifies if you want to open an empty Verifier setup.
Import Requirements	Specifies if you want to import only requirements from a *CSV file* or *Excel file*. The Import Form is displayed.
Add Implementations	Specifies if you want to add only implementations from an existing cellview of the type `maestro`. The Edit Typical Value Setup Form is displayed.

Related Topics

Launching Verifier

© 2023 All Rights Reserved.

Edit Typical Value Setup Form

The Edit Typical Value Setup form lets you configure or modify the typical setup based on the current value of the variables, parameters, or model files from the selected implementation cellview. By default, only the global variables and parameters from the selected implementation cellview are shown in this form.

The following table describes the fields available in the Edit Typical Value Setup form.

Field	Description
Corner name	Specifies the enabled corner name that contains the typical related items.
Implementation Typical Setup	This section lists the editable fields that are available in the form.
Name	Specifies the name for the variables, parameters, or model files. The name must be a unique one-line string text without spaces.
CurrentValue	Specifies the value from the global data settings or corner in the implementation cellview.
Typical	By default, the value is taken from the *Tools – Setup Typical Value* form. You can edit it based on the current value.
Shortcut menu commands	This section lists the shortcut menu commands that are available in the form.
Overwrite Typical	Overwrites the value of *Typical* with that of the *Current Value*. Overwrites only the numeric value for variables and parameters.

Related Topics

Setting Up the Typical Value for Implementations

© 2023 All Rights Reserved.

Export Mappings and Results for Requirements in Imported Files Form

The Export Mappings and Results for Requirements in Imported Files form lets you export the mappings and results for Requirements to a CSV or Microsoft Excel file.

The following table describes the fields of the Export Mappings and Results for Requirements in Imported Files form.

Field	Description
Imported Files	This section displays the list of files that have been imported in the current cellview.
File	Displays the relative path to an imported file. The settings in this form can be saved only if a file has been imported in the cellview.
What to Save	This section lets you specify the properties.
Property	Lets you select the properties to be exported. The available properties are;*Parent, ID, Title, Type, MinSpec, MaxSpec, Unit, Owner, Scope, Domain, Description, Deadline, Overall Status, Mapping, History, Result Data Age, Min Value, Typical Value, Max Value, Passed, Failed, No Results, Unmapped, Verification Space, Coverage, CellviewHolder, Disabled, FailCoverage, Hier, NoneCoverage, PassCoverage, ReadOnly, ReadOnly, Signoff, StatusDescription, Yield, Min, Max, Mean, StandardDeviation, SigmaToTarget, Cpk*. The exported file is saved in the following format; `<name_of_imported_file>_<date as mm_dd_yyyy>_<time as hh:mm:ss>.csv`
Summary Information	Adds a summary that includes header row information, author name, the creation date, and a summary of the results above the field information in the exported file.
Export to Setup Library	Saves the exported file in the Setup Library document folder.

© 2023 All Rights Reserved.

Related Topics

Exporting Requirements

© 2023 All Rights Reserved.

Export Mappings Form

The Export Mappings form lets you export the mappings between requirements and implementations to a CSV or Microsoft Excel file.

The following table describes the fields of the Export Mappings form.

Field	Description
CSV file	Exports the mappings to a CSV file.
Excel file	Exports the mappings to a Microsoft Excel file.
Browse	Lets you specify an existing file or a new user-defined filename in CSV or Excel format.

Related Topics

Exporting Mappings

© 2023 All Rights Reserved.

Export Requirements to CSV File Form

The Export Requirements to CSV File form lets you export the requirements in a Verifier cellview to a CSV file.

The following table describes the fields of the Export Requirements to CSV File form.

Field	Description
Imported Files	This section displays the list of files that have been imported in the current cellview.
File	Displays the relative path to an imported file. The settings in this form can be saved only if a file has been imported in the cellview.
What to Save	This section lets you specify the properties.
Property	Lets you select the properties to be exported. The available properties are: *Parent, ID, Title, Type, MinSpec, MaxSpec, Unit, Owner, Scope, Domain, Description, Deadline, Overall Status, Mapping, History, Result Data Age, Min Value, Typical Value, Max Value, Passed, Failed, No Results, Unmapped, Verification Space, Coverage, CellviewHolder, Disabled, FailCoverage, Hier, NoneCoverage, PassCoverage, ReadOnly, ReadOnly, Signoff, StatusDescription, Yield, Min, Max, Mean, StandardDeviation, SigmaToTarget, Cpk*. The exported file is saved in the following format; `<name_of_imported_file>_<date as mm_dd_yyyy>_<time as hh:mm:ss>.csv`
Summary Information	Adds a summary that includes header row information, author name, the creation date, and a summary of the results above the field information in the exported file.
Export to Setup Library	Saves the exported file in the Setup Library document folder.

Related Topics

Exporting Requirements

© 2023 All Rights Reserved.

Export Requirements to Excel File Form

The Export Requirements to Excel File form lets you export the requirements in a Verifier cellview to a Microsoft Excel file.

The following table describes the fields of the Export Requirements to Excel File form.

Field	Description
Imported Files	This section displays the list of files that have been imported in the current cellview.
File	Displays the relative path to an imported file. The settings in this form can be saved only if a file has been imported in the cellview.
What to Save	This section lets you specify the properties.
Property	Lets you select the properties to be exported. The available properties are;*Parent, ID, Title, Type, MinSpec, MaxSpec, Unit, Owner, Scope, Domain, Description, Deadline, Overall Status, Mapping, History, Result Data Age, Min Value, Typical Value, Max Value, Passed, Failed, No Results, Unmapped, Verification Space, Coverage, CellviewHolder, Disabled, FailCoverage, Hier, NoneCoverage, PassCoverage, ReadOnly, ReadOnly, Signoff, StatusDescription, Yield, Min, Max, Mean, StandardDeviation, SigmaToTarget, Cpk*. The exported file is saved in the following format; `<name_of_imported_file>_<date as mm_dd_yyyy>_<time as hh:mm:ss>.csv`
Summary Information	Adds a summary that includes header row information, author name, the creation date, and a summary of the results above the field information in the exported file.
Export to Setup Library	Saves the exported file in the Setup Library document folder.

Related Topics

Exporting Requirements

© 2023 All Rights Reserved.

Import Form

The Import form lets you include requirements from a CSV file or an Excel file. You can also include the implementations and mappings if your source is a Verifier cellview. You can include data by copying them or referencing them from the source.

The following table describes the fields of the Import form.

Field	Description
File	This section provides options to specify the source filename that contains the requirements you want to include during the import process.
CSV file	Includes the requirements from a CSV file. The *File*, *Rows*, *Header Columns*, and *Get Contents From File* group boxes are displayed.
Excel file	Includes the requirements from a Microsoft Excel file. The *File*, *Sheet*, *Rows, Header Columns*, and *Get Contents From File* group boxes are displayed.
Sheet	This section provides options to specify the settings for importing Excel sheets.
Sheet Names	Displays the sheets in the Excel file that will be imported. By default, all sheets are included. This option is displayed when you click *Excel file*. The default is *All*. The value changes when you deselect one of the sheets from the *Sheets in File* group box.
Get Sheets	Retrieves all the non-empty sheets that are available in the Excel file specified in the *File* group box. The included sheets are displayed in the *Sheets in File* group box. This option is displayed when you click *Excel file*. By default, all sheets are included.
Sheets in File	Lets you select one or more of the listed sheets that are retrieved from the specified Excel file. This option is displayed when you click *Excel file*. By default, all sheets are included.

© 2023 All Rights Reserved.

Field	Description
Select All	Lets you select all the non-empty sheets in the Excel file that contains the requirements you want to include. This option is displayed when you select *Excel file* and click *Get Sheets* and the specified file contains non-empty sheets. By default, all sheets are included.
Clear All	Lets you deselect all the non-empty sheets in the Excel file. This option is displayed when you click *Excel file*.
Rows	This section provides options to specify the valid rows of requirements to be imported.
Header row	Displays when you click CSV or Excel file and then choose *Get header definition from row.* Specify the number of row that contains the headers of requirements. By default, Verifier chooses one top header row to be ignored and none of the bottom most rows to be ignored.
Ignore the rows containing invalid properties	Specifies if any rows that contain invalid properties must be ignored during the import process.
Use folded format for composite requirements	Uses the row preceding the header as composite header. The data in folded format is converted into composite requirement structure based on the composite header specified in the file. This option is displayed only when you select the Excel file format for import. Selecting this option disables the *Map source columns to Verifier requirement headers* option.

© 2023 All Rights Reserved.

Field	Description
Header Columns	This section provides options to specify settings related to headers from the imported file and the requirements available in the cellview.
Get header definition from row	Select the row in the source file that contains the headers of the requirements. The headers in the source file must map to the Verifier headers in the given sequence: `Parent,ID,Title,Type,MinSpec,MaxSpec,Unit,Owner,Description` **Note:** When you add the custom fields from Excel and CSV files and use the *Get header definition from row* option, the second value specified for the field definition is used to define the header column. For details, refer to Specifying Values for Custom Fields.
Map source columns to Verifier requirement headers	Select to map the source columns to the Verifier requirement columns. You can choose the number of columns to be included from the *Number of Columns to Be Loaded* group box. You can then map the column number in the source file to the Verifier requirements headers using the mapping table.
Number of Columns to Be Loaded	This section provides options to specify the valid rows of requirements to be imported. This group box is displayed when you select *CSV file* or *Excel file* in the *File* group box and then click *Map source columns to Verifier requirement headers*.
Columns	Lets you specify the number of columns that must be loaded into the Verifier session. The headers of these columns are the key fields from the specified file that are used to create requirements. By default, Verifier is set to import the data from the first nine columns of the source file for the following nine requirement headers. `Parent,ID,Title,Type,MinSpec,MaxSpec,Unit,Owner,Description` You can modify the number of columns as well as the column that must be loaded from the imported file.

© 2023 All Rights Reserved.

Field	Description
Get Columns	Displays all the non-hidden and mapped columns from the first selected or non-empty Excel sheet. The headers of these columns are the key fields from the specified file that are used to create requirements. This option is displayed when you click *Excel file*.
Get Contents From File	This section provides options to view and validate the top-most rows of data from the specified file.
	Lets you view some rows in the source file to ensure that you have specified the correct file and mapping details. The number of rows that will be shown in this field is *Header Row* + 1. This option is displayed when you click *CSV file* or *Excel file*. By default, this option is deselected.

Related Topics

Importing Requirements from File

© 2023 All Rights Reserved.

Import from SDB File Form

Use the Import from SDB File form to import setups from one or more files into the Setup Library.

The following table describes the fields of the Import from SDB File form.

Field	Description
File name	Specifies the name of the SDB file that must be imported in the Setup Library. The field cannot be left empty.
Merge	Merges the new and modified settings from the SDB file into the Setup Library. No visual indicators are displayed.
Compare & Update	Opens the Compare and Update Settings form where a comparison of the Setup Library and the SDB file settings is displayed. All differences are color-coded and appropriate tooltips are displayed.
Done	Closes the Compare and Update Settings form.

Related Topics

Exporting and Importing SDB Setups

© 2023 All Rights Reserved.

Import Verifier Cellview Form

Use the Import Verifier Cellview form to select and add multiple implementation cellviews to the current Verifier session.

The following table describes the fields of the Import Verifier Cellview form.

Field	Description
Library	Includes the library containing the Verifier cellview.
Cell	Includes an available cell from the specified library.
View	Includes an available view from the specified library cell.
Operation	Lets you select one of the available modes for importing a cellview.
Copy	Specifies that the cellview details must be added in edit mode in the current session. The available options for the copy operation are: ■ Implementations and Mappings ■ Requirements
Reference	Specifies that the cellview details must be added in read-only mode in the current session. The following options for the reference operation are enabled by default and cannot be changed: ■ Implementations and Mappings ■ Requirements
Data	Lets you import the selected data while importing a cellview.
Implementations and Mappings	Specifies that all implementations and mappings within the cellview must be imported.
Requirements	Specifies that all requirements within the cellview must be imported.

Related Topics

Adding Multiple Implementations

© 2023 All Rights Reserved.

Manage Imported Files Form

Use the Manage Imported Files form to add, open, delete, and set cellview and file references to be used as the source of requirements and implementations in the current Verifier cellview.

The following table describes the fields of the Manage Imported Files form.

Field	Description
File	Displays the name of the imported file.
Update Action	Specifies the list of actions available for the imported file. ■ *None*: Specifies that no action needs to be taken. ■ *Merge*: Specifies that the variables, parameters, model files, requirements, and implementations from the imported file must be merged into the current cellview. ■ *Compare*: Specifies that the variables, parameters, model files, requirements, and implementations from the imported file must be compared with the current cellview.
Command buttons	This section lists the command buttons that are available in the form.
Add	Lets you add a file by importing it.
Remove	Lets you remove an imported file.
Compare and Merge	Displays the differences in the Show Differences in Requirements for file `relative_filename` form. All differences are displayed in blue text.
View File	Displays the imported file in a dialog box.
Settings	Lets you modify the import settings in the Import form.



Exercise caution when changing the references because it can impact your design verification project. For example, consider that you mapped your requirements with the corresponding implementations in your owner cellview. When you change the** Load Requirements **option to** no **for your owner cellview, the link with the master cellview gets removed and the requirements hierarchy gets deleted. The mappings are also removed.

© 2023 All Rights Reserved.

Related Topics

Copy Requirements from File

© 2023 All Rights Reserved.

Preferences Form

The Preferences form lets you customize Verifier. It organizes the preference options in the following tabs.

The following table describes the tabs available in the Preferences form.

Tab	Description
General	This page includes the general options that are applicable globally in a Verifier session, such as specification search order and coverage.
Planning	This page includes the common options related to planning of requirements and implementations, expiration period of implementations, Monte Carlo statistical outputs, and reliability outputs.
Run	This page includes the miscellaneous options, and options related to history management, and environment variables.
Distribution	This page includes the options related to run-related controls and time.
Reporting	This page includes the options related to creating, opening, and managing reports.

The settings you save in the Preferences form apply to the Verifier session. When you save the session information in a Verifier cellview, the preferences are also saved with the cellview.

In contrast, when you open the Verifier cellview in read-only mode, while you can make changes in the preferences, you cannot save these preferences with the cellview. These modified preferences are only applicable in the current session. When you close this read-only cellview or switch to the edit mode, these preferences are reset.

You can also customize Verifier using its environment variables. Many options in the Preferences form have corresponding environment variables. However, some options can be set only through the environment variables. For example, the *Custom Data* tab becomes available in the Preferences form only when you set the related environment variable `customFieldConfig`.

© 2023 All Rights Reserved.

General

The following table described the preference options that are available on the *General* tab.

Option	Description
Specification Search Order	This section provides options to let you specify the search order to be used for specifications.
Requirement specifications and check to implementation	Check if the specification values set in the requirements match the specification values set in the corresponding implementations. If the values match, Verifier uses those values to determine the verification status of the requirement. Otherwise, the specification check for that requirement fails. ■ The implementation specification is always used when the requirement specification is an expression or a string because Verifier cannot interpret the expressions. For example, if the requirement specification is a valid expression `>(0.5*VAR("vdd"))` and the implementation specification is `>(0.51*VAR("vdd"))`. This is a typical case of `Spec Check Fail` and the implementation specification is used even if the requirement specification is valid expression. ■ Verifier does not check the syntax of expression. You can overwrite the implementation specification with a requirement expression or string specification that has invalid syntax, but the status of that output will be 'Fail' after run. Environment Variable: reqspecscheckunits
Requirement specifications	Use the specification values set in the requirements to determine their verification status. If a requirement does not have any specification values, use the values set in the mapped implementations. Environment Variable: reqspecs
Implementation specifications	Use the specification values set in the implementations to determine the verification status of the mapped requirements. If an implementation does not have any specification values, use the values set in the requirement mapped to that implementation. Environment Variable: reqspecs

© 2023 All Rights Reserved.

Option	Description
Check unit	Select this option to check if the specification units set in the requirements and their corresponding implementations match. ■ All the implementations mapped to a requirement of the type *Spec Pass* must have the same specification values. ■ A check for specification units is only done when both the requirement and the implementation contain a unit value. Environment Variable: reqspecscheckunits
Evaluate Metric Prefix	Specifies settings of the metric prefix for specification comparison and evaluation. If enabled, the metric prefix from the specification unit is used for specification comparison and evaluation. ■ All the implementations mapped to a requirement of the type *Spec Pass* must have the same specification values. ■ By default, the specification units are not used in the specification comparison. They are used for documentation purposes and when the requirement and implementation specifications are compared. For more information, see Evaluation of Metric Prefix. Environment Variable: evaluateMetricPrefix
Coverage	The following options let you specify the coverage settings.
Weighted average for hierarchical coverage	Ensures that weighted average is used to calculate coverage hierarchically. Environment Variable: weightedAverageForCoverage
Show results for Verification Space parameter conditions	Enables result reporting for verification space parameter conditions. Environment Variable: resultForSpaceCoverage

© 2023 All Rights Reserved.

Option	Description
Current User Name	The following option lets you specify the user-specific settings.
	Specifies the current user name that Verifier should consider when processing requirements with owners. The current user name is also the default owner of new requirements. By default, Verifier displays the login identity of the current user in this field. If required, you can change the user name.

Planning

The following table describes the options you can set for implementation runs from the *Planning* tab.

You can access the features to run and monitor simulations from the *Run* tab of Verifier.

Option	Description
Common Options	This section provides options to specify commonly used planning settings.
Show cross mapping at cursor	Indicates the record mapped with the record currently highlighted by the mouse cursor. When this option is enabled, you can view the implementation mapped with a requirement by placing the pointer over the requirement. That requirement and the mapped implementation are highlighted in light gray. Similarly, you can place the pointer over the implementation to highlight the mapped requirement. Environment Variable: enablecrossselection
Map only one requirement to one implementation	Ensures that a requirement can be mapped to only one implementation. Environment Variable: mappingonetoone
Create requirements for outputs with no specification values	Ensures that a requirement is created for outputs for which you have not specified any specification values. Environment Variable: createRequirementsWithNoSpec

© 2023 All Rights Reserved.

Option	Description
Include disabled run, reliability, test, or output for implementation	Includes disabled tests when you import implementations. If you select this check box and choose *Show – Disabled* in the *Requirements* pane, it loads all enabled and disabled runs, reliability analyses, tests, or outputs. If this check box is not selected, it loads only enabled runs, reliability analyses, tests, or outputs. Environment Variable: importDisabledTests
Allow editing the enabled or disabled state for implementation run or test	Enables editing of the enabled or disabled state for implementation runs or tests.
Expiration Period of Implementation Results (in days)	This section provides options to specify the expiration limit for implementation results.
	Specifies the implementation as outdated when the results are older than the specified expiration period. You can specify the value as `0` to disable the expiration check. The upper limit for this field is `365`. Therefore, it always shows the implementation as outdated when the loaded results are older than 365 days. By default, Verifier displays `No expiration period` in this field. If required, you can change the value. Environment Variable: impresexpperiod
Monte Carlo Statistical Outputs	This section provides options to select Monte Carlo statistical outputs during simulation runs.
Yield	Environment Variable: displayStress
Mean	Environment Variable: displaymean
StandardDeviation	Environment Variable: displaystandarddeviation
Cpk	Environment Variable: displaycpk
Min	Environment Variable: displaymin
Max	Environment Variable: displaymax
SigmaToTarget	Environment Variable: displaysigmatotarget

© 2023 All Rights Reserved.

Option	Description
Reliability Outputs	This section provides options to specify settings for reliability outputs.
Show reliability outputs	
Fresh	Displays the reliability outputs for fresh tests. Environment Variable: displayFresh
Stress	Displays the reliability outputs for stress tests. Environment Variable: displayStress
Aged	Displays the reliability outputs for age tests. Environment Variable: displayAge
Show outputs for ages	
All	Displays the reliability outputs for all ages. Environment Variable: displayRelOutputAges
Last	Displays the reliability outputs for the last age. Environment Variable: displayRelOutputAges

Run

The following table describes the options you can set to run and monitor simulations from the *Run* tab.

Option	Description
Miscellaneous	This section provides options to specify miscellaneous settings for simulation runs.
Debug mode, open implementation interface	Runs implementations in debug mode, where the implementation interface, such as ADE Assembler, is displayed to help monitor the runs. When you select this option and run simulations, the ADE Assembler interface stays open. Environment Variable: impuimode

© 2023 All Rights Reserved.

Option	Description
Create ADE Assembler datasheet at run or reload time	Creates a datasheet at run time or reload time. This datasheet is saved and referenced only when you create the HTML report in Verifier. Environment Variable: autoCreateDataSheet
History Name	This section provides options to specify custom history names.
Prefix	Adds a prefix to the ADE Assembler history name. **Note:** The history name is saved in the format *historyNamePrefix*_srsc.0.RO. The default prefix is verifier. The available run modes in ADE Assembler are: ■ Single Run, Sweeps and Corners => srsc ■ Run Plan => runplan ■ Monte Carlo Sampling => mc
Append date	Adds the current date as a suffix to the history name. If enabled, the history name is saved in the format historyNamePrefix_srsc_*date*.0.RO. Example: verifier_srsc_20190812.0.RO.
Append time	Adds the current time as a suffix to the history name. If enabled, the history name is saved in the format historyNamePrefix_srsc_*time*.0.RO. Selecting both *Append date* and *Append time* displays the history name in the format historyNamePrefix_srsc_date_time.0.RO. Example: verifier_srsc_20190812_140245.0.RO.
Report Identical History	This section provides options to check histories in ADE Verifier and ADE Assembler.
Report identical histories before run	Performs implementation runs and loads external results only if there are changes since the last run. Environment Variable: usereportidenticalhistory

© 2023 All Rights Reserved.

Option	Description
Check the following:	
Assembler setup	Performs checks on the Assembler setup and test states. This option is enabled by default.
Design (schematic/ configurations)	Performs checks on schematic and configuration views for each test.
Simulation files (model/ definition files)	Performs checks for model and definition files. Verifier runs implementations based on the settings in the *Check the following items* group box.
Environment Variables	This section provides options to specify settings for defining environment variables.
Enabled	Lets Verifier use the *OverrideValue* for running simulations.
Tool.[Partition]	Specifies the name of the tool and its optional partition.
Name	Specifies the name of the environment variable.
Type	Specifies the type of the environment variable. Possible values: string, int, boolean, cyclic, float, toggle. **Note:** This field is not editable. The value is automatically updated based on the variable specified with *Tool.[Partition]* and *Name*.
CurrentValue	Specifies the currently used value of the environment variable set through the `.cdsenv` or `.cdsinit` file, or through the `envSetVal` SKILL function `()`. **Note:** This field is not editable and the value updates automatically based on the variable specified with *Tool.[Partition]* and *Name*.
OverrideValue	Specifies the value of the environment variable used for the simulation in ADE Verifier. This value overwrites the current value before running a simulation and restores the current value after the simulation is complete.

© 2023 All Rights Reserved.

Distribution

The following table describes the options to generate and display verification reports. You set these options from the *Distribution* tab of the Preferences form.

You access verification results from the *Results* tab of Verifier, and HTML reports from the *Tools* menu.

Option	Description
Run Controls	This section provides options to specify global settings for managing simulation runs using run setups.
Run Setup	Specifies the name of the run setup that controls the rest of the fields in the *Run Controls* group box. It loads existing run setups from the `.cadence` directories found using the Cadence Setup Search File (CSF) mechanism. If a run setup does not exist, you can specify a name in this field and save it as a new run setup.
Save	Saves the specified settings in the *Run Controls* group box as a run setup to the `.cadence` directories found using the Cadence Setup Search File (CSF) mechanism.
Delete	Deletes the selected run setup.
Maximum implementations to run in parallel	Specifies the maximum number of implementations that can run in parallel per Verifier session. Running multiple implementations in parallel might require more resources. Therefore, consider your system capabilities when increasing the maximum number. The default value of this field is `1` because, by default, Verifier is not set to run multiple implementations in parallel. Environment Variable: simparallelcount
Maximum batch runs to start in parallel	Specifies the maximum number of external `virtuoso` processes to launch, where each `virtuoso` process runs Verifier simulations. **Note:** Running multiple batch runs in parallel can require more resources. Therefore, consider your system capabilities when increasing the maximum number. The default value of this field is `1` because, by default, Verifier is not set to run multiple implementations in parallel.

© 2023 All Rights Reserved.

Option	Description
Executable or script for batch runs	Specifies the external program to use when running Virtuoso with *Batch Run*. For new cellviews, this field is populated from the environment variable settings. Existing cellviews use the setting in the preferences. Example: `bsub -K -P "IC:6.1.8:RD:test" -R "select[OSREL==EE60]" -R "rusage[mem=3000]" -W 200:0 -sp 50 -o /tmp/bsub_job_$$ virtuoso` To know more, see Distributing Batch Simulation Runs. The value of this field is also used for other Virtuoso processes started by Verifier, such as child processes started in the project directory setup. To know more, see Histories from the Project Directory Setup. Environment Variable: binaryName
Number of retries	Specifies the number of times Verifier retries running a failed simulation. Clicking the *Stop* button cancels the specified retries. Environment Variable: numberOfRetries

© 2023 All Rights Reserved.

Option	Description
Retry time threshold	Specifies the retry time threshold for a failed simulation run in the `<d><hour>:<minute>:<second>` format. The simulation is retried only if the time taken by the previous simulation is less than the specified value. The default is `00:00:00`. A few examples of the time format are as follows: ■ `1d` (1 day) ■ `42` (42 seconds) ■ `2:15:10` (2 hours, 5 minutes and 10 seconds) ■ `4h` (4 hours) ■ `1d 2:15:10` (1 day, 2 hours, 15 minutes and 10 seconds) Environment Variable: retryTimeThreshold **Note:** The threshold value is considered only when it is greater than 0.
Override job control mode	Overrides the job control mode of all the implementations with the mode selected from the drop-down list. Environment Variable: impJobControlMode **Note:** If you specify `""` for this option, the implementation cellview defines the job control mode using its saved or default settings.
Override simulation job policy	Overrides the simulation job policy for all the implementations that use the ICRP and LSCS modes. Environment Variable: impjobpolicy **Note:** If you specify `""` for this option, the implementation cellview defines the simulation job policy using its saved or default settings.

© 2023 All Rights Reserved.

Option	Description
Override netlisting job policy	Overrides the netlisting job policy for all the implementations that use LSCS mode. Environment Variable: impNetlistingJobPolicy **Note:** If you specify `""` for this option, the implementation cellview defines the netlisting job policy using its saved or default settings.
Estimated Run Time	This section provides options to specify settings related to estimated run time of simulations.
Display estimated run time as:	
Seconds	Displays the estimated run time in seconds. Environment Variable: estRunTimeDisplayFormat
Formatted Time	Displays the estimated run time as formatted time. Environment Variable: estRunTimeDisplayFormat
Automatically update estimated run time	Enables the automatic updating of the *EstRunTime* values after simulation runs are triggered from Verifier.

Reporting

The following table describes the options to generate and display verification reports. You set these options from the *Reporting* tab of the Preferences form.

You access verification results from the *Results* tab of Verifier and HTML reports from the *Tools* menu.

Option	Description
Create and Open Options	This section provides options to specify settings for creating and opening verification reports.
Auto-create all enabled reports	Creates all the enabled reports automatically after each interactive run is completed, or when a data update process is completed after external changes are found. Environment Variable: autocreatereports

© 2023 All Rights Reserved.

Option	Description
Overwrite reports	Overwrites HTML reports without any prompts for confirmation. Environment Variable: overwritereports
Link implementation datasheets to the created HTML report	Creates the ADE Assembler or ADE Explorer datasheets of the implementation cellviews from their history and link the datasheets to the HTML report generated by Verifier. You can click the links in the HTML report to access the datasheets. **Note:** It is possible that Virtuoso Visualization and Analysis XL (ViVA) windows open during the HTML report generation because of the datasheet creation process. You can close the ViVA windows. Environment Variable: linkdatasheets
Append implementation information to the created HTML report	Displays mapping information for the implementations in the HTML report.
Create HTML Report File	This section provides options to specify report generation settings.
	Generates the HTML report. You can specify the report filename and location in the associated field. If you do not specify this information, Verifier uses the default filename, *VerifierCellviewName*.html in the current directory. Environment Variable: createbatchhtmlreport, batchhtmlreport
HTML Report	This section provides options to specify the settings for saving HTML reports.
Directory	Path to the directory where the HTML report is saved. You can specify the report directory in the associated field. If this information is not specified, the default directory path is set to *./lib_cell_view*. Environment Variable: useTemplate, htmlReportDirectory

© 2023 All Rights Reserved.

Option	Description
Datetime suffix format	Customizable suffix format appended to the report directory name to indicate date and time. This field can contain only alphanumeric characters, '.', ':', '-', or '_'. The default format `yyyyMMdd_HHmmss` appends `currentDate_currentTime` to the directory name. Environment Variable: useTemplate, dateTimeSuffixFormat

Related Topics

Setting Virtuoso ADE Verifier Preferences

Verifier Environment Variables

Specifying Values for Custom Fields

Setting Up Run Preferences

Running and Stopping a Batch Script

Detailed Verification Report

Hierarchy for Referenced Cellviews

© 2023 All Rights Reserved.

Save As Form

Use the Save As form to specify the library, cell, and view while saving an existing cellview.The form lets you use filers and categories. It offers the auto-complete text feature when you type some text in a field. If you type invalid text, the input box gets highlighted in red. You can select only a valid view name in the form.

The following table lists the fields available in the Save As form.

Form	Objective
Library	Select one of the listed libraries. By default, all the available libraries are displayed.
Cell	Select one of the listed cells. By default, all the available cells are displayed.
View	Select one of the listed views. By default, only verifier views are available.

Related Topic

Launching Verifier

Verifier Cellviews and Sessions

© 2023 All Rights Reserved.

Set Requirement Owners Form

The Set Requirement Owners form lets you specify the owner name of the selected requirements. Use this form to set the same owner for multiple requirements, instead of specifying the owner for each of those requirements.

Field	Description
Requirement owner names	Lets you specify the name of the owner of the selected requirements. You can specify a comma-separated list of owners. However, Verifier considers only the first name in the list as the owner.

Related Topics

Setting Up Requirement Owners

© 2023 All Rights Reserved.

Sign Off Form

The Sign Off form lets you sign off the requirements of the type *Manual*, and the requirements that failed verification.

The following table describes the fields of the Sign Off form.

Field	Description
Signoff Data	This section provides options to specify all details related to a requirement signoff.
User name	Displays the user name of the signoff.
Lifetime	This section provides options to specify lifetime-related settings for a requirement that needs signoff.
	Select the appropriate option to specify the validity time frame of the signoff: ■ *Once*: The signoff is valid only for a specific set of simulation results and becomes disabled when newer simulation results become available. ■ *Indefinitely*: The signoff is valid for an infinite time. ■ *Until Date*: The signoff is valid up to the date specified in the corresponding field.
Apply Signoff?	Select the appropriate option to save signoff data: ■ *Yes*: Select to store the signoff information after applying the signoff to the requirement. ■ *No*: Select to store the signoff information without applying the signoff to the requirement.
Percentage Complete	This section provides options to specify the percentage of completion for a requirement that needs signoff.
	Enables the percentage list box for specifying the completion percentage of the selected requirement. The specified percentage value is also used to calculate the *OverallStatus* of requirements with type *Note*.
Comments	This section lets you specify comments related to a requirement signoff.
	Lets you specify useful information about the signoff.

© 2023 All Rights Reserved.

Field	Description
Basic Information (Read Only)	This section lets you view the basic information related to a requirement signoff in read-only format.
Owner name	Displays the owner name of the requirement being signed off.
Location	Displays the location of the requirement being signed off.
Date time	Displays the signoff date and time of the requirement being signed off.
Requirement Information (Read Only)	This section lets you view the requirement information related to a requirement signoff in read-only format.
ID	Displays the ID of the requirement being signed off.
Title	Displays the title of the requirement being signed off.
Description	This section lets you view the description of the requirement signoff.
	Displays the description of the requirement being signed off.

Related Topics

Requirements Signoff in Virtuoso ADE Verifier

© 2023 All Rights Reserved.

Typical Value Setup Form

The Typical Value Setup form lets you display the typical setup including the variables, parameters, and model files that are used to match the unique typical point for all implementations.

The following table describes the fields available in the Typical Value Setup form.

Field	Description
Name	Specify a name for the variable, parameter, or model file. The name must be a unique one-line string text without spaces.
Value	Specify a numeric value for the variable or parameter; unique section value for a model file.
Shortcut menu commands	This section lists the shortcut menu commands that are available in the form.
Add	Add the item with a unique name.
Remove	Remove the selected items.
Import from CSV	Import the variables or parameters from a CSV file. The columns headers must define the 'Name,Value' column. The value could be blank or a number.
Export to CSV	Export the typical setup to a CSV file.

Related Topics

Typical Value Setup

© 2023 All Rights Reserved.

vManager Setup Form

The vManager Setup form lets you specify the settings to manage a vManager connection in the current Verifier session.

The following table describes the fields of the vManager Setup form.

Field	Description
Enabled	Enables or disables the vManager connection. When this check box is checked, all fields on the form are editable, and let you perform the various tasks, such as upload and download of data. Deselecting this check box disconnects the connection with vManager and disables all fields on the form.
Connection	This section lets you specify the connection settings.
Hostname	Specify the hostname of vManager server.
Port	Specify the port of vManager server.
Timeout	Specify the duration in seconds after which the connection terminates when the vManager server does not answer.
Security	This section lets you specify the security settings.
User	Specify the user name of the vManager server.
Password/ Token	Specify the password or the token for the user name.
Show Password /Show Token	Select if you want to display the password or the token text instead of the '*' characters.
Setup	This section lets you specify the settings related to the vManager setup.
Project	Specify a project. Project data is populated automatically when a connection to the vManager server is established.
vPlan	
Location	Displays the selected state of the *vPlan in database* option from the vManager server. Possible values: *On disk* (default), *In database*.

© 2023 All Rights Reserved.

Field	Description
File	Specifies the path to an existing vplan file. You can specify a new `*.vplanx` filename to create a new vPlan using Verifier. You can use UNIX shell variables and relative paths to specify the vPlan file name such as: `$HOME/abc.vplanx` This field is displayed when *Location* is set to *On disk*.
Path	Specifies the path to a vPlan in the vManager database. Selecting an existing vPlan path from the list updates the *Name* list with the corresponding vPlan names. By default, the *Path* value is empty and the *Name* list contains names of all the vPlans in the vManager database. This field is displayed when *Location* is set to *In database*.
Name	Specifies the name of a vPlan in the vManager database. You can specify a unique name to create a new vPlan in the vManager database with the specified path. Selecting an existing vPlan name in the *Name* list updates the *Path* field with the corresponding path. This field is displayed when *Location* is set to *In database*.
vManager version	Displays the version number of the vManager server. This is a read-only value.

Related Topics

Setting Up the vManager Connection

© 2023 All Rights Reserved.

B

Verifier Environment Variables

This topic describes the Virtuoso ADE Verifier (Verifier) environment variables that are supported for public use and can help in customizing the default settings using the `.cdsenv` or `.cdsinit` file, or the Virtuoso CIW for a session.

To set an environment variable in the `.cdsenv` file, use the following format:

```
verifier.preferences createbatchlogfile string "YES"
```

To set an environment variable in the `.cdsinit` file or Virtuoso CIW, use the following format:

```
envSetVal("verifier.preferences" "createbatchlogfile" 'string "YES")
```

Verifier environment variables are grouped under the following family.partition:

Family.partition	
Environment Variables to Set Preferences	**Environment Variables to Manage Results**
Environment Variables to Manage Custom Fields	Environment Variables to Manage Requirements
Environment Variables to Manage File Export	Environment Variables to Manage Verifier Runs
Environment Variables to Manage Verifier GUI Settings	Environment Variables to Manage Snapshots
Environment Variables to Manage HTML Reports	Environment Variables to Manage Session Windows
Environment Variables to Manage Implementations	Environment Variables to Manage vManager Settings

© 2023 All Rights Reserved.

Environment Variables to Set Preferences

The following table lists the environment variables grouped under the `verifier.preferences` family.partition:

allowEditingDisabledState	enablecrossselection
appendDate	estRunTimeDisplayFormat
appendTime	evaluateMetricPrefix
autoCreateDataSheet	historyNamePrefix
autocreatereports	htmlReportDirectory
autoUpdateAll	impJobControlMode
batchhtmlreport	impjobpolicy
binaryName	impNetlistingJobPolicy
checkdesigninreportidenticalhistory	importDisabledTests
checksimfilesinreportidenticalhistory	impresexpperiod
createbatchhtmlreport	impuimode
createRequirementsWithNoSpec	linkdatasheets
dateTimeSuffixFormat	mappingonetoone
displayAge	numberOfRetries
displaycpk	overwritereports
displayFresh	retryTimeThreshold
displaymax	reqspecs
displaymean	reqspecscheckunits
displaymin	resultForSpaceCoverage
displayRelOutputAges	simparallelcount
displaysigmatotarget	tagsToOverrideWithVerificationSpace
displaystandarddeviation	usereportidenticalhistory
displayStress	weightedAverageForCoverage
displayyield	

© 2023 All Rights Reserved.

allowEditingDisabledState

```
verifier.preferences allowEditingDisabledState boolean { t | nil }
```

Description

Allows editing the enabled or disabled state of implementation runs or tests.

The default is `nil`.

GUI Equivalent

Command	*Edit – Preferences – Planning*
Field	*Allow editing the enabled or disabled state for implementation run or test*

Examples

```
envGetVal("verifier.preferences" "allowEditingDisabledState")
envSetVal("verifier.preferences" "allowEditingDisabledState" 'boolean t)
```

Related Topics

Allow editing the enabled or disabled state for implementation run or test

© 2023 All Rights Reserved.

appendDate

```
verifier.preferences appendDate boolean { t | nil }
```

Description

Adds the current date as a suffix to the history name.

The default is `t`.

GUI Equivalent

Command	*Edit – Preferences – Run*
Field	*Append date*

Examples

```
envGetVal("verifier.preferences" "appendDate")
envSetVal("verifier.preferences" "appendDate" 'boolean nil)
```

Related Topics

Append date

© 2023 All Rights Reserved.

appendTime

```
verifier.preferences appendTime boolean { t | nil }
```

Description

Adds the current time as a suffix to the history name.

The default is `t`.

GUI Equivalent

Command	*Edit – Preferences – Run*
Field	*Append date*

Examples

```
envGetVal("verifier.preferences" "appendTime")
envSetVal("verifier.preferences" "appendTime" 'boolean nil)
```

Related Topics

Append time

© 2023 All Rights Reserved.

autoCreateDataSheet

```
verifier.preferences autoCreateDataSheet boolean { t | nil }
```

Description

Creates an ADE Assembler datasheet during run or reload implementation, when it is set to `t`. It saves the datasheet file name with the tag `dataSheetFile` in the run summary data files. If the datasheet already exists, it only creates a link to the datasheet during the HTML report generation.

The default is `nil`, which means that the implementation datasheet does not exist and is recreated during the HTML report generation.

GUI Equivalent

Command	*Edit – Preferences – Run*
Field	*Create ADE Assembler datasheet at run or reload time*

Examples

```
envGetVal("verifier.preferences" "autoCreateDataSheet")
envSetVal("verifier.preferences" "autoCreateDataSheet" 'boolean t)
```

Related Topics

Create ADE Assembler datasheet at run or reload time

© 2023 All Rights Reserved.

autocreatereports

```
verifier.preferences autocreatereports string { "NO" | "YES" }
```

Description

When set to `"YES"`, Verifier creates all the enabled reports automatically after each interactive run is completed, or when a data update process is completed after external changes were found.

The default is `NO`.

GUI Equivalent

Command	*Edit – Preferences – Reporting*
Field	*Auto-create all enabled reports*

Examples

```
envGetVal("verifier.preferences" "autocreatereports")
envSetVal("verifier.preferences" "autocreatereports" 'string "YES")
```

Related Topics

Auto-create all enabled reports

© 2023 All Rights Reserved.

autoUpdateAll

```
verifier.preferences autoUpdateAll boolean { t | nil }
```

Description

Automatically updates the implementations to match the updates in the cellviews.

The default is `nil`.

GUI Equivalent

None

Examples

```
envGetVal("verifier.preferences" "autoUpdateAll")
envSetVal("verifier.preferences" "autoUpdateAll" 'boolean t)
```

Related Topics

Simulation Runs Using Auto-Synced Requirements

© 2023 All Rights Reserved.

batchhtmlreport

```
verifier.preferences batchhtmlreport string { "string value" }
```

Description

Specifies the name and location of the verification report in HTML format. If you do not specify the filename and location, and createbatchhtmlreport is set to `"YES"`, Verifier saves the report as *`VerifierCellName`*`.html` in the current directory.

The default is `YES`.

GUI Equivalent

Command	*Edit – Preferences – Reporting*
Field	*Create HTML Report File*

Examples

```
envGetVal("verifier.preferences" "batchhtmlreport")
envSetVal("verifier.preferences" "batchhtmlreport" 'string "./mySimDir/
latestReport.html")
```

Related Topics

Create HTML Report File

© 2023 All Rights Reserved.

binaryName

```
verifier.preferences binaryName string { "program_name" }
```

Description

Specifies the external program to use when running Virtuoso with *Batch Run*.

The default is `virtuoso`.

GUI Equivalent

Command	*Edit – Preferences – Distribution*
Field	*Executable or script for batch runs*

Examples

```
envGetVal("verifier.preferences" "binaryName")
envSetVal("verifier.preferences" "binaryName" 'string "virtuoso")
```

Related Topics

Executable or script for batch runs

© 2023 All Rights Reserved.

checkdesigninreportidenticalhistory

```
verifier.preferences checkdesigninreportidenticalhistory string { "NO" | "YES" }
```

Description

When set to `"NO"`, Verifier does not check the schematic and configuration views of each test to detect changes. It runs the test only if changes were detected.

The default is `NO`.

GUI Equivalent

Command	*Edit – Preferences – Run*
Field	*Detect design changes*

Examples

```
envGetVal("verifier.preferences" "checkdesigninreportidenticalhistory")
envSetVal("verifier.preferences" "checkdesigninreportidenticalhistory" 'string
"YES")
```

Related Topics

Detect design changes

© 2023 All Rights Reserved.

checksimfilesinreportidenticalhistory

```
verifier.preferences checksimfilesinreportidenticalhistory string { "NO" | "YES" }
```

Description

When set to `"NO"`, Verifier does not check the simulation files in the nominal run for Spectre. It runs the associated tests only if changes were detected. This check is not considered if the nominal corner is disabled.

The default is `NO`.

GUI Equivalent

Command	*Edit – Preferences – Run*
Field	*Detect simulation file changes in nominal run*

Examples

```
envGetVal("verifier.preferences" "checksimfilesinreportidenticalhistory")
envSetVal("verifier.preferences" "checksimfilesinreportidenticalhistory" 'string
"YES")
```

Related Topics

Detect simulation file changes in nominal run

© 2023 All Rights Reserved.

createbatchhtmlreport

```
verifier.preferences createbatchhtmlreport string { "NO" | "YES" }
```

Description

When set to `"YES"`, Verifier generates the verification report in HTML.

The default is `YES`.

GUI Equivalent

Command	*Edit – Preferences – Planning*
Field	*Create HTML Report File*

Examples

```
envGetVal("verifier.preferences" "createbatchhtmlreport")
envSetVal("verifier.preferences" "createbatchhtmlreport" 'string "YES")
```

Related Topics

Create HTML Report File

© 2023 All Rights Reserved.

createRequirementsWithNoSpec

```
verifier.preferences createRequirementsWithNoSpec boolean { t | nil }
```

Description

When set to `t`, Verifier creates requirements without any specifications.

When set to `nil`, the Verifier Preferences option *Create requirements for outputs with no specification values* is turned off. Requirements are not created for expressions without any specs in the implementations pane. Additionally, no requirements are added when the shortcut menu option *Create Requirement* is selected in the implementations pane.

The default is `t`.

GUI Equivalent

Command	*Edit – Preferences – Planning*
Field	*Create requirements for outputs with no specification values*

Examples

```
envGetVal("verifier.preferences" "createRequirementsWithNoSpec")
envSetVal("verifier.preferences" "createRequirementsWithNoSpec" 'boolean nil)
```

Related Topics

Create requirements for outputs with no specification values

© 2023 All Rights Reserved.

dateTimeSuffixFormat

```
verifier.preferences dateTimeSuffixFormat string "yyyyMMdd_HHmmss"
```

Description

Specifies the customizable suffix format appended to the report directory name.

The suffix indicates the date and time the report is created and supports only alphanumeric characters, '.', ':', '-', or '_'.

The default format is *yyyyMMdd_HHmmss*, meaning that *<currentDate_currentTime>* is appended to the directory name. For example, if the current date is `November 20, 2021`, and the current time is `10:20:30`, the dateTime suffix will be `20211120_102030`.

GUI Equivalent

Command	*Edit – Preferences – Reporting – HTML Report*
Field	*Datetime suffix format*

Examples

```
envGetVal("verifier.preferences" "dateTimeSuffixFormat")
envSetVal("verifier.preferences" "dateTimeSuffixFormat" 'string
"yyyyMMdd_HHmmss")
```

Related Topics

HTML Reports in Interactive Format

Reporting

© 2023 All Rights Reserved.

displayAge

```
verifier.preferences displayAge boolean { t | nil }
```

Desciption

Enables displaying the reliability outputs for aged tests.

The default is `t`.

GUI Equivalent

Command	*Edit – Preferences – Planning*
Field	*Show reliability outputs*

Examples

```
envGetVal("verifier.preferences" "displayAge")
envSetVal("verifier.preferences" "displayAge" 'boolean nil)
```

Related Topics

Show reliability outputs

© 2023 All Rights Reserved.

displaycpk

```
verifier.preferences displaycpk string { "YES" | "NO" }
```

Description

Displays Cpk in the implementations hierarchy.

The default is `"YES"`.

GUI Equivalent

Command	*Edit – Preferences – Planning – Monte Carlo Statistical Outputs*
Field	*Cpk*

Examples

```
envGetVal("verifier.preferences" "displaycpk")
envSetVal("verifier.preferences" "displaycpk" 'string "NO")
```

Related Topics

Monte Carlo Statistical Outputs

© 2023 All Rights Reserved.

displayFresh

```
verifier.preferences displayFresh boolean { t | nil }
```

Description

Displays the reliability outputs for fresh tests.

The default is `t`.

GUI Equivalent

Command	*Edit – Preferences – Planning*
Field	*Show reliability outputs*

Examples

```
envGetVal("verifier.preferences" "displayFresh")
envSetVal("verifier.preferences" "displayFresh" 'boolean t)
```

Related Topics

Show reliability outputs

© 2023 All Rights Reserved.

displaymax

```
verifier.preferences displaymax string { "NO" | "" }
```

Description

Displays the Max value in the implementations hierarchy.

The default is `"NO"`.

GUI Equivalent

Command	*Edit – Preferences – Planning – Monte Carlo Statistical Outputs*
Field	*Max*

Examples

```
envGetVal("verifier.preferences" "displaymax")
envSetVal("verifier.preferences" "displaymax" 'string "")
```

Related Topics

Monte Carlo Statistical Outputs

© 2023 All Rights Reserved.

displaymean

```
verifier.preferences displaymean string { "NO" | "YES" }
```

Description

Displays mean in the implementations hierarchy.

The default is `"YES"`.

GUI Equivalent

Command	*Edit – Preferences – Planning – Monte Carlo Statistical Outputs*
Field	*Mean*

Examples

```
envGetVal("verifier.preferences" "displaymean")
envSetVal("verifier.preferences" "displaymean" 'string "NO")
```

Related Topics

Monte Carlo Statistical Outputs

© 2023 All Rights Reserved.

displaymin

```
verifier.preferences displaymin string { "NO" | "" }
```

Description

Displays the Min value in the implementations hierarchy.

The default is `"NO"`.

GUI Equivalent

Command	*Edit – Preferences – Planning – Monte Carlo Statistical Outputs*
Field	*Min*

Examples

```
envGetVal("verifier.preferences" "displaymin")
envSetVal("verifier.preferences" "displaymin" 'string "")
```

Related Topics

Monte Carlo Statistical Outputs

© 2023 All Rights Reserved.

displayRelOutputAges

```
verifier.preferences displayRelOutputAges cyclic { "All" | "Last" }
```

Description

Displays the reliability outputs for specified ages.

- `"All"` : The reliability outputs are displayed for all age points.
- `"Last"` : The reliability outputs are displayed for the last age point.

The default is `"All"`.

GUI Equivalent

Command	*Edit – Preferences – Planning*
Field	*Show outputs for ages*

Examples

```
envGetVal("verifier.preferences" "displayRelOutputAges")
envSetVal("verifier.preferences" "displayRelOutputAges" 'cyclic "Last")
```

Related Topics

Show outputs for ages

© 2023 All Rights Reserved.

displaysigmatotarget

```
verifier.preferences displaysigmatotarget string { "NO" | "" }
```

Description

Displays the sigma to target value in the implementations hierarchy.

If there is no specification defined for the output in the maestro cellview, Verifier is not able to get the results for this *SigmaToTarget* value.

The default is `"NO"`.

GUI Equivalent

Command	*Edit – Preferences – Planning – Monte Carlo Statistical Outputs*
Field	*Sigma to Target*

Examples

```
envGetVal("verifier.preferences" "displaysigmatotarget")
envSetVal("verifier.preferences" "displaysigmatotarget" 'string "")
```

Related Topics

Monte Carlo Statistical Outputs

© 2023 All Rights Reserved.

displaystandarddeviation

```
verifier.preferences displaystandarddeviation string { "Yes" | "NO" }
```

Description

Displays standard deviation in the implementations hierarchy.

The default is `"YES"`.

GUI Equivalent

Command	*Edit – Preferences – Planning – Monte Carlo Statistical Outputs*
Field	*Standard Deviation*

Examples

```
envGetVal("verifier.preferences" "displaystandarddeviation")
envSetVal("verifier.preferences" "displaystandarddeviation" 'string "NO")
```

Related Topics

Monte Carlo Statistical Outputs

© 2023 All Rights Reserved.

displayStress

```
verifier.preferences displayStress boolean { t | nil }
```

Description

Displays the reliability outputs for stress tests.

The default is `t`.

GUI Equivalent

Command	*Edit – Preferences – Planning*
Field	*Show reliability outputs*

Examples

```
envGetVal("verifier.preferences" "displaystress")
envSetVal("verifier.preferences" "displayStress" 'boolean nil)
```

Related Topics

Show reliability outputs

© 2023 All Rights Reserved.

displayyield

```
verifier.preferences displayyield string { "YES" | "NO" }
```

Description

Displays yield in the implementations hierarchy.

The default is `"YES"`.

GUI Equivalent

Command	*Edit – Preferences – Planning – Monte Carlo Statistical Outputs*
Field	*Yield*

Examples

```
envGetVal("verifier.preferences" "displayyield")
envSetVal("verifier.preferences" "displayyeild" 'string "NO")
```

Related Topics

Monte Carlo Statistical Outputs

© 2023 All Rights Reserved.

enablecrossselection

```
verifier.preferences enablecrossselection string { "Yes" | "NO" }
```

Description

When set to `"YES"`, you can highlight the implementation mapped with a requirement by placing the pointer over the requirement. Similarly, you can place the pointer over the implementation to highlight the mapped requirement.

The default is `"YES"`.

GUI Equivalent

Command	*Edit – Preferences – Planning*
Field	*Show cross mapping at cursor*

Examples

```
envGetVal("verifier.preferences" "enablecrossselection")
envSetVal("verifier.preferences" "enablecrossselection" 'string "NO")
```

Related Topics

Show cross mapping at cursor

© 2023 All Rights Reserved.

estRunTimeDisplayFormat

```
verifier.preferences estRunTimeDisplayFormat cyclic { "Seconds" | "Formatted Time"
     }
```

Description

Sets the *Display Estimated Run Time as* option to *Seconds* or *Formatted Time*.

The default is `"Seconds"`.

GUI Equivalent

Command	*Edit – Preferences – Distribution*
Field	*Display Estimated Run Time*

Examples

```
envGetVal("verifier.preferences" "estRunTimeDisplayFormat")
envSetVal("verifier.preferences" "estRunTimeDisplayFormat" 'cyclic "Formatted
Time")
```

Related Topics

Simulation Runs based on Priority and Estimated Run Time

© 2023 All Rights Reserved.

evaluateMetricPrefix

```
verifier.preferences evaluateMetricPrefix boolean { t | nil }
```

Description

When set to `t`, Verifier checks the metric prefixes for requirements. This option is hidden when the *Specification Search Order* is set to *Implementation Specification*.

The default is `nil`.

GUI Equivalent

Command	*Edit – Preferences – General*
Field	*Evaluate Metric Prefix*

Examples

```
envGetVal("verifier.preferences" "evaluateMetricPrefix")
envSetVal("verifier.preferences" "evaluateMetricPrefix" 'boolean t)
```

Related Topics

Evaluate Metric Prefix

© 2023 All Rights Reserved.

historyNamePrefix

```
verifier.preferences historyNamePrefix string "prefix_text"
```

Description

Adds the specified prefix to the ADE Assembler history name.

The default is `"verifier"`.

GUI Equivalent

Command	*Edit – Preferences – Run*
Field	*Prefix*

Examples

```
envGetVal("verifier.preferences" "historyNamePrefix")
envSetVal("verifier.preferences" "historyNamePrefix" 'string "verifier")
```

Related Topics

Prefix

© 2023 All Rights Reserved.

htmlReportDirectory

```
verifier.preferences htmlReportDirectory string "./lib_cell_view"
```

Description

Specifies the path to the directory where the HTML report is saved.

The path can contain shell environment variables (`$`) and the home alias (`~/`).

The default is an empty string, meaning that the path is automatically set to *`./lib_cell_view`*.

GUI Equivalent

Command	*Edit – Preferences – Reporting – HTML Report*
Field	*Directory*

Examples

```
envGetVal("verifier.preferences" "htmlReportDirectory")
envSetVal("verifier.preferences" "htmlReportDirectory" 'string "myVDRLppCg")
```

Related Topics

HTML Reports in Interactive Format

Reporting

© 2023 All Rights Reserved.

impJobControlMode

```
verifier.preferences impJobControlMode cyclic { "" | "LSCS" | "ICRP" }
```

Description

Specifies the job control mode that must be used for all the implementation cellviews.

The valid values are:

- `""` does not make any changes to the job control mode.
- `"LSCS"` sets the job control mode to `LSCS` mode.
- `"ICRP"` sets the job control mode to `ICRP` mode.

The default is `""`.

GUI Equivalent

Command	*Edit – Preferences – Distribution*
Field	*Override job control mode*

Examples

```
envGetVal("verifier.preferences" "impJobControlMode")
envSetVal("verifier.preferences" "impJobControlMode" 'cyclic "LSCS")
envSetVal("verifier.preferences" "impJobControlMode" 'cyclic "ICRP")
```

Related Topics

Override job control mode

© 2023 All Rights Reserved.

impjobpolicy

```
verifier.preferences impjobpolicy string { "" | "Maestro Default" }
```

Description

Specifies the run job policy that must be used for all the implementation cellviews.

The valid values are:

- `""` does not make any changes to the job policy.
- `"Maestro Default"` sets the run job policy to the default specified in the ADE Assembler cellview.



Tip

You can define a job policy in ADE Assembler and use it as the default policy in ADE Verifier.

The default is `""`.

GUI Equivalent

Command	*Edit – Preferences – Distribution*
Field	*Override simulation job policy*

Examples

```
envGetVal("verifier.preferences" "impjobpolicy")
envSetVal("verifier.preferences" "impjobpolicy" 'string "Maestro Default")
```

Related Topics

Override simulation job policy

© 2023 All Rights Reserved.

impNetlistingJobPolicy

```
verifier.preferences impNetlistingJobPolicy string { "" |
     "user_defined_job_policy" | "Netlisting Default" }
```

Description

Specifies the netlisting job policy that must be used for all the implementation cellviews that use the LSCS job control mode.

The valid values are:

- `""` does not make any changes to the netlisting job policy.
- `"user_defined_job_policy"` sets the netlisting job policy that you create and specify in ADE Assembler.
- `"Netlisting Default"` sets the netlisting job policy to the default specified in the ADE Assembler cellview.

The default is `""`.

GUI Equivalent

Command	*Edit – Preferences – Distribution*
Field	*Override netlisting job policy*

Examples

```
envGetVal("verifier.preferences" "impNetlistingJobPolicy")
envSetVal("verifier.preferences" "impNetlistingJobPolicy" 'string
"user_defined_job_policy")
envSetVal("verifier.preferences" "impNetlistingJobPolicy" 'string "Netlisting
Default")
```

Related Topics

Override netlisting job policy

© 2023 All Rights Reserved.

importDisabledTests

```
verifier.preferences importDisabledTests boolean { t | nil }
```

Description

Includes disabled tests, runs, or outputs while importing implementations.

The default is `nil`.

GUI Equivalent

Command	*Edit – Preferences – Planning*
Field	*Include disabled run, reliability, test, or output for implementation*

Examples

```
envGetVal("verifier.preferences" "importDisabledTests")
envSetVal("verifier.preferences" "importDisabledTests" 'boolean t)
```

Related Topics

Include disabled run, reliability, test, or output for implementation

© 2023 All Rights Reserved.

impresexpperiod

```
verifier.preferences impresexpperiod string { "numeric_value"}
```

Description

When set to any numeric value greater than `0`, identifies an implementation as outdated if the results are older than the specified expiration period.

The default is `"0"`.

GUI Equivalent

Command	*Edit – Preferences – Planning*
Field	*Expiration Period of Implementation Results (in days)*

Examples

```
envGetVal("verifier.preferences" "impresexpperiod")
envSetVal("verifier.preferences" "impresexpperiod" 'string "50")
```

Related Topics

Expiration Period of Implementation Results (in days)

© 2023 All Rights Reserved.

impuimode

```
verifier.preferences impuimode string { "NO" | "YES" }
```

Description

When set to `"YES"`, Verifier runs the implementations in debug mode, where the implementation interface, such as ADE Assembler, is displayed to monitor the runs.

The default is `"NO"`.

GUI Equivalent

Command	*Edit – Preferences – Run*
Field	*Debug mode, open implementation interface*

Examples

```
envGetVal("verifier.preferences" "impuimode")
envSetVal("verifier.preferences" "impuimode" 'string "YES")
```

Related Topics

Debug mode, open implementation interface

© 2023 All Rights Reserved.

linkdatasheets

```
verifier.preferences linkdatasheets string { "YES" | "NO" }
```

Description

When set to `"YES"`, Verifier creates the ADE Assembler datasheets of the implementation cellviews from their history and links the datasheets to the HTML report generated by Verifier. You can click the links in the HTML report to access the datasheets.

The default is `NO`, which means that the datasheets are not linked to the HTML reports.

The CDS environment variable settings are applicable only for a new Verifier cellview. For the existing Verifier cellview, Verifier reads the preferences option value from the saved cellview and not from the specified CDS environment variable.Therefore, you can enable the *Link implementation datasheets to the created HTML report* check box, and save the Verifier cellview to ensure that the settings are saved for reference in future.

When the `linkdatasheets` and `useTemplate` environment variables are enabled, the datasheets are saved in a new directory `assembler_datasheets`, which is created in the same directory as the interactive HTML report.

GUI Equivalent

Command	*Edit – Preferences – Reporting*
Field	*Link implementation datasheets to the created HTML report*

Examples

```
envGetVal("verifier.preferences" "linkdatasheets")
envSetVal("verifier.preferences" "linkdatasheets" 'string "YES")
```

Related Topics

Link implementation datasheets to the created HTML report

© 2023 All Rights Reserved.

mappingonetoone

```
verifier.preferences mappingonetoone string { "YES" | "NO" }
```

Description

When set to `"YES"`, Verifier ensures that a requirement can be mapped to only one implementation.

The default is `"NO"`.

GUI Equivalent

Command	*Edit – Preferences – Planning*
Field	*Map only one requirement to one implementation*

Examples

```
envGetVal("verifier.preferences" "mappingonetoone")
envSetVal("verifier.preferences" "mappingonetoone" 'string "YES")
```

Related Topics

Map only one requirement to one implementation

© 2023 All Rights Reserved.

numberOfRetries

`verifier.preferences numberOfRetries int` *`numeric_value`*

Description

Specifies the number of times a failed simulation must be retried during a run within the range `0-10`.

The default is `0`.

GUI Equivalent

Command	*Edit – Preferences – Distribution*
Field	*Number of retries*

Examples

```
envGetVal("verifier.preferences" "numberOfRetries")
envSetVal("verifier.preferences" "numberOfRetries" 'int 10)
```

Related Topics

Number of retries

© 2023 All Rights Reserved.

overwritereports

```
verifier.preferences overwritereports string { "NO" | "YES" }
```

Description

When set to `"YES"`, Verifier overwrites the HTML reports without any prompts for confirmation.

The default is "`YES`".

GUI Equivalent

Command	*Edit – Preferences – Reporting*
Field	*Overwrite reports*

Examples

```
envGetVal("verifier.preferences" "overwritereports")
envSetVal("verifier.preferences" "overwritereports" 'string "NO")
```

Related Topics

Overwrite reports

© 2023 All Rights Reserved.

retryTimeThreshold

`verifier.preferences retryTimeThreshold int` *`numeric_value`*

Description

Specifies the threshold time for a simulation rerun as an integer value greater than or equal to `0`.

The default is `0`.

GUI Equivalent

Command	*Edit – Preferences – Distribution*
Field	*Retry time threshold*

Examples

```
envGetVal("verifier.preferences" "retryTimeThreshold")
envSetVal("verifier.preferences" "retryTimeThreshold" 'int 5)
```

Related Topics

Retry time threshold

© 2023 All Rights Reserved.

reqspecs

```
verifier.preferences reqspecs string { "Use Requirement Spec and Check" | "Use
     Requirement Spec" | "Use Implementation Spec" }
```

Description

Sets the search order to identify the specifications that Verifier must use.

- `"Use Requirement Spec and Check"`: Default. Checks if the specification values set in the requirements match the specification values set in the corresponding implementations. If the values match, Verifier uses those values to determine the verification status of the requirement. Otherwise, the specification check for that requirement fails.
- `"Use Requirement Spec"`: Uses the requirement specification. If it is missing, use the implementation specification.
- `"Use Implementation Spec"`: Uses the specification values set in the implementations to determine the verification status of the mapped requirements. If an implementation does not have any specification value, use the values set in the requirement mapped to that implementation.

GUI Equivalent

Command	*Edit – Preferences – General – Specifications Search Order*
Field	■ *Requirement specification and check to implementation* ■ *Requirement specification* ■ *Implementation specification*

Examples

```
envGetVal("verifier.preferences" "reqspecs")
envSetVal("verifier.preferences" "reqspecs" 'string "Use Requirement Spec")
envSetVal("verifier.preferences" "reqspecs" 'string "Use Implementation Spec")
```

Related Topics

Specifications Search Order

© 2023 All Rights Reserved.

reqspecscheckunits

```
verifier.preferences reqspecscheckunits string { "YES" | "NO" }
```

Description

When set to `"YES"`, Verifier checks if the specification unit set in the requirements and their corresponding implementations match. Set this variable when `reqspecs` is set to `"Use Requirement Spec and Check"`.

The default is `"NO"`.

GUI Equivalent

None

Examples

```
envGetVal("verifier.preferences" "reqspecscheckunits")
envSetVal("verifier.preferences" "reqspecscheckunits" 'string "YES")
```

Related Topics

reqspecs (Environment variable)

© 2023 All Rights Reserved.

resultForSpaceCoverage

```
verifier.preferences resultForSpaceCoverage boolean { t | nil }
```

Description

Enables result reporting for verification space parameter conditions.

The default is `t`.

GUI Equivalent

Command	*Edit – Preferences – General*
Field	*Show results for Verification Space parameter conditions*

Examples

```
envGetVal("verifier.preferences" "resultForSpaceCoverage")
envSetVal("verifier.preferences" "resultForSpaceCoverage" 'boolean nil)
```

Related Topics

Show results for Verification Space parameter conditions

© 2023 All Rights Reserved.

simparallelcount

```
verifier.preferences simparallelcount string { "number_of_parallel_runs" }
```

Description

Specifies the maximum number of simulations that can run in parallel.

The default is `"1"`.

GUI Equivalent

Command	*Edit – Preferences – Distribution*
Field	*Maximum implementations to run in parallel*

Examples

```
envGetVal("verifier.preferences" "simparallelcount")
envSetVal("verifier.preferences" "simparallelcount" 'string "3")
```

Related Topics

Maximum implementations to run in parallel

© 2023 All Rights Reserved.

tagsToOverrideWithVerificationSpace

```
verifier.preferences tagsToOverrideWithVerificationSpace string { "corners" |
     "sweeps" | "corners sweeps" }
```

Description

When you select the `Local Run` or `Batch with Space` run modes and run simulations, Verifier merges the corners and sweeps in the active setup with corners and sweeps defined in the Verification Space.

- `corners` (default) specifies that corners from the active setup are overridden by the corners from the verification space during simulation.
- `sweeps` specifies that sweeps from the active setup are overridden by the sweeps from the verification space during simulation.
- `corners sweeps` specifies that corners and sweeps from the active setup are overridden by the corners and sweeps from the verification space during simulation.

To reset the variable to the default behavior:

```
envSetVal("verifier.preferences" "tagsToOverrideWithVerificationSpace" 'string "
")
```

Suppose you have an active setup with corners `C0,C1,C2` and a verification space with corners `C0`,`C1`, and `C3`. Here, `tagsToOverrideWithVerificationSpace` set to `corners` overrides the corners from the active setup. It runs `C0`, `C1`, `C3` from the verification space, and does not run corner C2.

The default behavior of `tagsToOverrideWithVerificationSpace` is not overriding, but merging the corners from the active setup and the verification space. It will merge the corners `C0`, `C1`, `C2`, and `C3`.

GUI Equivalent

None

Examples

```
envGetVal("verifier.preferences" "tagsToOverrideWithVerificationSpace")
envSetVal("verifier.preferences" "tagsToOverrideWithVerificationSpace" 'string
"sweeps")
envSetVal("verifier.preferences" "tagsToOverrideWithVerificationSpace" 'string
"corner sweeps")
```


© 2023 All Rights Reserved.

Related Topics

Importing Sweeps and Corners in the Setup Library

Running Simulations with Verification Spaces

Modification of the Setup Library Cellview

© 2023 All Rights Reserved.

usereportidenticalhistory

```
verifier.preferences usereportidenticalhistory string { "NO" | "YES" }
```

Description

When set to `"YES"`, Verifier performs interactive runs in incremental mode.

The default is `"NO"`.

GUI Equivalent

Command	*Edit – Preferences – Run*
Field	*Report identical histories before run*

Examples

```
envGetVal("verifier.preferences" "usereportidenticalhistory")
envSetVal("verifier.preferences" "usereportidenticalhistory" 'string "YES")
```

Related Topics

Report identical histories before run

© 2023 All Rights Reserved.

weightedAverageForCoverage

```
verifier.preferences weightedAverageForCoverage boolean { t | nil }
```

Description

When set to `t`, Verifier calculates the weighted average for coverage.

The default is `t`.

GUI Equivalent

Command	*Edit – Preferences – General*
Field	*Weighted average for hierarchical coverage*

Examples

```
envGetVal("verifier.preferences" "weightedAverageForCoverage")
envSetVal("verifier.preferences" "weightedAverageForCoverage" 'boolean nil)
```

Related Topics

Weighted average for hierarchical coverage

© 2023 All Rights Reserved.

Environment Variables to Manage Results

The following table lists the environment variables grouped under the `verifier.results` family.partition:

allowLoadingIncompleteHistory
autoMergeDefault
calcStatisticsAcrossCorners
calcStatisticsForNonMC
ignoreError
maxFailures

© 2023 All Rights Reserved.

allowLoadingIncompleteHistory

```
verifier.results allowLoadingIncompleteHistory boolean { t | nil }
```

Description

By default, Verifier does not load external results from an incomplete history. However, when `allowLoadingIncompleteHistory` is set to `t`, Verifier allows loading results of incomplete or in-progress simulations.

The default is `nil`.

GUI Equivalent

None

Examples

```
envGetVal("verifier.results" "allowLoadingIncompleteHistory")
envSetVal("verifier.results" "allowLoadingIncompleteHistory" 'boolean t)
```

Related Topics

Viewing Results from an Incomplete History

© 2023 All Rights Reserved.

autoMergeDefault

```
verifier.results autoMergeDefault cyclic { "None" | "Merge" | "Compare" }
```

Description

Specifies the default behavior when requirements from a file are being imported.

The default is `"None"`.

GUI Equivalent

None

Examples

```
envGetval("verifier.results" "autoMergeDefault")
envSetval("verifier.results" "autoMergeDefault" 'cyclic "Merge")
envSetval("verifier.results" "autoMergeDefault" 'cyclic "Compare")
```

Related Topics

Copy Requirements from File

© 2023 All Rights Reserved.

calcStatisticsAcrossCorners

```
verifier.results calcStatisticsAcrossCorners boolean { t | nil }
```

Description

When `calcStatisticsAcrossCorners` is set to `t`, Verifier calculates the *Min* or *Max* values of statistical outputs from the list of statistical values for each sweep and corner in ADE Assembler.

The default is `nil`, which means that Verifier calculates the *Min* or *Max* values of statistical outputs from the list of statistical values for each sweep "`xxx(summary)`" in ADE Assembler.

GUI Equivalent

None

Examples

```
envGetVal("verifier.results" "calcStatisticsAcrossCorners")
envSetVal("verifier.results" "calcStatisticsAcrossCorners" 'boolean t)
```

Related Topics

Verification Status of Requirements

© 2023 All Rights Reserved.

calcStatisticsForNonMC

```
verifier.results calcStatisticsForNonMC boolean { t | nil }
```

Description

By default, Verifier shows all statistical data such as *Yield*, *Min*, *Max*, *StandardDeviation*, *SigmaToTarget*, and *CPK* for Monte Carlo simulations. However, in some cases it might be useful to calculate such statistical values for other types of simulations. When `calcStatisticsForNonMC` is set to `t`, Verifier calculates, saves, or loads the statistical values to or from the run summary data file for any type of simulation.

The default is `nil`.

GUI Equivalent

None

Examples

```
envGetVal("verifier.results" "calcStatisticsForNonMC")
envSetVal("verifier.results" "calcStatisticsForNonMC" 'boolean t)
```

Related Topics

Verification Status of Requirements

© 2023 All Rights Reserved.

ignoreError

```
verifier.results ignoreError boolean { t | nil }
```

Description

Ignores or includes any kinds of errors, such as simulation, netlist, or evaluation errors, during evaluation of requirements.

The default is `t`, which means that the errors are ignored while evaluating requirement.

GUI Equivalent

None

Examples

```
envGetVal("verifier.results" "ignoreError")
envSetVal("verifier.results" "ignoreError" 'boolean nil)
```

Related Topics

Factors that Impact Verification Status

© 2023 All Rights Reserved.

maxFailures

`verifier.results maxfailures int` *`numeric_value`*

Description

Controls the maximum number of failed tests or outputs that are allowed. The allowed range is any number between 1 and 1000.

The default is `1`.

GUI Equivalent

None

Examples

```
envGetVal("verifier.results" "maxfailures")
envSetVal("verifier.results" "maxfailures" 'int 1000)
```

Related Topics

Results of Failed Outputs or Tests

© 2023 All Rights Reserved.

Environment Variables to Manage File Export

The following table lists the environment variables grouped under the `verifier.export` family.partition:

addSummaryInformation
fields

© 2023 All Rights Reserved.

addSummaryInformation

```
verifier.export addSummaryInformation boolean { t | nil }
```

Description

Specifies if the *Summary Information* field is exported or not in the CSV and Excel files.

The default is `nil`.

GUI Equivalent

Command	*File – Export – CSV*
	File – Export – Excel
	File – Export – Imported File
Field	*Summary Information*

Examples

```
envGetVal("verifier.export" "addSummaryInformation")
envSetVal("verifier.export" "addSummaryInformation" 'boolean t)
```

Related Topics

Exporting Requirements

© 2023 All Rights Reserved.

fields

```
verifier.export fields string "comma_separated_list_of_fieldNames"
```

Description

Specifies the predefined fields to be exported in the CSV and Excel files. This set of fields does not include the *Summary Information* field.

The default is
`"Parent,ID,Title,Type,MinSpec,MaxSpec,Unit,Owner,Scope,Domain,Description,Deadline,Overall Status,Mapping,History,Result Data Age,Min Value,Typical Value,Max Value,Passed,Failed,No Results,Unmapped,Verification Space,Coverage".`

GUI Equivalent

Command	*File – Export – CSV*
	File – Export – Excel
	File – Export – Imported File
Field	*Various*

Examples

```
envGetVal("verifier.export" "fields")
```

```
envSetVal("verifier.export" "fields" 'string
"ID,Title,Type,MinSpec,MaxSpec,Unit,Owner,Scope,Description,Deadline,Overall
Status,Mapping,History,Result Data Age,Min Value,Typical Value,Max
Value,Passed,Failed,No Results,Unmapped")
```

Related Topics

Exporting Requirements

© 2023 All Rights Reserved.

Environment Variables to Manage Requirements

The following table lists the environment variables grouped under the `verifier.requirement` family.partition:

changeParentOnMerge
columnHeadersForImport
deadlineFormat
extRefPrefix
idCreationFunction
updateInheritedMappings

© 2023 All Rights Reserved.

changeParentOnMerge

```
verifier.requirement changeParentOnMerge boolean { t | nil }
```

Description

Controls if the hierarchy is considered when comparing or merging an imported file.

The default is `nil`, which means that the hierarchy is ignored while comparing or merging an imported file.

GUI Equivalent

None

Examples

```
envGetVal("verifier.requirement" "changeParentOnMerge")
envSetVal("verifier.requirement" "changeParentOnMerge" 'boolean t)
```

Related Topics

Copy Requirements from File

© 2023 All Rights Reserved.

columnHeadersForImport

```
verifier.requirement columnHeadersForImport string { "comma-separated-list-of-
     column-names" }
```

Description

Replaces the default column headers with the user-defined column headers for importing requirements from CSV or Excel file in the Import Requirements form.

The defaults are
`"Parent,ID,Title,Type,MinSpec,MaxSpec,Unit,Owner,Description".`

GUI Equivalent

None

Examples

```
envGetVal("verifier.requirement" "columnHeadersForImport")
envSetVal("verifier.requirement" "columnHeadersForImport" 'string
"ID,Title,Type,Unit,Owner,Description")
```

Related Topics

Header Columns

© 2023 All Rights Reserved.

deadlineFormat

```
verifier.requirement deadlineFormat string { "date_format" }
```

Description

Specifies the date format used to display the deadline of a requirement as a string. For example, *dd:mm:yy* or *dd:mm:yyyy*.

The default is `""`.

GUI Equivalent

None

Examples

```
envGetVal("verifier.requirement" "deadlineFormat")
envSetVal("verifier.requirement" "deadlineFormat" 'string "dd:mm:yyyy")
```

Related Topics

Deadline

© 2023 All Rights Reserved.

extRefPrefix

```
verifier.requirement extRefPrefix string { "prefix_string" }
```

Description

Specifies the prefix that is added to the referenced requirement IDs and implementation names to differentiate them from local requirements and implementations.

The default is `"r"`.

GUI Equivalent

None

Examples

```
envSetVal("verifier.requirement" "extRefPrefix")
envSetVal("verifier.requirement" "extRefPrefix" 'string "refx")
```

Related Topics

Hierarchy for Referenced Cellviews

© 2023 All Rights Reserved.

idCreationFunction

```
verifier.requirement idCreationFunction string { "name_of_function" }
```

Description

Specifies a function that can be used to create the requirement ID when you add a new requirement or import the requirement without a defined ID in Verifier. The function name must not have any arguments and must return a string value.

The default is `""`.

GUI Equivalent

None

Examples

```
envGetVal("verifier.requirement" "idCreationFunction")
envSetVal("verifier.requirement" "idCreationFunction" 'string "createMyNewID")
```


© 2023 All Rights Reserved.

updateInheritedMappings

```
verifier.requirement updateInheritedMappings cyclic { "Popup" | "Never" |
     "Always")" }
```

Description

Controls the display of popup messages when mappings of composite requirement are deleted or modified.

- `"Popup"` displays a popup message that requires a `Yes` or `No` input.

  When you modify or delete the mappings of the composite requirement:

  If you select `Yes`, the same operation is done for those child requirements that have the same mappings as that of the composite parent.

  If you select `No`, the mappings of the child requirements are retained.

- `"Never"` displays no popup, and deletes or modifies only the mappings of the composite parent. This behavior is similar to selecting `No` for the `Popup` value.

- `"Always"` displays no popup, and deletes or modifies the mappings of both the composite parent as well as it child requirements that have the same mappings as the parent. This behavior is similar to selecting `Yes` for the `Popup` value.

The default is `"Popup"`.

GUI Equivalent

None

Examples

```
envGetVal("verifier.requirement" "updateInheritedMappings")
envSetVal("verifier.requirement" "updateInheritedMappings" 'cyclic "Never")
envSetVal("verifier.requirement" "updateInheritedMappings" 'cyclic "Always")
```

Related Topics

Overwriting Mappings in Composite Requirements

© 2023 All Rights Reserved.

Environment Variables to Manage Session Windows

The following table lists the environment variables grouped under the `verifier.window` family.partition:

height
width

© 2023 All Rights Reserved.

height

`verifier.window height integer` *`numeric_value`*

Description

Controls the default session window height. The acceptable value is a numeric value greater than `0`.

The default is `""`.

GUI Equivalent

None

Examples

```
envGetVal("verifier.window" "height")
envSetVal("verifier.window" "height" 'int 1050)
```

Related Topics

Workspaces

© 2023 All Rights Reserved.

width

```
verifier.window width int 1
```

Description

Controls the default session window width. The acceptable value is a numeric value greater than `0`.

The default is `1200`.

GUI Equivalent

None

Examples

```
envGetVal("verifier.window" "width")
envSetVal("verifier.window" "width" int 1680)
```

Related Topics

Workspaces

© 2023 All Rights Reserved.

Environment Variables to Manage Verifier Runs

The following table lists the environment variables grouped under the `verifier.run` family.partition:

clearAllTestJobPolicies
distributeImpSet
envVarConfigFile
impReportDirectory
impTimeoutScaleFactor
retryUnfinishedOrErrorPoints
simMode
simulationTimeout
stopBatchRunOnServerClose
stopBatchRunOnServerClose

© 2023 All Rights Reserved.

clearAllTestJobPolicies

```
verifier.run clearAllTestJobPolicies boolean { t | nil }
```

Description

Clears all the test level job policies for an implementation before running a simulation.

The default is `nil`.

GUI Equivalent

None

Examples

```
envGetVal("verifier.run" "clearAllTestJobPolicies")
envSetVal("verifier.run" "clearAllTestJobPolicies" 'boolean t)
```

Related Topics

impJobControlMode (Environment variable)

impjobpolicy (Environment variable)

impNetlistingJobPolicy (Environment variable)

Simulation of Implementations

© 2023 All Rights Reserved.

distributeImpSet

```
verifier.run distributeImpSet boolean { t | nil }
```

Description

Distributes all implementations within an implementation set when the simulation is run in Batch Run mode. All implementations are distributed according to the preference options *Maximum implementations to run in parallel* and *Maximum batch runs to start in parallel*. Additionally, the implementation set must be without any hierarchy or dependencies.

The default is `t`.

GUI Equivalent

None

Examples

```
envGetVal("verifier.run" "distributeImpSet")
envSetVal("verifier.run" "distributeImpSet" 'boolean nil)
```

Related Topics

Maximum implementations to run in parallel

Maximum batch runs to start in parallel

© 2023 All Rights Reserved.

envVarConfigFile

```
verifier.run envVarConfigFile string { "CSV_file_name" }
```

Description

Defines the CSV file name and path for environment variable configuration.

The default is `""`.

GUI Equivalent

Command	*Edit – Preferences – Run*
Field	*Environment Variables*

Examples

```
envGetVal("verifier.run" "envVarConfigFile")
envSetVal("verifier.run" "envVarConfigFile" 'string "./myEnvVars.csv")
```

Related Topics

Environment Variables

© 2023 All Rights Reserved.

impReportDirectory

```
verifier.run impReportDirectory string "directory_name"
```

Description

Specifies the path to the directory where the datasheet is saved when the useTemplate and autoCreateDataSheet environment variables are enabled.

The path can contain shell environment variables (`$`) and the home alias (`~/`).

The default is an empty string, meaning that the path is automatically set to *impReportDirectory*`/assembler_datasheets` and the datasheet is copied to the same directory as the interactive HTML report.

GUI Equivalent

Command	*Edit – Preferences – Reporting – HTML Report*
Field	*Directory*

Examples

```
envGetVal("verifier.run" "impReportDirectory")
envSetVal("verifier.run" "impReportDirectory" 'string "myDir")
```

Related Topics

HTML Reports in Interactive Format

Reporting

© 2023 All Rights Reserved.

impTimeoutScaleFactor

```
verifier.run impTimeoutScaleFactor int numeric_value
```

Description

Specifies the implementation timeout scale factor, where the timeout is based on the *EstRunTime* for each implementation. To enable the timeout, set this environment variable to a numeric value and the *EstRunTime* to a time value greater than zero. The value must be a number between `0` and `1000`.

If the implementation continues to run beyond the specified time, which is the product of the values of `impTimeoutScaleFactor` and *EstRunTime*, the implementation is stopped. This check is started when the run status of the implementation changes to `Running`.

This environment variable is useful to stop unresponsive simulation runs in regressions, so that the regressions can complete without delays. The following are some cases when the simulation runs must be stopped:

- When the simulation is taking an unusual amount of time to complete because of a convergence issue.
- When the simulation cannot complete because of a blockage caused by a firewall or insufficient memory.

By setting this environment variable, you can stop such simulation runs so the whole regression is not delayed because of a single implementation that is not responding anymore.

GUI Equivalent

None

Examples

```
envGetVal("verifier.run" "impTimeoutScaleFactor")
envSetVal("verifier.run" "impTimeoutScaleFactor" 'int 4)
```

Related Topics

EstRunTime

Simulation Runs based on Priority and Estimated Run Time

© 2023 All Rights Reserved.

retryUnfinishedOrErrorPoints

```
verifier.run retryUnfinishedOrErrorPoints boolean { t | nil }
```

Description

Reruns only the unfinished or error points for a failed simulation. Setting this variable to `t` adds the following to the *Batch Mode* script:

```
;; Set batch run options
     _verifBatchRunOptions = list(nil 'numRetries 1 'maxRetries 2 'resHistory
"verifier_srsc_20210120_173238.0.RO")
     ;; Other command lines to run simulation ;;
     ;; Clear batch run options
     _verifBatchRunOptions = 'unbound
```

The default is `t`.

GUI Equivalent

None

Examples

```
envGetVal("verifier.run" "retryUnfinishedOrErrorPoints")
envSetVal("verifier.run" "retryUnfinishedOrErrorPoints" 'boolean t)
```

Related Topics

Number of retries

Retry time threshold

© 2023 All Rights Reserved.

simMode

```
verifier.run simMode cyclic { "Local Run" | "Batch Run" | "Local with SPACE" |
     "Batch with SPACE" }
```

Description

Specifies the default run mode when you open a cellview. This run mode is saved or loaded to and from the `settings.v3` file.

- `"Local Run"` specifies that the simulation is run in the local mode.
- `"Batch Run"` specifies that the simulation is run in the batch mode.
- `"Local with SPACE"` specifies that the simulation is run in the local mode with verification spaces.
- `"Batch with SPACE"` specifies that the simulation is run in the batch mode with verification spaces.

The default is `"Local Run"`.

GUI Equivalent

Command	*Run*
Field	*Run Mode*

Examples

```
envGetVal("verifier.run" "simMode")
envSetVal("verifier.run" "simMode" 'cyclic "Batch Run")
envSetVal("verifier.run" "simMode" 'cyclic "Local with SPACE")
envSetVal("verifier.run" "simMode" 'cyclic "Batch with SPACE")
```

Related Topics

Simulation of Implementations

Running Simulations with Verification Spaces

© 2023 All Rights Reserved.

simulationTimeout

`verifier.run simulationTimeout int` *`number_of_seconds`*

Description

Specifies the simulation as finished with an error message if no update is received from ADE Assembler within the specified `simulationTimeout` seconds, which can be any numeric value between 0 and 1000. If the simulation continues to run, it is not disturbed or aborted.

The default is `0`, which means that there is no timeout.

GUI Equivalent

None

Examples

```
envGetVal("verifier.run" "simulationTimeout")
envSetVal("verifier.run" "simulationTimeout" 'int 200)
```

Related Topics

Simulation of Implementations

Running and Stopping Simulations

© 2023 All Rights Reserved.

stopBatchRunOnServerClose

```
verifier.run stopBatchRunOnServerClose boolean { t | nil }
```

Description

Specifies whether to stop all batch run simulations if the Virtuoso session that launched the batch run is closed.

The default is `nil`.

GUI Equivalent

None

Examples

```
envGetVal("verifier.run" "stopBatchRunOnServerClose")
envSetVal("verifier.run" "stopBatchRunOnServerClose" 'boolean t)
```

Related Topics

Simulation of Implementations

Running and Stopping Simulations

Running and Stopping Implementations in Batch Mode

© 2023 All Rights Reserved.

stopTimeout

`verifier.run stopTimeout int` *`numeric_value`*

Description

Specifies the timeout limit for stopping simulation runs. After this stop time, the simulation is considered to have failed. The acceptable range of stopping timeout is `0-100000`, and the unit is `seconds`.

The default value is `300`.

GUI Equivalent

None

Examples

```
envGetVal("verifier.run" "stopTimeout")
envSetVal("verifier.run" "stopTimeout" 'int 10000)
```

Related Topics

Simulation of Implementations

Running and Stopping Simulations

Running and Stopping Implementations in Batch Mode

© 2023 All Rights Reserved.

Environment Variables to Manage Verifier GUI Settings

The following table lists the environment variables grouped under the `verifier.gui` family.partition:

displayReqHierId
filterOptions
impColumnOrder
impHiddenColumns
impHiddenColumns
resColumnOrder
resHiddenColumns
reqColumnOrder
reqHiddenColumns
runColumnOrder
runHiddenColumns
significantDigits
snapEdColumnOrder
snapEdHiddenColumns

© 2023 All Rights Reserved.

displayReqHierId

```
verifier.gui displayReqHierId boolean { t | nil }
```

Description

Controls the display of hierarchical numbers of requirements in the *Setup* tab.

The default is `t`.

GUI Equivalent

None

Examples

```
envGetVal("verifier.gui" "displayReqHierId")
envSetVal("verifier.gui" "displayReqHierId" 'boolean nil)
```

Related Topics

Exporting Requirements

© 2023 All Rights Reserved.

filterOptions

```
verifier.gui filterOptions string { "comma-separated-list-of-filter-names" }
```

Description

Enables the default set of filters for new Verifier cellviews.

The default is `"Mapped,Unmapped,Spec Check Fail,Passing,Non Analog,Other Cellviews"`.

GUI Equivalent

None

Examples

```
envGetVal("verifier.gui" "filterOptions")
envSetVal("verifier.gui" "filterOptions" 'string "Unmapped,Spec Check Fail,Non Analog,Other Cellviews")
```

Related Topics

Filters in the Requirements Hierarchy

Filtering Requirements Based on Text Criteria

Filtering Requirements Based on Status

Filters in the Implementation List

Filtering Implementations Based on Text Criteria

Filtering Implementations Based on Status

© 2023 All Rights Reserved.

impColumnOrder

```
verifier.gui impColumnOrder string { "comma-separated-list-of-column-names" }
```

Description

Specifies the order in which columns appear in the *Implementations* pane of the *Setup* tab

The default is `"Hier,Run,Output,Library,Cell,View,History,RunPlan,Test,Specification,Unit,Mapped,MappedHier"`.

GUI Equivalent

None

Examples

```
envGetVal("verifier.gui" "impColumnOrder")
```

```
envSetVal("verifier.gui" "impColumnOrder" 'string
"Hier,Run,View,History,RunPlan,Test,Mapped,MappedHier")
```

Related Topics

Details of an Implementation

© 2023 All Rights Reserved.

impHiddenColumns

```
verifier.gui impHiddenColumns string { "comma-separated-list-of-column-names" }
```

Description

Specifies the columns that are hidden by default in the *Implementations* pane of the *Setup* tab.

The default is `"Library,Cell,View,Test,Mapped"`.

GUI Equivalent

None

Examples

```
envGetVal("verifier.gui" "impHiddenColumns")
envSetVal("verifier.gui" "impHiddenColumns" 'string "Cell,View,Test,Mapped")
```

Related Topics

Details of an Implementation

© 2023 All Rights Reserved.

resColumnOrder

```
verifier.gui resColumnOrder string { "comma-separated-list-of-column-names" }
```

Description

Specifies the order in which columns appear in the *Results* tab.

The default is
`"Hier,ID,Title,Type,MinSpec,MaxSpec,Unit,VerificationSpace,Owner,CellviewHolder,Scope,Domain,Deadline,Description,ReadOnly,OverallStatus,MinValue,TypicalValue,MaxValue,Coverage,Yield,Min,Max,Mean,StandardDeviation,SigmaToTarget,Cpk,Output,ImpSpec,ImpUnit,History,Status".`

GUI Equivalent

None

Examples

```
envGetVal("verifier.gui" "resColumnOrder")
```

```
envSetVal("verifier.gui" "resColumnOrder" 'string
"Title,Type,MinSpec,MaxSpec,Unit,VerificationSpace,Owner,Deadline,Description,OverallStatus,MinValue,TypicalValue,MaxValue,Coverage,Output,ImpSpec,ImpUnit,History,Status")
```

Related Topics

Verification Status of Requirements

© 2023 All Rights Reserved.

resHiddenColumns

```
verifier.gui resHiddenColumns string { "comma-separated-list-of-column-names" }
```

Description

Specifies the columns that are hidden by default in the *Results* tab.

The default is
`"ID,Type,VerificationSpace,Owner,CellviewHolder,Scope,Domain,Deadline,Description,ReadOnly,Yield,Min,Max,Mean,StandardDeviation,SigmaToTarget,Cpk"`.

GUI Equivalent

None

Examples

```
envGetVal("verifier.gui" "resHiddenColumns")
envSetVal("verifier.gui" "resHiddenColumns" 'string
"ReadOnly,Yield,Min,Max,Mean,StandardDeviation,SigmaToTarget,Cpk")
```

Related Topics

Verification Status of Requirements

© 2023 All Rights Reserved.

reqColumnOrder

```
verifier.gui reqColumnOrder string { "comma-separated-list-of-column-names" }
```

Description

Controls the display order of the columns in the *Requirements* pane of the *Setup* tab.

The default is
`"Hier,ID,Title,Type,MinSpec,MaxSpec,Unit,VerificationSpace,Owner,CellviewHolder,Scope,Domain,Deadline,Description,ReadOnly,Mapped"`.

GUI Equivalent

None

Examples

```
envGetVal("verifier.gui" "reqColumnOrder")
```

```
envSetVal("verifier.gui" "reqColumnOrder" 'string
"Title,Hier,ID,Type,Description,Mapped,MinSpec,MaxSpec,Unit,VerificationSpace,Owner,CellviewHolder,Scope,Domain,Deadline,ReadOnly")
```

Related Topics

Details of a Requirement

© 2023 All Rights Reserved.

reqHiddenColumns

```
verifier.gui reqHiddenColumns string { "comma-separated-list-of-column-names" }
```

Description

Specifies the columns that are hidden in the *Requirements* pane of the *Setup* tab.

The default is
`"ID,Owner,CellviewHolder,Scope,Domain,Deadline,ReadOnly,Mapped"`.

GUI Equivalent

None

Examples

```
envGetVal("verifier.gui" "reqHiddenColumns")
envSetVal("verifier.gui" "reqHiddenColumns" 'string
"CellviewHolder,Scope,Domain,Deadline,ReadOnly")
```

Related Topics

Details of a Requirement

© 2023 All Rights Reserved.

runColumnOrder

```
verifier.gui runColumnOrder string { "comma-separated-list-of-column-names" }
```

Description

Specifies the order in which columns appear in the *Run* tab.

The default is
`"Hier,Library,Cell,View,History,RunName,Run,Go,Stop,Status,ID,Title,VerificationSpace,PreRunScript"`.

GUI Equivalent

None

Examples

```
envGetVal("verifier.gui" "runColumnOrder")
```

```
envSetVal("verifier.gui" "runColumnOrder" 'string
"ID,Hier,Title,Status,RunName,Run,Library,Cell,View,History,Go,Stop,VerificationSpace,PreRunScript")
```

Related Topics

Implementation Runs in ADE Verifier

© 2023 All Rights Reserved.

runHiddenColumns

```
verifier.gui runHiddenColumns string { "comma-separated-list-of-column-names" }
```

Description

Specifies the columns that are hidden by default in the *Run* tab.

The default is `"Library,Cell,View,History,RunName,Priority,EstRunTime"`.

GUI Equivalent

None

Examples

```
envGetVal("verifier.gui" "runHiddenColumns")
envSetVal("verifier.gui" "runHiddenColumns" 'string
"RunName,Priority,EstRunTime")
```

Related Topics

Implementation Runs in ADE Verifier

© 2023 All Rights Reserved.

significantDigits

```
verifier.gui significantDigits int numeric_value
```

Description

Sets the default rounding off schema for the significant digits to calculate overall status and progress. The acceptable value is any numeric value greater than `2`.

The default is 4.



Important

For *Overall Progress*, the percentage of progress bar takes 2 significant digits for text and 4 significant digits for tool tip as default. For the *OverallStatus* column on *Results* tab, the percentage takes 4 significant digits as default. The percentage for *pass* rate is rounded down whereas the percentage for *fail*, *unmapped*, and *no results* is rounded up.

GUI Equivalent

None

Examples

```
envGetVal("verifier.gui" "significantDigits")
envSetVal("verifier.gui" "significantDigits" 'int 3)
```

Related Topics

Overall Progress, Coverage, and Overall Coverage

© 2023 All Rights Reserved.

snapEdColumnOrder

```
verifier.gui snapEdColumnOrder string { "comma-separated-list-of-column-names"
     }
```

Description

Specifies the column order in the Snapshots Editor assistant.

The default is `"Visible,Name,Comment,Created,User,Locked"`.

GUI Equivalent

None

Examples

```
envGetVal("verifier.gui" "snapEdColumnOrder")
envSetVal("verifier.gui" "snapEdColumnOrder" 'string
"Name,Comment,Visible,Created,User,Locked")
```

Related Topics

Snapshots User Interface in ADE Verifier

© 2023 All Rights Reserved.

snapEdHiddenColumns

```
verifier.gui snapEdHiddenColumns string { "comma-separated-list-of-column-
     names" }
```

Description

Specifies the hidden columns in the Snapshots Editor assistant.

The default is `"Created,User"`.

GUI Equivalent

None

Examples

```
envSetVal("verifier.gui" "snapEdHiddenColumns")
envSetVal("verifier.gui" "snapEdHiddenColumns" 'string "Comment,User,Locked")
```

Related Topics

Snapshots User Interface in ADE Verifier

© 2023 All Rights Reserved.

Environment Variables to Manage HTML Reports

The following table lists the environment variables grouped under the `verifier.html` family.partition:

includeReferenceIDPrefix
includeRequirementId
useTemplate
templateName

© 2023 All Rights Reserved.

includeReferenceIDPrefix

```
verifier.html includeReferenceIDPrefix boolean { t | nil }
```

Description

Hides the ExtRef IDs of referenced cellviews in the HTML reports of Verifier cellviews.

The default is `nil`.

GUI Equivalent

None

Examples

```
envGetVal("verifier.html" "includeReferenceIDPrefix")
envSetVal("verifier.html" "includeReferenceIDPrefix" 'boolean t)
```

Related Topics

HTML Reports in Interactive Format

© 2023 All Rights Reserved.

includeRequirementId

```
verifier.html includeRequirementId boolean { t | nil }
```

Description

Shows the requirement ID in the Result Information section of the HTML report.

The default is `nil`.

GUI Equivalent

None

Examples

```
envGetVal("verifier.html" "includeRequirementId")
envSetVal("verifier.html" "includeRequirementId" 'boolean t)
```

Related Topics

HTML Reports in Interactive Format

© 2023 All Rights Reserved.

useTemplate

```
verifier.html useTemplate boolean { t | nil }
```

Description

Displays the HTML report in a template-based interactive format.

The default is `t`. Setting this variable to `nil` displays the HTML report in the basic format.

GUI Equivalent

None

Examples

```
envGetVal("verifier.html" "useTemplate")
envSetVal("verifier.html" "useTemplate" 'boolean nil)
```

Related Topics

HTML Reports in Interactive Format

HTML Reports in Basic Format

© 2023 All Rights Reserved.

templateName

```
verifier.html templateName string "report-file-names"
```

Description

Specifies the name of the template file used for report generation. It is also the name of the output file.

The default is `index.html`.

GUI Equivalent

None

Examples

```
envGetVal("verifier.html" "templateName")
envSetVal("verifier.html" "templateName" 'string "myFile.txt")
```

Related Topics

HTML Reports in Interactive Format

HTML Reports in Basic Format

Customization of HTML Reports in Interactive Format

Customizing the Interactive HTML Report Template

Examples of Interactive HTML Report Customization

© 2023 All Rights Reserved.

Environment Variables to Manage Custom Fields

The following table lists the environment variables grouped under the `verifier.customField` family.partition:

csvfile (customFieldConfig)
csvfile (customReqFieldConfig)

© 2023 All Rights Reserved.

csvfile (customFieldConfig)

```
verifier.customFieldConfig csvfile string { "filename-and-location" }
```

Description

Specifies the CSV filename and location that contains the custom project fields.

The default is `""`.

GUI Equivalent

None

Examples

```
envGetVal("verifier.customFieldConfig" "csvfile")
envSetVal("verifier.customFieldConfig" "csvfile" 'string "./
myProjCustomFields.csv")
```

Related Topics

Custom Fields

© 2023 All Rights Reserved.

csvfile (customReqFieldConfig)

```
verifier.customReqFieldConfig csvfile string { "filename-and-location" }
```

Description

Specifies the CSV filename and location that contains the custom fields for the requirements in a verification project.

The default is `""`.

GUI Equivalent

None

Examples

```
envGetVal("verifier.preferences" "csvfile")
envSetVal("verifier.preferences" "csvfile" 'string "./myReqCustomFields.csv")
```

Related Topics

Custom Fields

© 2023 All Rights Reserved.

Environment Variables to Manage Snapshots

The following table lists the environment variables grouped under the `verifier.snapshots` family.partition:

ignoreDisabledElements
ignoreImpSetupElements
ignoreImpStateElements
impElementOrderSensitivity
includeImplementations
toleranceProperties

© 2023 All Rights Reserved.

ignoreDisabledElements

```
verifier.snapshots ignoreDisabledElements boolean { t | nil }
```

Description

Specifies whether disabled elements must be ignored during the comparison of implementations from snapshots.

When set to `t`, the disabled elements, such as tests, runs, or corners are ignored. These elements can include the tests that are enabled in the source or the reference cellview but disabled in the destination as well as during comparison. Also, they can include the tests that are disabled in the source or the reference cellview but enabled in the destination or during comparson. During comparison, the corners and runs are marked as different with their enabled or disabled state at the top level. All the different child elements from disabled tests, corners, and runs are ignored.

The default is `t`.

GUI Equivalent

None

Examples

```
envGetVal("verifier.snapshots" "ignoreDisabledElements")
envSetVal("verifier.snapshots" "ignoreDisabledElements" 'boolean nil)
```

Related Topics

Ignoring Elements or their Order when Comparing Implementations from Snapshots

© 2023 All Rights Reserved.

ignoreImpSetupElements

```
verifier.snapshots ignoreImpSetupElements string { "comma-separated-list-of-
     setup-elements" }
```

Description

Specifies the elements to be ignored during the comparison of `*.sdb` files.

The default is
`"plottingoptions,sortVariableValues,overwritehistoryname,uuid"`.

GUI Equivalent

None

Examples

```
envGetVal("verifier.snapshots" "ignoreImpSetupElements")
envSetVal("verifier.snapshots" "ignoreImpSetupElements" 'string
"sortVariableValues,overwritehistoryname")
```

Related Topics

Ignoring Elements or their Order when Comparing Implementations from Snapshots

© 2023 All Rights Reserved.

ignoreImpStateElements

```
verifier.snapshots ignoreImpStateElements string { "comma-separated-list-of-
     state-elements" }
```

Description

Specifies the elements to be ignored during the comparison of `*.state` files.

The default is `""`.

GUI Equivalent

None

Examples

```
envGetVal("verifier.snapshots" "ignoreImpStateElements")
envSetVal("verifier.snapshots" "ignoreImpStateElements" 'string "Locked,Status")
```

Related Topics

Ignoring Elements or their Order when Comparing Implementations from Snapshots

© 2023 All Rights Reserved.

impElementOrderSensitivity

```
verifier.snapshots impElementOrderSensitivity cyclic { "Insensitive" | "Sensitive"
     | "SameAsReportIdenticalhistory" }
```

Description

Controls the sensitivity of the order of differences in the XML files that are created during snapshot comparison.

- `SameAsReportIdenticalHistory` considers only those differences in order elements that are reported by identical histories in maestro cellviews.
- `Sensitive` considers all differences in element order during comparison.
- `Insensitive` ignores all differences in element order during comparison.

The default is `"SameAsReportIdenticalHistory"`.

GUI Equivalent

None

Examples

```
envGetVal("verifier.snapshots" "impElementOrderSensitivity")
envSetVal("verifier.snapshots" "impElementOrderSensitivity" 'cyclic "Sensitive")
envSetVal("verifier.snapshots" "impElementOrderSensitivity" 'cyclic
"Insensitive")
```

Related Topics

Ignoring Elements or their Order when Comparing Implementations from Snapshots

© 2023 All Rights Reserved.

includeImplementations

```
verifier.snapshots includeImplementations boolean { t | nil }
```

Description

Controls whether the design and simulation setup data from the maestro cellview must be saved within snapshots by including implementations.

The default is `t`.

GUI Equivalent

None

Examples

```
envGetVal("verifier.snapshots" "includeImplementations")
envSetVal("verifier.snapshots" "includeImplementations" 'boolean nil)
```

Related Topics

Adding or Removing a Snapshot

© 2023 All Rights Reserved.

toleranceProperties

```
verifier.snapshots toleranceProperties string { "comma-separated-list-of-
     properties" }
```

Description

Controls the properties to which the tolerance values are applied. The possible values are `Passed`,`Failed`,`Unmapped`,`NoResults`,`MinSpec`,`MaxSpec`,`MinValue`,`TypicalValue`, `MaxValue`,`OverallStatus`.

The defaults are `"MinValue,TypicalValue,MaxValue"`.

GUI Equivalent

Command	
Field	*Applying Tolerance Value*

Examples

```
envGetVal("verifier.snapshots" "toleranceProperties")
envSetVal("verifier.snapshots" "toleranceProperties" 'string
"MinValue,TypicalValue,MaxValue,")
```

Related Topics

Application of Tolerance Values in Snapshots

© 2023 All Rights Reserved.

Environment Variables to Manage Implementations

The following table lists the environment variables grouped under the `verifier.implementation` family.partition:

additionalProjectDirectory
autoCreateRanOkRequirement
autoSyncOutputFilter
customCell
customHistory
customLib
customView
includeOnlyMappableOutputsInCount
loadToActiveWhileViewAndReuseSession

© 2023 All Rights Reserved.

additionalProjectDirectory

```
verifier.implementation additionalProjectDirectory string { "comma-separated-
     paths of-project-directories" }
```

Description

Creates the additional project directories.

You can specify multiple paths as a comma-separated list of values. If a path contains special characters, such as white spaces, you do not need to escape the special characters by using the '\' character. For example, the path */path/with/white\ space/* can be specified as */path/with/white space/*.

To add a project directory, you must specify a non-empty project directory while ensuring that the directory path is readable and valid. For this, you can also use the UNIX environment variables ($) and home directory (~).

For paths including white spaces (such as "ab c"), you do not need to escape the character ' '.

The default is `""`.

GUI Equivalent

Command	
Field	*Project Directory Setup*

Examples

```
envGetVal("verifier.implementation" "additionalProjectDirectory")
envSetVal("verifier.implementation" "additionalProjectDirectory" 'string "$HOME/
simulation,simulation,./abc/simulation")
```

Related Topics

Project Directory Setup

© 2023 All Rights Reserved.

autoCreateRanOkRequirement

```
verifier.implementation autoCreateRanOkRequirement string { "comma-separated-
     list-of-implementation-elements" }
```

Description

Creates the *RanOk* requirements automatically with the specified elements.

The default is s are `"Implementation,Run,Reliability,Test".`

GUI Equivalent

None

Examples

```
envGetVal("verifier.implementation" "autoCreateRanOkRequirement")
envSetVal("verifier.implementation" "autoCreateRanOkRequirement" 'string
"Implementation,Run")
```

Related Topics

Creating Requirements from Implementations

© 2023 All Rights Reserved.

autoSyncOutputFilter

```
verifier.implementation autoSyncOutputFilter string { "regular-expression-
     format-string" }
```

Description

Filters the outputs in auto-synced implementations based on the specified criteria, which can be specified using regular expression format strings.

Some regular expression format strings and their results are as follows:

- `".*DCGain.*"` - Filters any output that contains `DCGain` in the name
- `"^DCGain.*"` - Filters any output whose name begins with `DCGain`
- `"C?DCGain"` - Filters outputs whose name is `CDCGain` or `DCGain`
- `"DCGain1|DCGain2"` - Filters the output whose name is `"DCGain1" or "DCGain2"`
- `"DCGain Bandwidth"` - Filters the output whose name is `"DCGain Bandwidth"`

The default is `""`.

GUI Equivalent

None

Examples

```
envGetVal("verifier.implementation" "autoSyncOutputFilter")
envSetVal("verifier.implementation" "autoSyncOutputFilter" 'string
"DCGain1|DCGain2")
envSetVal("verifier.implementation" "autoSyncOutputFilter" 'string "DCGain,
DCGain1")
```

Related Topics

Simulation Runs Using Auto-Synced Requirements

© 2023 All Rights Reserved.

customCell

```
verifier.implementation customCell string { "cell-name" }
```

Description

Defines the cell for a new custom implementation.

The default is `"C"`.

GUI Equivalent

None

Examples

```
envGetVal("verifier.implementation" "customCell")
envSetVal("verifier.implementation" "customCell" 'string "myCell")
```

Related Topics

Custom Implementations

© 2023 All Rights Reserved.

customHistory

```
verifier.implementation customHistory string { "history-name" }
```

Description

Defines the history for a new custom implementation.

The default is `"H"`.

GUI Equivalent

None

Examples

```
envGetVal("verifier.implementation" "customHistory")
envSetVal("verifier.implementation" "customHistory" 'string "Created1Jan2022")
```

Related Topics

Custom Implementations

© 2023 All Rights Reserved.

customLib

```
verifier.implementation customLib string { "library-name" }
```

Description

Defines the library for a new custom implementation.

The default is `"CustomL"`.

GUI Equivalent

None

Examples

```
envGetVal("verifier.implementation" "customLib")
envSetVal("verifier.implementation" "customLib" 'string "myCustomLib")
```

Related Topics

Custom Implementations

© 2023 All Rights Reserved.

customView

```
verifier.implementation customView string { "view-name" }
```

Description

Defines the view for a new custom implementation.

The default is `"V"`.

GUI Equivalent

None

Examples

```
envGetVal("verifier.implementation" "customView")
envSetVal("verifier.implementation" "customView" 'string "myNewView")
```

Related Topics

Custom Implementations

© 2023 All Rights Reserved.

includeOnlyMappableOutputsInCount

```
verifier.implementation includeOnlyMappableOutputsInCount boolean { t | nil }
```

Description

Ignores the tests, runs, reliability setups, and cellview, and only considers the implementation outputs for counting the mapped items or the mapping percentage.

The default is `nil`.

GUI Equivalent

None

Examples

```
envGetVal("verifier.implementation" "includeOnlyMappableOutputsInCount")
envSetVal("verifier.implementation" "includeOnlyMappableOutputsInCount" 'boolean
t)
```

Related Topics

MappedHier

© 2023 All Rights Reserved.

loadToActiveWhileViewAndReuseSession

```
verifier.implementation loadToActiveWhileViewAndReuseSession boolean { t | nil }
```

Description

Loads the history of the currently opened ADE Assembler cellview to the active setup and enables running simulations with the modified setup in the Local Run mode. The ADE Assembler cellview remains open during this custom simulation.

The default is `nil`.

GUI Equivalent

None

Examples

```
envGetVal("verifier.implementation" "loadToActiveWhileViewAndReuseSession")
envSetVal("verifier.implementation" "loadToActiveWhileViewAndReuseSession"
'boolean t)
```

Related Topics

Customizing Simulations with ADE Assembler Cellviews

© 2023 All Rights Reserved.

Environment Variables to Manage vManager Settings

The following table lists the environment variables grouped under the `verifier.vmgr` family.partition:

itemLimit
hostname
lookupHostnameTimeout
password
port
project
timeout
username
vPlan

© 2023 All Rights Reserved.

itemLimit

`verifier.vmgr itemLimit int` *`numeric-value`*

Description

Controls the number of requirements that Verifier can load as one block from vManager during a query to retrieve requirements. If the number of requirements to be downloaded is higher that this number, the download request is split into multiple requests. The acceptable value is a number between 100 and 100000.

The default is `1000`.

GUI Equivalent

None

Examples

```
envGetVal("verifier.vmgr" "itemLimit")
envSetVal("verifier.vmgr" "itemLimit" 'int 2000)
```

Related Topics

Setting Up the vManager Connection

© 2023 All Rights Reserved.

hostname

```
verifier.vmgr hostname string { "server-host-name" }
```

Description

Specifies the hostname of the vManager server.

The default is `nil`.

GUI Equivalent

Command	*Tools – vManager – Setup vManager*
Field	*Hostname*

Examples

```
envGetVal("verifier.vmgr" "hostname")
envSetVal("verifier.vmgr" "hostname" 'string "myVManagerServer")
```

Related Topics

vManager Setup Form

© 2023 All Rights Reserved.

lookupHostnameTimeout

```
verifier.vmgr lookupHostnameTimeout int numeric-value
```

Description

Controls the maximum time after which lookup for the specified host name of the vManager server aborts. When set to `0`, skips the lookup for the host name and requires you to specify the correct host name. The acceptable value is a number between 0 and 240.

The default is `5`.

GUI Equivalent

Command	*Tools – vManager – Setup vManager*
Field	*Hostname*

Examples

```
envGetVal("verifier.vmgr" "lookupHostnameTimeout")
envSetVal("verifier.vmgr" "lookupHostnameTimeout" 'int 10)
```

Related Topics

vManager Setup Form

© 2023 All Rights Reserved.

password

```
verifier.vmgr password string { "password-string" }
```

Description

Specifies the password for logging onto the vManager server.

The default is `""`.

GUI Equivalent

Command	*Tools – vManager – Setup vManager*
Field	*Password/Token*

Examples

```
envGetVal("verifier.vmgr" "password")
envSetVal("verifier.vmgr" "password" 'string "abcd")
```

Related Topics

vManager Setup Form

© 2023 All Rights Reserved.

port

`verifier.vmgr port int` *`numeric-value`*

Description

Specifies the port settings of the vManager server. The acceptable value is a number between `0` and `100000`.

The default is `0`.

GUI Equivalent

Command	*Tools – vManager – Setup vManager*
Field	*Port*

Examples

```
envGetVal("verifier.vmgr" "port")
envSetVal("verifier.vmgr" "port" 'int 5000)
```

Related Topics

vManager Setup Form

© 2023 All Rights Reserved.

project

```
verifier.vmgr project string { "project-name" }
```

Description

Specifies the name of the project to use on the vManager server.

The default is `""`.

GUI Equivalent

Command	*Tools – vManager – Setup vManager*
Field	*Project*

Examples

```
envGetVal("verifier.vmgr" "project")
envSetVal("verifier.vmgr" "project" 'string "myProjectName")
```

Related Topics

vManager Setup Form

© 2023 All Rights Reserved.

timeout

`verifier.vmgr timeout int` *`numeric-value`*

Description

Controls the time after which the vManager connection is terminated if the vManager server is unresponsive. This time should be set to a duration long enough where the vManager server is highly loaded and responses are slow. The acceptable value is a number between `0` and `100000`.

The default is `120`.

GUI Equivalent

Command	*Tools – vManager – Setup vManager*
Field	*Timeout*

Examples

```
envGetVal("verifier.vmgr" "timeout")
envSetVal("verifier.vmgr" "timeout" 'int 240)
```

Related Topics

vManager Setup Form

© 2023 All Rights Reserved.

username

```
verifier.vmgr username string { "user-name" }
```

Description

Specifies the username for logging onto the vManager server.

The default is `""`.

GUI Equivalent

Command	*Tools – vManager – Setup vManager*
Field	*User*

Examples

```
envGetVal("verifier.vmgr" "username")
envSetVal("verifier.vmgr" "username" 'string "user1")
```

Related Topics

vManager Setup Form

© 2023 All Rights Reserved.

vPlan

```
verifier.vmgr vPlan string { "plan-name" }
```

Description

Specifies the vPlan file associated with the Verifier cellview.

The default is `""`.

GUI Equivalent

Command	*Tools – vManager – Setup vManager*
Field	*vPlan*

Examples

```
envGetVal("verifier.vmgr" "vPlan")
envSetVal("verifier.vmgr" "vPlan" 'string "myPlan")
```

Related Topics

vManager Setup Form

© 2023 All Rights Reserved.

C

Guidelines for Debugging in Virtuoso ADE Verifier

Virtuoso ADE Verifier (Verifier) is a comprehensive application to perform requirements-driven verification of analog designs. This topic helps you with information to debug Verifier when you observe incorrect tool behavior.

As a Verifier user, you might experience issues or unexpected behavior in the tool. This document helps you to investigate the root cause of such problem areas. It could lead you to resolve the problems yourself, or you might need to contact Cadence support for further analysis.

Important

Use the debugging features of Verifier only while investigating the problem and turn them off afterwards. The features might have a negative impact on the performance of the tool and / or might influence the behavior.

Verifier provides many options to debug for incorrect results. However, there is an extent to which you can debug the tool. In complex cases, contact a Cadence representative.

When you contact Cadence support, apply these generic guidelines to help Cadence representative find and resolve the problem:

- Describe the problem as precisely as possible to the Cadence representative.
- List the steps to reproduce the issue in detail.
- Check if the issue is specific to a testcase.
- Make a note of the ISR stream that has the problem areas. Verify other ISRs as well if a similar issue exists.
- Collect log files, snapshots, or videos of the issue.
- Provide a testcase, if possible.

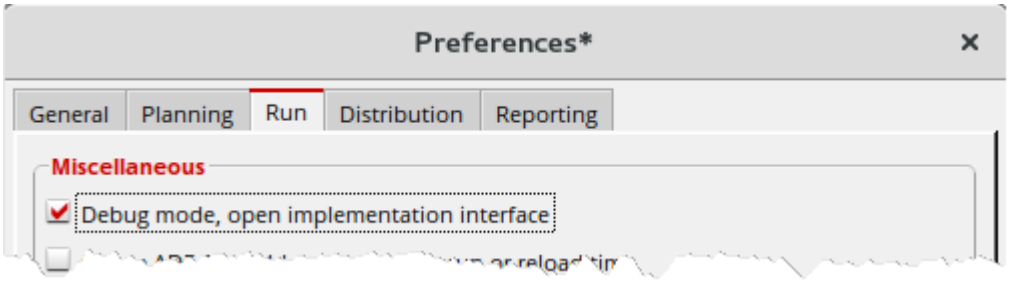
© 2023 All Rights Reserved.

Analysis of the Log File

The important messages that need immediate attention are displayed in CIW.

Use of Debug Mode

The issues related to ADE Assembler might not be visible because they are running in the background when working in ADE Verifier. For debugging such issues, you can run implementations in debug mode in a user interface, where the implementation interface, such as ADE Assembler, gets displayed to monitor the runs. Select the *Debug mode, open implementation interface* option in the *Run* tab of the Preferences form.



Selecting this option keeps the ADE Assembler UI open for you to debug. You can close it manually.

Analysis of Implementation Run Summary Data

To debug the issues related to an implementation run, check the location of the implementation run summary data using the `palswhere` environment variable. Ensure that you are able to access the data at the given location. The data exists in the form of XML files at the following location.



© 2023 All Rights Reserved.