



Engineering What's Ahead.

©2022 ANSYS, Inc.

All Rights Reserved.

Unauthorized use, distribution
or duplication is prohibited.

RedHawk-SC™ User Manual



Ansystech, Inc.
Southpointe
2600 ANSYS Drive
Canonsburg, PA 15317
ansysinfo@ansys.com
(T) 724-746-3304
(F) 724-514-9494

Release 2022 R1
May 2022

ANSYS, Inc. is certified to ISO
9001:2015

Contents

1: Introduction to Ansys RedHawk-SC™ and SeaScape™ Framework.....	11
1.1. Using RedHawk-SC in the Design Flow.....	12
1.2. The SeaScape™ Framework.....	12
1.2.1. Features and Benefits.....	13
1.2.2. SeaScape Master and Workers.....	14
1.2.3. Scheduling and Executing Jobs.....	15
1.2.4. Saving Data to Disk.....	16
2: User Interface and Input Data Preparation.....	17
2.1. User Interface.....	17
2.1.1. Overview of the SeaScape Database	17
2.1.2. Methodology Overview.....	19
2.1.2.1. Main Views.....	19
2.1.2.2. Supplementary Views.....	21
2.1.3. Setting the RedHawk-SC Environment.....	21
2.1.4. Launching Workers.....	22
2.1.5. Working With Python Scripts.....	22
2.1.6. Launching RedHawk-SC.....	25
2.1.7. Exploring Log Files.....	26
2.1.8. Viewing Command Help.....	26
2.1.9. Launching the RedHawk-SC Console.....	27
2.1.10. Launching the Scheduler GUI.....	28
2.2. Preparing Input Data.....	29
2.2.1. Specifying Input Data.....	29
2.2.1.1. Technology Data.....	29
2.2.1.2. Design Data.....	29
2.2.1.3. Library Data.....	30
2.2.1.4. Macro Data.....	32
2.2.1.5. Input Data Specification Example.....	32
2.2.2. Creating Base Views.....	32
2.2.3. RedHawk-SC Data Integrity Checks.....	33
2.2.4. Checking Data Integrity.....	35
3: RedHawk-SC Views.....	37
3.1. SeaScape Views.....	37
3.2. DesignView.....	39
3.3. ModifiedDesignView.....	41
3.4. ExtractView.....	42
3.5. TimingView.....	43
3.6. ScenarioView.....	44
3.7. AnalysisView.....	46
3.8. AnalysisComboView.....	47

3.9. ElectromigrationView.....	47
3.10. ThermalView.....	48
3.11. LibertyView.....	49
3.12. MacroView.....	52
3.13. TechView.....	52
3.14. ValueChangeView.....	53
3.15. SwitchingActivityView.....	54
3.16. PowerView.....	55
3.17. SimulationView.....	56
4: Early Grid Analysis.....	59
4.1. Reporting Shorts.....	59
4.1.1. Reporting Shorts Summary.....	60
4.1.2. Reporting Shorts Details.....	60
4.1.3. Querying Custom Shorts Data.....	61
4.1.4. Viewing Shorts.....	61
4.2. Reporting Disconnected Instances and Wires.....	62
4.2.1. Reporting Disconnected Pins.....	63
4.2.2. Querying Disconnected Instances.....	64
4.2.3. Viewing Disconnects in GUI.....	64
4.3. Performing Build Quality Metric (BQM) Analysis.....	65
4.3.1. Generating Probes and Currents.....	67
4.3.2. Generating Instance Voltage Drop Heatmaps.....	68
4.3.3. Viewing BQM Analysis Results.....	68
4.3.4. Performing BQM Analysis for Macro Cells.....	72
4.4. Tracing Shortest Path Resistance (SPR).....	74
4.4.1. Generating Shortest Path Resistance Data.....	74
4.4.2. Querying SPR of Instance Pins.....	75
4.4.3. Querying Pin SPR Path Data.....	76
4.4.4. Viewing Pin SPR Values.....	77
4.4.5. Querying SPR From a Location.....	80
4.4.6. Viewing SPR Path For a Location.....	81
4.4.7. Generating SPR Text Report.....	81
4.5. Computing Effective Resistance.....	84
4.5.1. Computing Effective Resistance for Macro Cells.....	86
4.5.2. Generating Effective Resistance Text Report.....	86
4.5.3. Querying Effective Resistance of Instance Pins and Probes.....	87
4.5.4. Querying Phantom Layers and Locations.....	88
4.5.5. Viewing Effective Resistance Heatmaps.....	89
4.6. Reporting Missing Vias.....	90
4.6.1. Generating Missing Via Locations Text Report.....	91
4.6.2. Viewing Missing Vias.....	91
4.7. Creating Statistical Heatmaps for Multiple Parameters.....	92
4.7.1. Computing Resistance Gradient from BQM Data.....	93
4.7.1.1. Viewing Resistance Gradient Heatmaps.....	94

4.7.2. Analyzing PeakTW Currents.....	94
4.7.2.1. Creating PeakTW Heatmaps.....	96
4.7.2.2. Viewing PeakTW Heatmaps.....	97
4.7.3. Analyzing Slack.....	97
4.7.3.1. Viewing Slack Gradient Heatmaps.....	98
4.7.4. Combining Heatmaps.....	99
4.7.4.1. Viewing Consolidated Heatmaps.....	99
5: Computing Power, Static IR Drop and EM Analysis.....	103
5.1. Static Analysis Methodology.....	103
5.2. Static Analysis Flow.....	104
5.3. Setting Switching Activity.....	105
5.3.1. Defining Activity.....	105
5.3.2. Loading SAIF Files.....	108
5.3.3. Loading VCD or FSDB Files.....	109
5.3.4. Using Vectorless and Vector-Based Methods Together.....	111
5.3.5. Reporting Switching Activity Values.....	112
5.4. Computing Power.....	112
5.4.1. Determining Power Using Switching Activity.....	113
5.4.2. Determining Power Values from IPF.....	114
5.4.3. Creating PowerView from Existing ScenarioView.....	115
5.4.4. Determining Power of Macro Cells.....	115
5.4.5. Querying PowerView.....	118
5.4.6. Reporting Power Calculation.....	119
5.5. Scaling Power.....	120
5.5.1. Reporting Scaled Power.....	125
5.6. Performing Static Analysis.....	125
5.7. Reporting Static Analysis Results.....	126
5.8. Viewing Static Analysis Results in GUI.....	129
5.9. Generating Histograms.....	136
5.10. Static Electromigration Analysis.....	138
5.10.1. Performing DC Electromigration Analysis.....	139
5.10.2. Reporting Power Electromigration Results.....	139
5.10.3. Viewing Electromigration Results in GUI.....	140
6: Vectorless Dynamic Analysis.....	143
6.1. DVD Analysis Methodology.....	143
6.2. DVD Analysis Flow.....	144
6.3. Creating No-Propagation Vectorless (NPV) Scenarios.....	145
6.3.1. Inputting object_settings Syntax as JSON Files.....	148
6.3.2. Specifying Frame Length.....	149
6.3.3. Specifying Target Power.....	150
6.3.4. Specifying Probability of Initial Events.....	151
6.3.5. Specifying Toggle Rates.....	152
6.3.5.1. Reading Instance Toggle Rate Files.....	153
6.3.5.2. Controlling Clock Pins Switching.....	154

6.3.5.3. Configuring MBFF Switching.....	154
6.3.5.4. Maximizing Switching Coverage.....	155
6.3.6. Loading PowerView.....	155
6.3.6.1. Matching Power by Cell Type.....	156
6.3.6.2. Matching Macro Power.....	157
6.3.7. Specifying Macro Modes.....	158
6.3.8. Defining Default Clock Policy.....	158
6.3.9. Defining Arrival Time Policy.....	158
6.3.10. Specifying Switching Sequence for Standard Cell Instances.....	159
6.3.11. Using Multiple Input Types Together.....	161
6.4. Creating Logic Propagation Scenarios.....	163
6.4.1. Propagating Clock Events.....	164
6.4.2. Propagating Data Events.....	165
6.4.2.1. Initiating Activity at Primary Input Ports.....	166
6.4.2.2. Specifying Activity at Start Points.....	166
6.4.2.3. Using PowerView to Specify Activity at Start Points.....	167
6.4.2.4. Toggle Rate Distribution at Start Points.....	168
6.4.2.5. Propagating Events through Combinational Network.....	168
6.4.3. Determining Event Arrival Time.....	169
6.4.3.1. Specifying Arrival Time Precedence.....	170
6.4.4. Determining Transition Time.....	170
6.4.5. Inputting Current Waveform Text File for Instance.....	172
6.4.6. Controlling Events per Signal With LSO.....	173
6.5. Power Constrained Vectorless Scenario.....	174
6.5.1. Features.....	175
6.5.2. PCVS Flow.....	175
6.5.3. PCVS Inputs.....	178
6.5.4. PCVS Outputs.....	182
6.6. Vectorless Scan Flow.....	190
6.6.1. Scan Constraint Files.....	190
6.6.2. Flow Setup in RedHawk-SC.....	192
6.6.3. Scan Shift Operation.....	193
6.7. Controlling Macro Switching in RedHawk-SC.....	196
6.7.1. Input Settings.....	196
6.7.2. Reviewing Results.....	202
6.8. Event Replay Flow.....	207
6.8.1. Methodology.....	208
6.8.2. Examples.....	209
6.9. Performing Dynamic Voltage Drop Analysis.....	210
6.9.1. Specifying Level of Statistics.....	211
6.9.2. DVD Reporting Parameters.....	211
6.9.2.1. Timing Window Based Parameters.....	211
6.9.2.2. Effective DVD Window Based Parameters.....	213
6.10. Reporting Dynamic Analysis Results.....	215

6.11. Viewing Dynamic Analysis Results in GUI.....	219
6.12. Querying Analysis Data.....	223
6.13. Querying Instance Data.....	224
6.14. Generating DVD Histograms.....	228
6.15. Recommended Flow.....	228
6.16. Dynamic PG Electromigration Analysis.....	229
6.16.1. Performing RMS and Peak Electromigration Analysis.....	229
6.16.2. Reporting RMS and Peak Electromigration Results.....	230
6.16.3. Viewing DVD Electromigration Results in GUI.....	231
6.17. Handling Decoupling Capacitance.....	231
6.17.1. Stamping Decoupling Capacitance.....	232
6.17.2. Querying Decoupling Capacitance Data.....	234
7: Dynamic Analysis With Vectors.....	237
7.1. Switching Scenario.....	237
7.2. Processing VCD Files.....	238
7.3. ValueChangeView.....	238
7.3.1. Need and Concept.....	239
7.3.2. Name Mapping.....	240
7.3.2.1. Name Mapping File.....	242
7.3.2.2. Examples of Name Mapping File.....	243
7.3.2.3. Name Mapping Rules.....	246
7.3.2.4. MBFF Specific Settings.....	247
7.3.3. Concept of Time Slices.....	248
7.3.4. X State Handling.....	248
7.3.5. Parsing Long Duration FSDB.....	249
7.4. Vector-Based Dynamic Analysis.....	249
7.4.1. Logic Propagation and No Propagation Scenarios.....	250
7.4.2. Types of VCD Files.....	250
7.4.3. VCD Settings in ScenarioView.....	254
7.4.4. Filtering Glitches from VCD in NPV ScenarioView.....	258
7.4.5. Enabling Macro Non-Clock Active Pins to Generate Events.....	258
7.5. Checking VCD Annotation.....	259
7.5.1. VCV Annotation Reporting.....	259
7.5.2. VCD Details of an Instance or Net.....	260
8: Signal Electromigration Analysis.....	263
8.1. Signal Electromigration Analysis Methodology.....	263
8.2. Signal Electromigration Flow.....	264
8.3. Creating Timing View.....	266
8.4. Creating SwitchingActivityView.....	266
8.5. Creating ExtractView.....	266
8.6. Creating Simulation View.....	267
8.7. Creating SignalNetCurrentView.....	267
8.7.1. Handling Ports and Pins.....	267
8.7.2. Including and Excluding Nets.....	268

8.7.3. Analyzing Constant Signal Nets for Electromigration.....	269
8.7.4. Analyzing Clock Mesh Drivers.....	269
8.7.5. Customizing Signal Net Parameters.....	270
8.7.6. Specifying Current Distribution Fractions for Signal Pin.....	272
8.8. Creating Electromigration View.....	272
8.9. Viewing Signal Electromigration Results in GUI.....	273
8.10. Reporting Signal Electromigration Results.....	279
9: Rampup Analysis for Power-Gated Designs.....	291
9.1. Power Gating.....	291
9.2. Input Requirements.....	294
9.2.1. LibertyView.....	294
9.2.2. ScenarioView.....	296
9.3. Rampup Analysis Flow in RedHawk-SC.....	297
9.4. Analyzing Rampup Impact Using GUI Heatmaps.....	302
9.5. Scripts for Rampup and Absolute Turn-On Time Heatmaps.....	302
9.6. Scripts for Rampup Reporting.....	304
9.7. Evaluating Results.....	305
10: Chip Power Modeling.....	309
10.1. Overview.....	309
10.2. Creating CPM Using Single Worker.....	311
10.2.1. Prerequisites to Generate GrmView With Single Worker.....	312
10.2.2. Generating GrmView With Single Worker.....	313
10.2.3. Creating CPM Ports.....	317
10.2.4. CPM Script Example.....	320
10.3. Creating CPM Using Multiple Workers.....	321
10.4. Storing CPM Output.....	322
10.5. Validating CPM	327
11: Multichip Analysis.....	329
11.1. Multichip Concepts.....	329
11.2. Multichip Flow Overview.....	331
11.3. Design Data Requirements.....	332
11.4. Creating Multichip Configuration File Using Setup Utility.....	334
11.4.1. Invoke Multi-Die Setup.....	336
11.4.2. Setup Path and Work Directory.....	336
11.4.3. Create New Project.....	337
11.4.4. Add Dies.....	338
11.4.5. Input Design Data.....	339
11.4.6. Add Off-Chip Model Prototype.....	345
11.4.7. Generate the Config File.....	345
11.5. Creating Command File for Multichip Analysis.....	351
11.6. Multichip Analysis Flows.....	354
11.6.1. Full Detailed Flow.....	355
11.6.2. On-the-Fly Model Based Flow.....	356
11.6.3. Model Based Approach.....	358

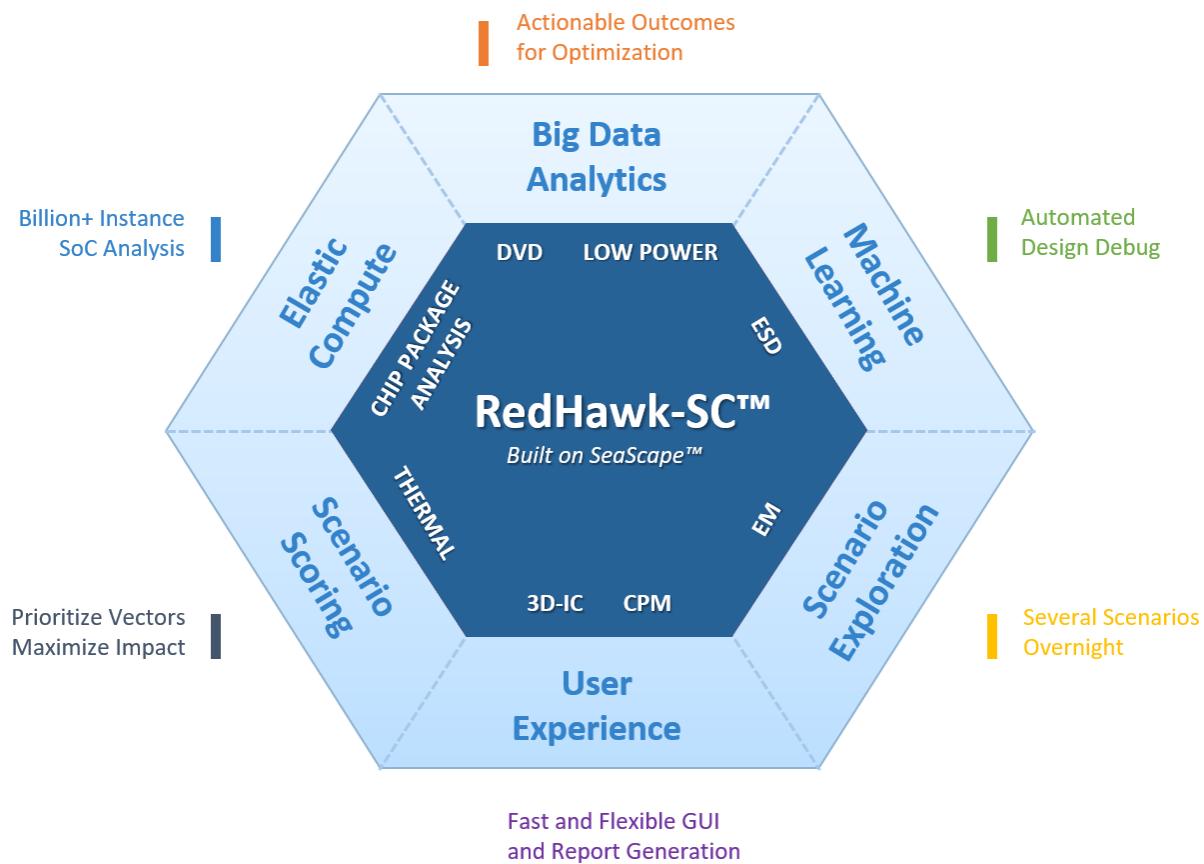
11.6.4. Interposer PDN Quality Checks (Interposer Only Flow).....	360
11.7. Multichip-Specific GUI Features.....	364
11.8. Reporting Multichip Analysis Results.....	366
11.9. SPR for Multichip Designs.....	369
11.9.1. Viewing Multichip SPR Data in GUI.....	370
12: Package and Board Analysis in RedHawk-SC.....	375
12.1. RLC Model.....	375
12.2. RLCK SPICE Netlist.....	377
12.3. S-Parameter Model.....	380
12.3.1. Specifying S-Parameter Models in RedHawk-SC.....	382
12.3.2. Including S-Parameters in RedHawk-SC Analysis.....	383
12.4. Important Functions of Package Module.....	383
13: SelfHeat Analysis.....	391
13.1. Sources of SelfHeat.....	392
13.2. SelfHeat Flows.....	393
13.3. SelfHeat Analysis Overview	395
13.3.1. Input Data Preparation.....	396
13.3.2. Setting Up the Environment.....	399
13.3.3. Creating Thermal View.....	400
13.3.4. Post Thermal Electromigration Analysis.....	401
13.4. SelfHeat Reports.....	403
13.4.1. Reporting Instance Temperature	404
13.4.2. Reporting Edge Temperature.....	405
13.5. Reviewing Results in GUI.....	407
14: Power Profiling Across Long Vectors.....	411
14.1. Power Profile Inputs.....	411
14.2. Power Profile Outputs.....	416
14.3. Utilities for Vector Window Selection.....	424
14.4. Integrated Command for Vector Selection.....	425
15: Using the RedHawk-SC Infrastructure.....	431
15.1. Managing Workload.....	431
15.1.1. Creating and Launching Workers.....	431
15.2. RedHawk-SC Scheduler GUI.....	436
15.2.1. Overview.....	436
15.2.2. Invoking the Scheduler GUI.....	441
15.2.3. Features of the Scheduler GUI.....	442
15.2.4. Debugging With Scheduler.....	448
15.3. Managing Disk Space.....	451
15.3.1. Overview.....	452
15.3.2. Methods of Disk Storage.....	453
15.3.3. Merging Distributed Databases.....	457
15.3.4. Profiling Databases.....	458
15.3.5. Controlling Memory Consumption Versus Runtime.....	459
15.4. Resiliency.....	459

16:	RedHawk-SC GUI.....	463
16.1.	Launching the RedHawk-SC Console.....	463
16.2.	Opening Databases.....	465
16.3.	Viewing the Layout Window.....	468
16.3.1.	Viewing Selector Panels.....	473
16.3.2.	Viewing Analysis Results.....	486
16.4.	Managing Workers.....	487
16.5.	Using the Script Creation Wizard.....	489
16.6.	Using RedHawk-SC Help.....	491
17.1.1.	Copyright and Trademark Information.....	497

1: Introduction to Ansys RedHawk-SC and SeaScape Framework

Ansys RedHawk-SC is the next-generation SoC power-noise and reliability framework to enable advanced design analysis. RedHawk-SC is built on Ansys SeaScape™, the world's first purpose-built, big data architecture for multiphysics simulations. SeaScape framework provides per-core scalability, flexible design data access, instantaneous design bring-up, data analytics and many other revolutionary capabilities. RedHawk-SC is the industry gold-standard for SoC power noise and reliability sign-off.

The following figure shows the new features that RedHawk-SC enables over the classic RedHawk tool.



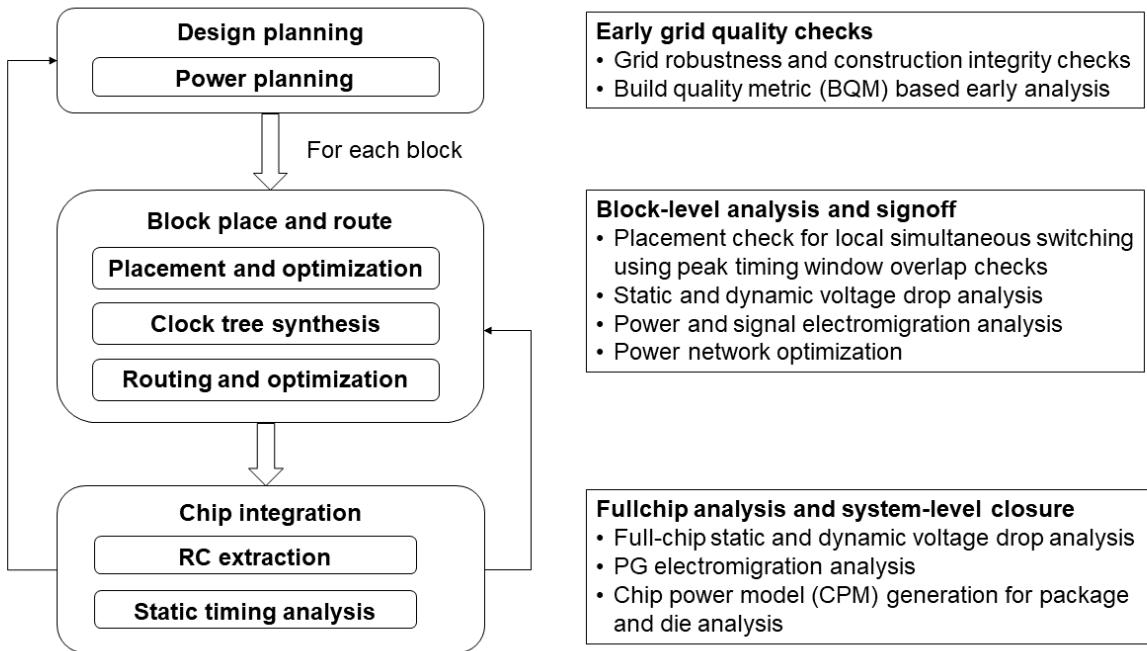
Some key analyses available in RedHawk-SC are as follows:

- Advanced power analytics
- Static and DVD diagnostics
- Chip-Package coanalysis
- Thermal analysis
- 3DIC analysis
- Chip Power Model generation
- Power and Signal Electromigration analysis
- Electrostatic Discharge analysis

1.1. Using RedHawk-SC in the Design Flow

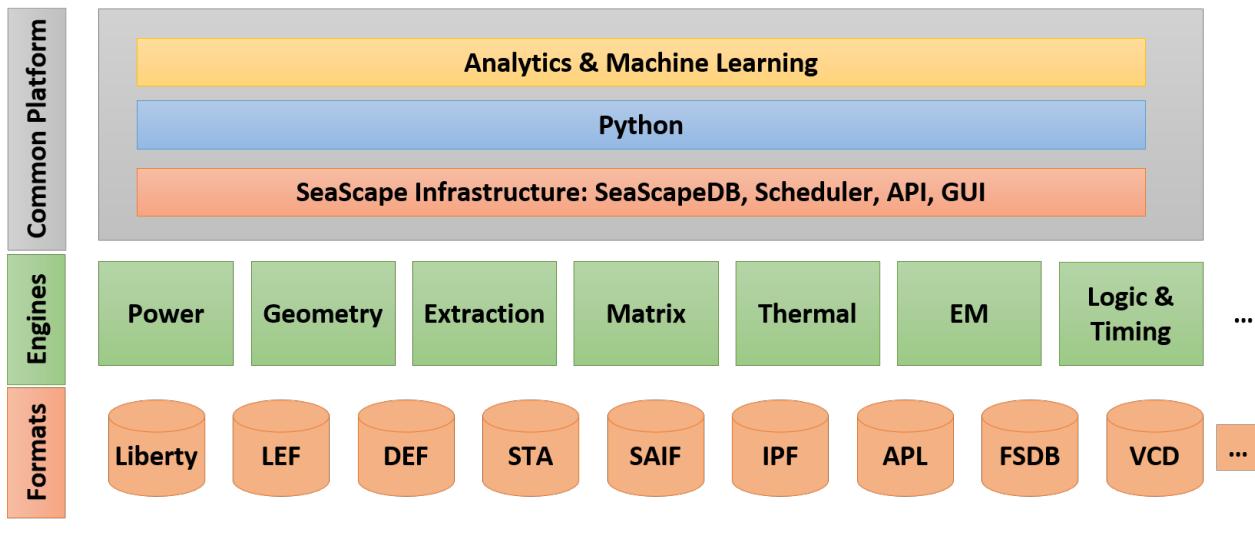
The RedHawk-SC tool supports power integrity checks and analysis at different stages of the design flow.

Figure 1. Using RedHawk-SC in the Design Flow



1.2. The SeaScape™ Framework

SeaScape™ is an EDA-specific distributed and scalable framework for actionable analytics. The RedHawk-SC tool consumes various input data formats and processes the data using the required engines for power integrity signoff. The SeaScape infrastructure includes the SeaScape database, scheduler, Python APIs, and a GUI. You can access these resources by using simple Python scripts and analyze your designs for power integrity.

Figure 2. SeaScape Overview

1.2.1. Features and Benefits

The RedHawk-SC tool uses the SeaScape framework to split power analysis into multiple jobs that are executed on different hosts in an execution farm, such as Univa Grid Engine or IBM Spectrum LSF. Using the SeaScape framework renders the following features and benefits to the RedHawk-SC tool:

- Elastic compute scalability and multi-site collaboration

With scalability across multiple cores using big data techniques, the RedHawk-SC tool helps you to sign off designs with more than a billion instances in a few hours. The tool can run large designs using low memory cores even if these reside on different machines. No dedicated machines are needed.

The tool can start working as soon as a single core is available and proportionately speeds up as more cores become available. Because the tool can employ unused cores, it increases usage rates of compute farms and reduces overall hardware costs.

Large designs require large compute resources, increasing the risk of CPU failure during analysis. The tool has the resiliency to automatically recover from network failure and unresponsive cores or machines.

Large design work flows involve integrating several blocks at different stages of completion through the design life cycle. The tool has adaptive partitioning for varying levels of design details (black box to full chip) with optimal run time.

Users across multiple sites can simultaneously view, debug, and explore design and simulation results.

- Big data analytics

Using custom data analytics, you can query, identify, and prioritize key design issues in a small time. The tool offers combined display and analytics across multiple views, heat maps for design quality check analysis, combined analysis across multiple scenarios for coverage analysis and issue diagnostics, and custom heat-map support.

- Machine learning support

The tool supports machine learning by collecting and analyzing data from different designs from current and prior simulations and designs. This enables identifying missed design weaknesses and automating time-consuming manual procedures.

- Fast design closure

With increasing number of cores in CPU and graphics processing unit (GPU) sub-systems in SoCs, you must isolate switching combinations that generate chip-package-PCB resonance conditions causing voltage-drop induced failure. The tool enables you to rapidly evaluate several switching possibilities and highlights the operating modes to be avoided.

You can also score vectors across multiple parameters, identify appropriate vectors, and isolate cycles in these vectors for power-noise signoff. This enables you to perform targeted simulations while meeting coverage expectations.

The tool can automatically time-slice large vectors, and simulate portions of the vector in parallel. This enables you to analyze large workloads on chips in a small time span.

- Multiphysics optimization

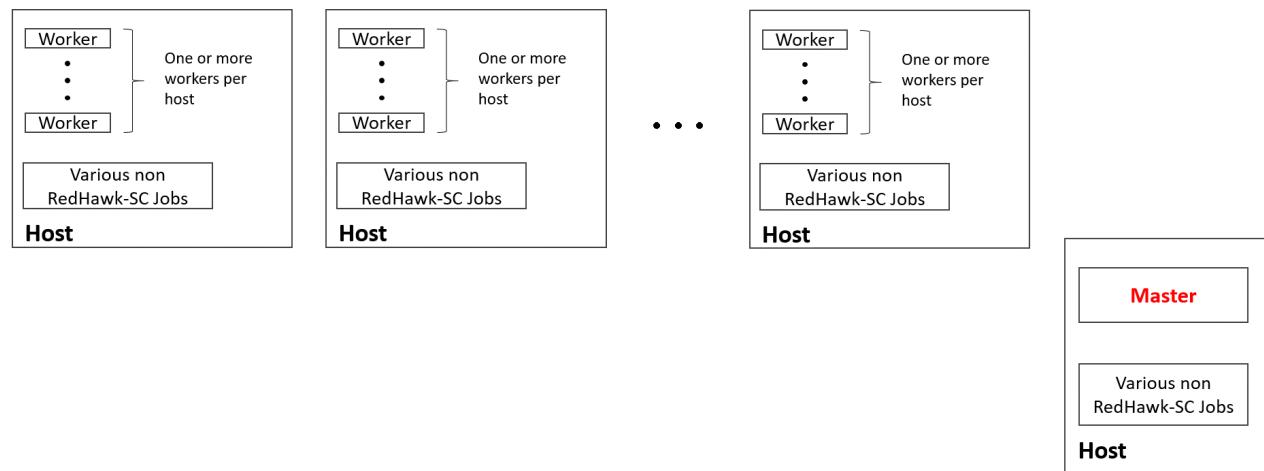
In advanced process nodes, margin-based design methodologies force overdesign and guard banding resulting in large die sizes and long design schedules. With the RedHawk-SC tool, you can simulate several scenarios and run multiphysics analytics in your design ECO cycle to target fixes in high stress areas, therefore, reducing overdesign in other parts of the chip. The design fixes can be made conveniently through standard interfaces using existing place and route solutions.

- Simultaneous analysis and instantaneous results

The tool allows simultaneous power and signal-line analysis on a full-chip SoC with full hierarchical support. You can view and monitor progress of the analysis in parallel with ongoing simulation. You can also debug results with current heat maps that display nodes to identify the extracted circuit for advanced current flow and electromigration modeling.

1.2.2. SeaScape Master and Workers

Figure 3. Overview of the SeaScape Master and Workers



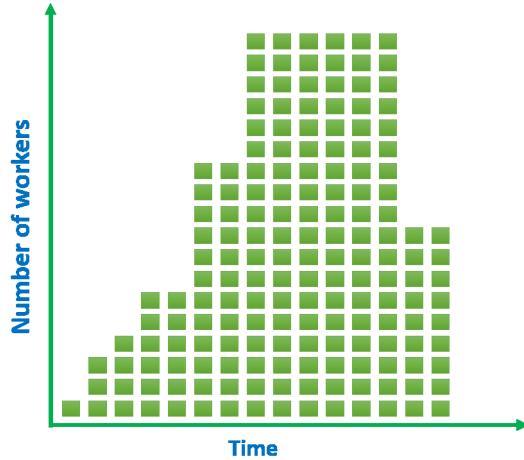
The master process dispatches and monitors jobs. The master process provides you with a console for quick interactive queries and, in general, does not run any jobs. Depending on demand, the master launches one or more worker processes using specified commands.

A worker process is a job execution daemon that takes up one CPU core memory unit. Typically, workers are launched in compute clusters, such as Univa Grid Engine, IBM Spectrum LSF, and Altair Accelerator (previously known as NetworkComputer). To optimize the number of workers required for a job, you should configure the RedHawk-SC tool to automatically launch workers on demand. The tool also enables you to manually launch workers. There is no distinction between the workers launched in either way.

Each worker process communicates to another worker process and with the master process through TCP/IP Sockets. A worker communicates the status of the job that it is executing together with its health status to the master. Further, the master process regularly pings the worker processes for their health status.

By default, processing begins as soon as a single worker process is ready instead of waiting for the availability of all the required workers as shown in the following figure. You can also specify the number of available workers for processing to begin.

Figure 4. Elastic Compute Environment

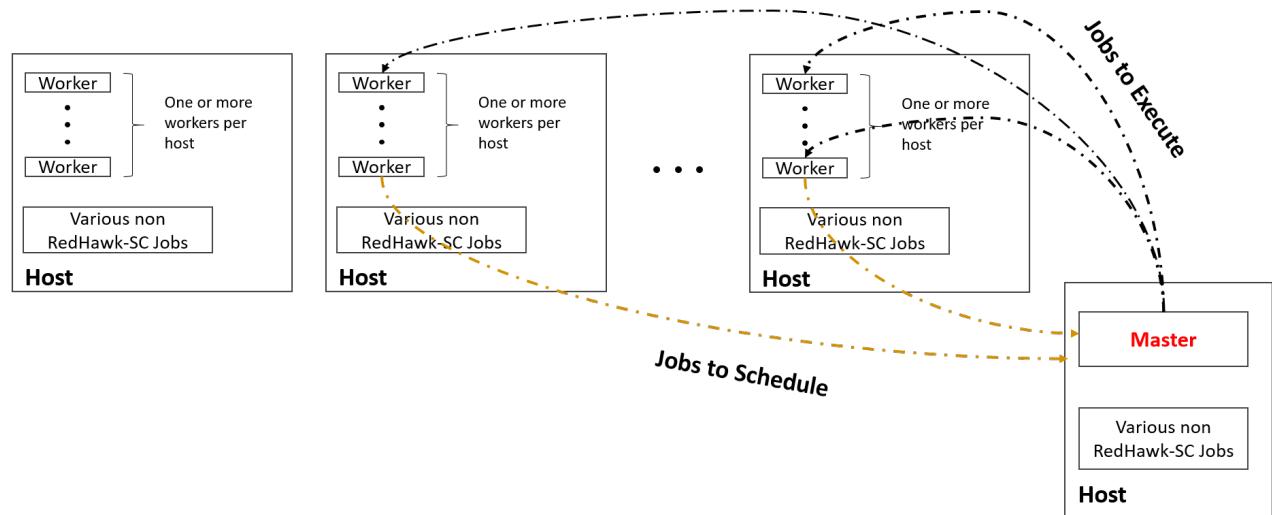


When any one of the workers die, the tool automatically kills all the other workers and terminates the analysis. This frees up the execution farm space from jobs without producible results. If the power analysis terminates prematurely, for example, due to bad cores or network issues, the tool tries to complete the analysis by invoking its resiliency feature.

1.2.3. Scheduling and Executing Jobs

The RedHawk-SC uses Python as the interface language.

Figure 5. Scheduling and Executing Jobs



For an input file with commands to run, the tool chunks each command into smaller jobs that are scheduled by the master for execution by the workers.

A job can have conditions or dependencies that might be satisfied only when other jobs complete execution. A job can create other jobs with their individual conditions. Scheduling the jobs enable seamless execution without holding up the master.

The RedHawk-SC Scheduler GUI enables you to view and track the status and execution details of a job. See [Launching the Scheduler GUI](#) on page 28.

1.2.4. Saving Data to Disk

The RedHawk-SC tool stores data processed from external inputs, such as LEF and DEF files, processing data, and output data in SeaScape databases (DBs).

The tool can read data from and write data into multiple SeaScape DBs in a single session. Also, multiple RedHawk-SC sessions can simultaneously access a SeaScape DB.

Typically, the SeaScape DB is a Linux directory that contains multiple views based on the input data types and each view holds related data. The data is partitioned into chunks for efficient processing.

The data is initially stored in each worker's memory and later written to a disk in the background. The data generated from power analysis, such as heatmaps, are saved to the disk as asynchronous jobs. The save operation does not hamper the progress of the analysis.

The following storage options are available:

- A shared disk location is accessible from all the worker host machines. A central disk is a single shared disk accessible by all workers.
- A distributed disk location is a local disk space in the worker host machine that is not accessible by workers running in other host machines.

For detailed information about these storage options, see [Methods of Disk Storage](#) on page 453.

2: User Interface and Input Data Preparation

This chapter describes the RedHawk-SC user interface and input data preparation methodology.

2.1. User Interface

The following topics describe the user interfaces to work in the RedHawk-SC environment.

2.1.1. Overview of the SeaScape Database

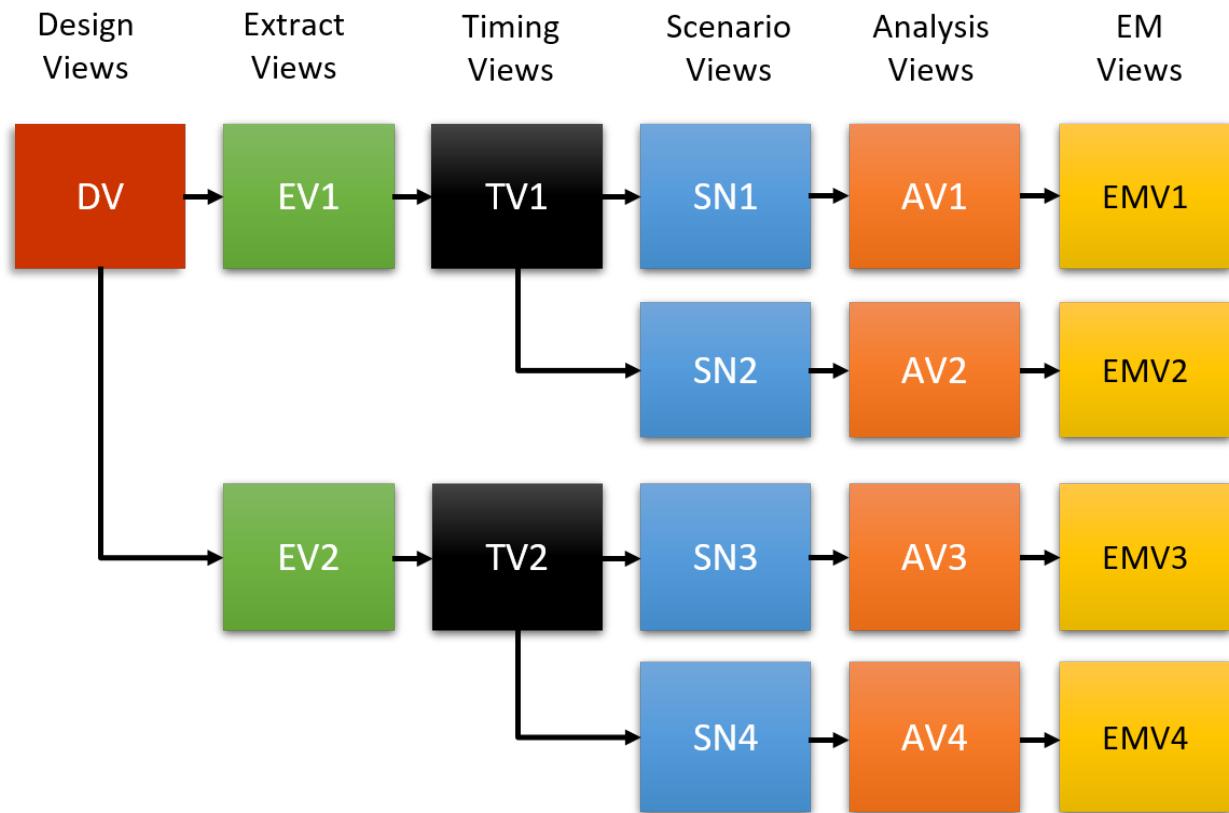
A SeaScape DB is a Linux directory containing multiple subdirectories called views. A view is a representation of a set of related data based on the input data type. Each view stores data from different stages of power integrity analysis.

Figure 6. Views in a SeaScape Database



SeaScape views have the following properties:

- Once a view is saved, it can be loaded multiple times.
- When you save a view, another version of the view with the same name is generated. For example, if you save the view, tv, it is automatically saved with the name, tv#v1. #v1 represents another version of the view.
- You can restart a run from any view in the SeaScape DB for fast turnaround time without having to recreate the already existing views.
- A view has associated attributes that you can query.
- You can have multiple views for your design. These views can be in a single DB or span across multiple DBs.

Figure 7. Concept of Multiple Views

- A single RedHawk-SC session can have views for:
 - Different corners (Liberty and Extraction)
 - Different conditions (voltage, temperature)
 - Different modes of operation
 - Different vector sets
 - Different vectorless settings
 - Different conditions for IR drop versus electromigration

For example, in the [figure](#), SN1 and SN2 represent VCD and vectorless scenarios, EV1 and EV2 represent two different extraction corners. Also, related views can be traced back, such as, EMV4<-- AV4 <-- SN4 <-- TV2 <-- EV2 <-- DV.

Creating SeaScape Databases and Views

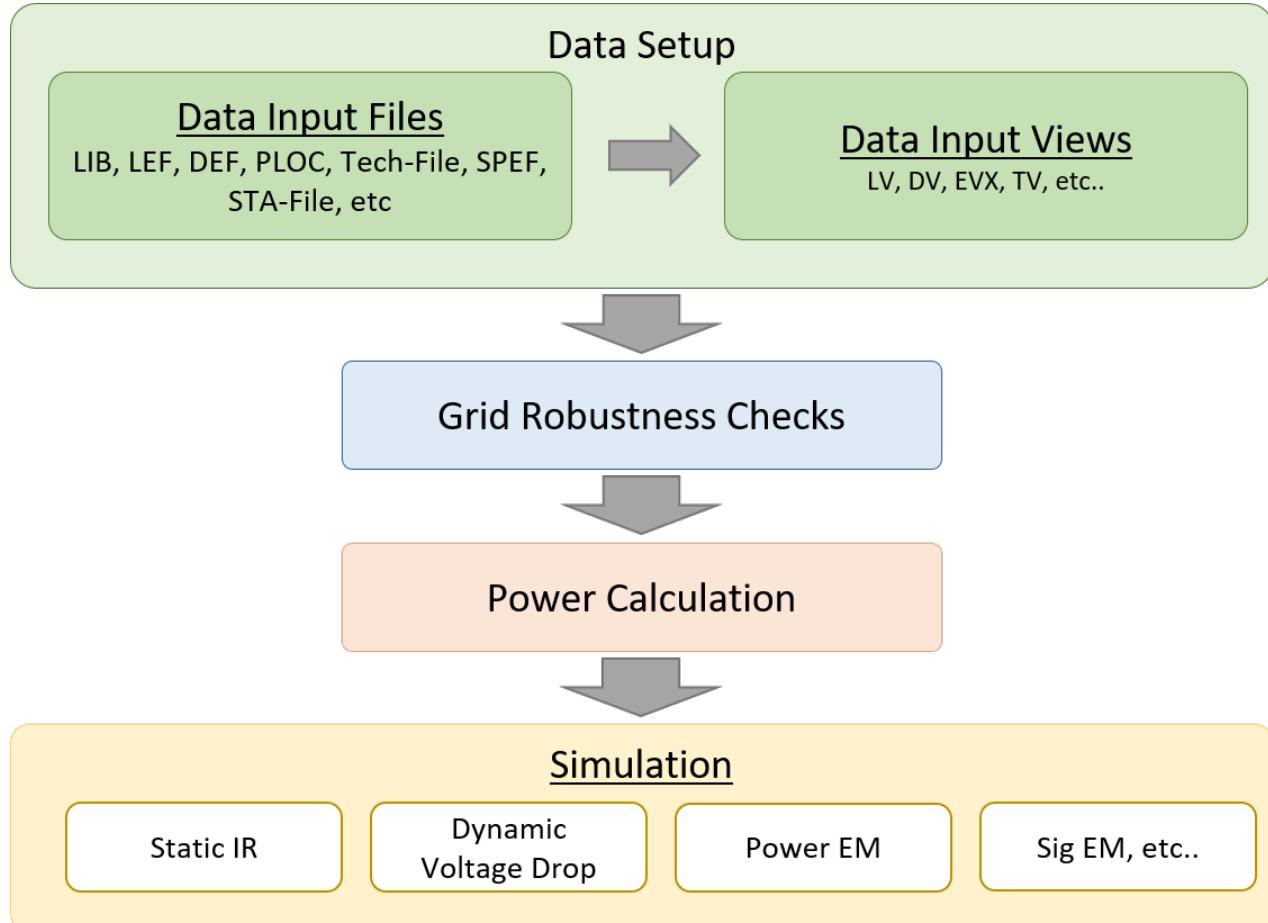
To create a SeaScape DB, use the `open_db` command and specify the DB name. If the DB exists, the command opens the DB. Multiple SeaScape DBs can be opened simultaneously. For more information about using the `open_db` command, see [Methods of Disk Storage](#) on page 453.

To create a view in the DB, use the `create_<type>_view` command where `type` is the view type, such as, design and timing. You can specify the directory name while creating the view.

For more information about SeaScape views, see [RedHawk-SC Views](#) on page 37.

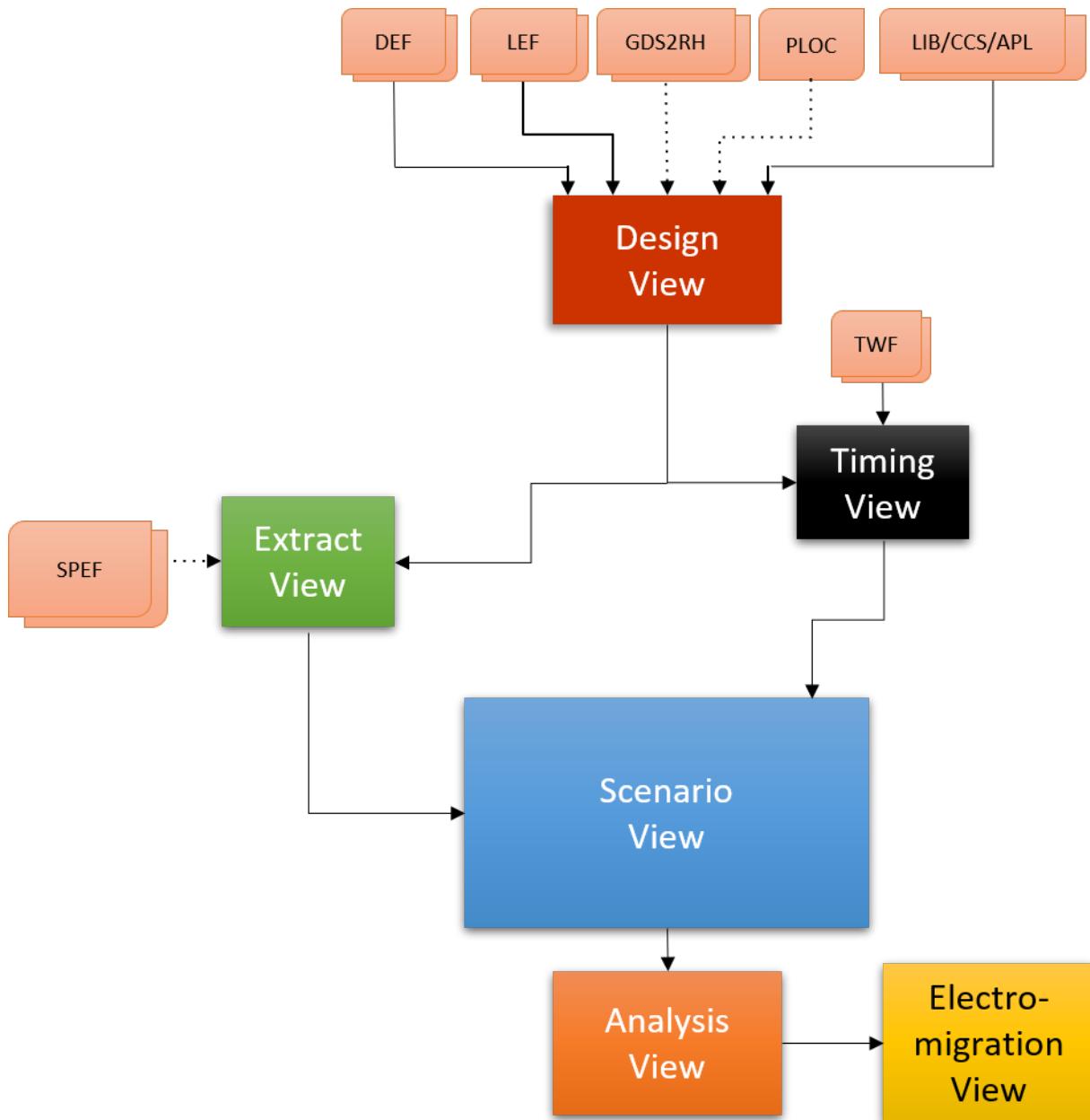
2.1.2. Methodology Overview

The following figure shows the high-level flow to run the RedHawk-SC power integrity analysis flow. In the flow, each step might involve creating multiple views. Though the steps shown are sequential, some views are also created simultaneously. The RedHawk-SC Scheduler manages the view dependencies and parallel execution.



2.1.2.1. Main Views

SeaScape views are classified into main views and supplementary views. The following figure shows the sequence of creation of the main views and the input data consumed by these views.



The functions of the main views are listed as follows:

View	Function
Design View	Stores netlist and layout information Stores library information (through LibertyView)
ExtractView	Stores RC extraction and SPR data Imports and stores SPEF
TimingView	Stores imported STA file and SDC constraints Stores clock associations and slews Creates timing graph

View	Function
ScenarioView	Calculates DC instance currents for static scenario Creates switching scenario and calculates transient instance currents for dynamic scenario
AnalysisView	Stores simulation results
ElectromigrationView	Reads in electromigration rules and generates electromigration results for each wire and via segment

2.1.2.2. Supplementary Views

The RedHawk-SC tool also supports several supplementary views to the main views. The functions of some of the supplementary views are listed as follows:

View	Function
LibertyView	Stores Liberty, APL, and other electrical data
MacroView	Reads in macro models, such as gds2rh and Totem CMM models
TechView	Stores extraction technology information from Ansys technology file, or derived from ITF or iRCX files
ValueChangeView	Stores imported FSDB and VCD information
SwitchingActivityView	Stores propagated and unpropagated toggles Reads in toggle rate settings from SAIF,VCD,FSDB, and other files
PowerView	Stores imported instance power file Processes power and toggle scaling Computes power based on propagated or unpropagated toggles from SwitchingActivityView
SimulationView	Sets up for analysis and reads in package models

2.1.3. Setting the RedHawk-SC Environment

To use the RedHawk-SC tool, follow these steps:

1. Download the RedHawk-SC executable in your environment and then untar it. Install the RedHawk-SC executable.

The typical directory structure is:

```
<path_to_untar>/seascape_release/latest/linux_x86_64_rhel7/bin/redhawk_sc
```

2. Specify the license file name.

You can directly invoke the RedHawk-SC tool using the installation path after you specify the license file. To specify the license file name, use the `LM_LICENSE_FILE` variable. You need not define any environment variables other than the license file name.

3. Launch the RedHawk-SC tool. See [Launching RedHawk-SC](#) on page 25.

2.1.4. Launching Workers

SeaScape supports multiple job scheduling platforms. You can launch workers on the following platforms:

- Your local machine
- Job scheduling platforms including Univa Grid Engine, IBM Spectrum LSF, Altair Accelerator, NetBatch, and SSH
- Cloud based services, including AWS and Microsoft Azure

To create launchers in your respective environments, use the following commands:

- `create_local_launcher(name, num_workers_per_launch=None, ...)`

Creates launcher on your local machine

- `create_ssh_launcher(name, [<comma-separated list of hosts>])`

Creates SSH-based launcher

- `create_grid_launcher(name, submit_command, num_workers_per_launch=None...)`

Creates launcher in your environment

To automatically launch workers as needed, use the `register_default_launcher` command after creating a launcher, for example:

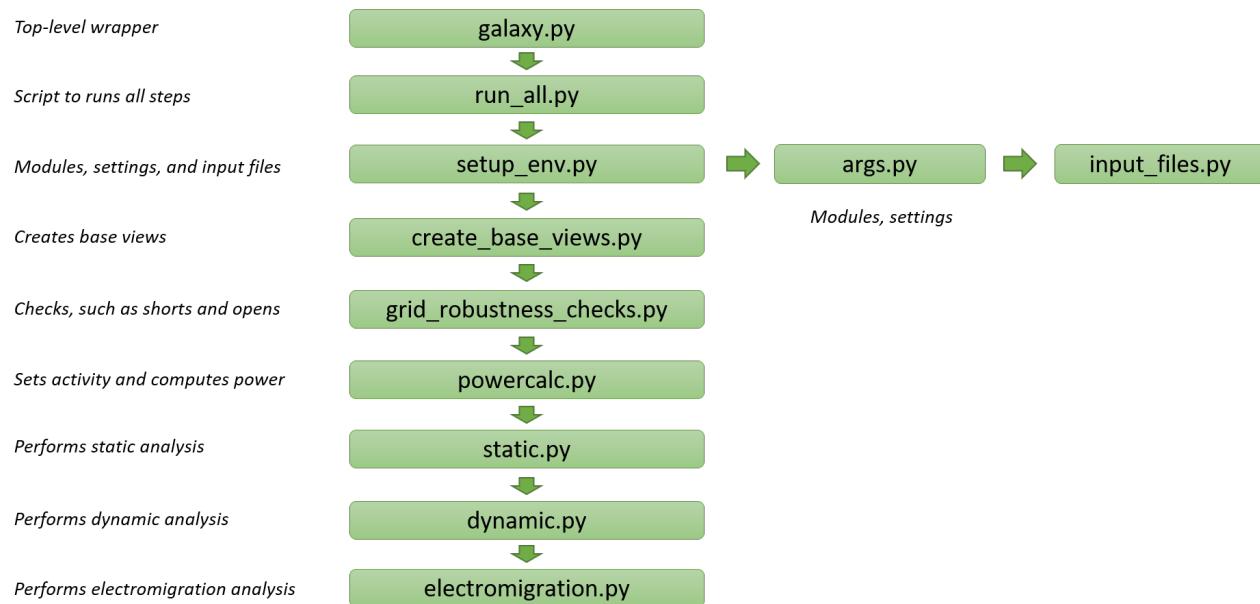
```
register_default_launcher(launcher=LL1, min_num_workers=5, max_num_workers=50)
```

For more details, see [Creating and Launching Workers](#) on page 431.

2.1.5. Working With Python Scripts

RedHawk-SC uses Python as the interface language. You use simple Python scripts to run the tool. The flowchart shows how to execute a typical RedHawk-SC flow with the use of Python script files. You can use a single script file or multiple files. This flow uses multiple files for modularity and ease-of-use. The sections following the flowchart show some of the Python files and snippets.

Note: The following script example is also used in RedHawk-SC training sessions with the training test-case design. The training testcase design is available to you with the RedHawk-SC executable.

Figure 8. Using Python Scripts in RedHawk-SC Flow

galaxy.py

Top-level wrapper file to invoke all the steps in the flow.

```
design_name='galaxy'
include('scripts/run_all.py')
```

run_all.py

Runs the included script files.

```
include('setup_env.py')
include('create_base_views.py')
include('grid_robustness_checks.py')
include('powercalc.py')
include('static.py')
include('dynamic.py')
include('electromigration.py')
```

setup_env.py

Sets up the environment and includes the args.py file to set the options and arguments for view creation commands.

```
import pprint

#SeaScape Scheduler for viewing workers and their jobs real-time
gp.open_scheduler_window()
```

```
#Replace these lines with your environment launcher command
ll = create_local_launcher('local')
register_default_launcher(ll)

# Set the design_data_path variable to central design data path area
design_data_path = '../design_data/'

# Set the DB location settings using the open_db command
db = gp.open_db('db')

# Auto-load view tags from an existing db
# Needed only for incremental(jump-start) runs
populate_view_tags()

# Set the arguments for various view creation commands in args.py
include('args.py')
```

args.py

Specifies modules to import, arguments, and settings, and includes the input_files.py file, which has pointers to the input data files.

```
# Import required Python modules
import package

# Include all design input file pointers
include('../input_files.py')

#Include all tool options and analysis settings
options = get_default_options()
focus_pg_nets = ['VDD', 'VSS']
voltage_levels = {'VDD':1.1, 'VSS':0.0}
swa_settings = {
    'default_clock_period' : 8e-09,
    'default_clock_pin_toggle_rate' : 2.00,
    . . . }
```

input_files.py

Specifies the data input files and settings. You can specify the input files as Python lists or dictionaries depending on requirements.

```
# Specify design files
def_files = [
    design_data_path + '/defs/Galaxy.def'
    . . .
]

lef_files = [
    design_data_path + '/lefs/switch_cell.lef',
    . . .
]
```

create_base_views.py

Calls the arguments in the args.py file in view creation commands. The args.py file, in turn, calls the input_files.py file that contains input data specification.

```
... ...
dv0 = db.create_design_view(tech_view=nv, lib_views=lv, **dv0_args)
...
...
```

2.1.6. Launching RedHawk-SC

You can use multiple options and arguments to launch the RedHawk-SC tool:

```
<installation_path>/bin/redhawk_sc <python_script> -i --console -r <gp_dir> -g
```

These options and arguments are defined as follows:

- <python_script>

A Python script that contains the commands for execution by the tool. When this script is the only argument, the tool runs in the Batch mode by default. For example,

```
<path_to_rhsc_installation>/bin/redhawk_sc galaxy.py
```

The tool executes the listed commands in the script and automatically exits the session. If the script has GUI-based commands, such as scheduler GUI and console, the tool opens interactive interfaces as needed.

- -i

Launches an interactive shell on the launch terminal. You can use the shell to sequentially run queries or execution commands. If you use the python script with the `-i` option, the tool directly executes the script and the interactive shell opens. For example,

```
<path_to_rhsc_installation>/bin/redhawk_sc -i galaxy.py
```

The shell remains active even after the script is executed. To automatically exit the shell, use the `exit()` command in the script.

- --console

Launches the RedHawk-SC console and opens an interactive shell at the launch terminal. Use the `--console` option to launch a new run or open an existing database to view and analyze results. See [Launching the RedHawk-SC Console](#) on page 27.

- -r <gp_dir>

Remotely attaches to an active or a completed session. This option enables you to query data from an active session.

```
<path_to_rhsc_installation>/bin/redhawk_sc -r <gp_dir>
```

For example, if an active session has completed ScenarioView creation and is currently creating the AnalysisView, you can remotely login to the session and query data for all completed stages without waiting for the run to complete. Multiple users can attach to the same session from multiple machines to query data and view results.

- -g

Launches the Scheduler GUI. See [Launching the Scheduler GUI](#) on page 28.

- -l

Suppresses creating a new gp directory.

- `-l -r <gp_dir> -g`

Remotely attaches to an active or a completed session without creating a new gp directory, and opens the scheduler window to go through the job distribution across workers, memory consumption per job or worker, and runtime per stage.

You can use these options to remotely monitor active jobs. You can launch jobs in batch mode in a non-interactive queue and launch the Scheduler GUI from a local interactive terminal to view progress of the job. For completed sessions, you can use these options to profile performance and runtime.

2.1.7. Exploring Log Files

By default, RedHawk-SC log files are stored in a directory named gp or gp.<number>. For each session, the tool creates a new directory in the run area. For example:

- Session 1 creates gp
- Session 2 creates gp.1
- Session 3 creates gp.2

The latest.gp link automatically points to the latest gp directory in the run area. Some of the important files and directories in the gp directory are as follows:

- `run.log`
Main log file that contains all the information, error, warning, and assert messages.
- `run_v.log`
Contains more detailed debug data than `run.log`
- `Worker*.log`
Contains per worker information, error, and warning messages.
- `.._w(xx)_split.bash.log`
Contains information about launcher setup issues when workers do not start.
- `input_scripts/`
Contains all scripts included in the run.
- `interactive_history.py`
Contains a copy of all commands sourced on the RedHawk-SC shell.

2.1.8. Viewing Command Help

You can access the help for a RedHawk-SC command from the RedHawk-SC interactive shell. For information about a RedHawk-SC command, specify the `help` command at the RedHawk-SC tool command prompt as follows:

```
help(SeaScapeDB.create_modified_design_view)
```

or

```
help(<db_name>.create_design_view)
```

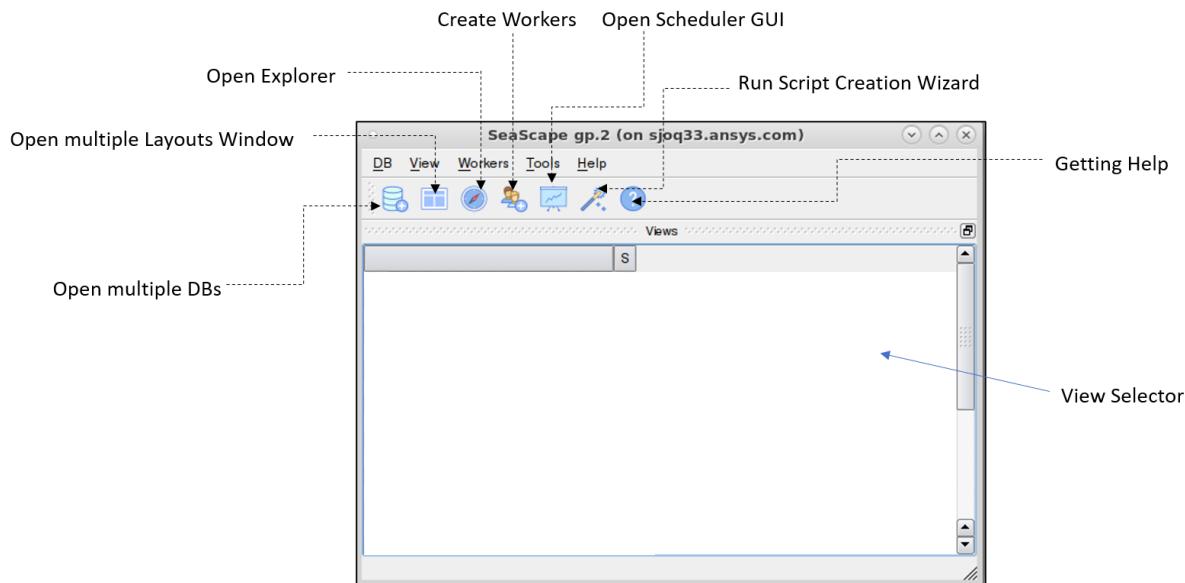
where `<db_name>` is the name of the actual database that you create. For example, if your database name is *galaxy*, specify:

```
help(galaxy.create_design_view)
```

You can also access the command help from GUI. See [Viewing Command Help From Console](#) on page 492.

2.1.9. Launching the RedHawk-SC Console

The RedHawk-SC console is an interactive analytics interface.



To open the console from the installation directory, use the following command:

```
<path_to_rhsc_installation>/bin/redhawk_sc <python_script> --console
```

To open the console from the RedHawk-SC interactive shell, use the following command:

```
open_console_window()
```

The main functions of the console are as follows:

- **Open DB**
Opens multiple DBs and displays all the views of the DB.
- **Layout Window**
Opens the design layout window.
- **Create Workers**
Launches workers interactively.
- **Open Scheduler GUI**
Opens the Scheduler GUI.

- **Script creation wizard**

Assists you to create scripts with commonly used Python code.

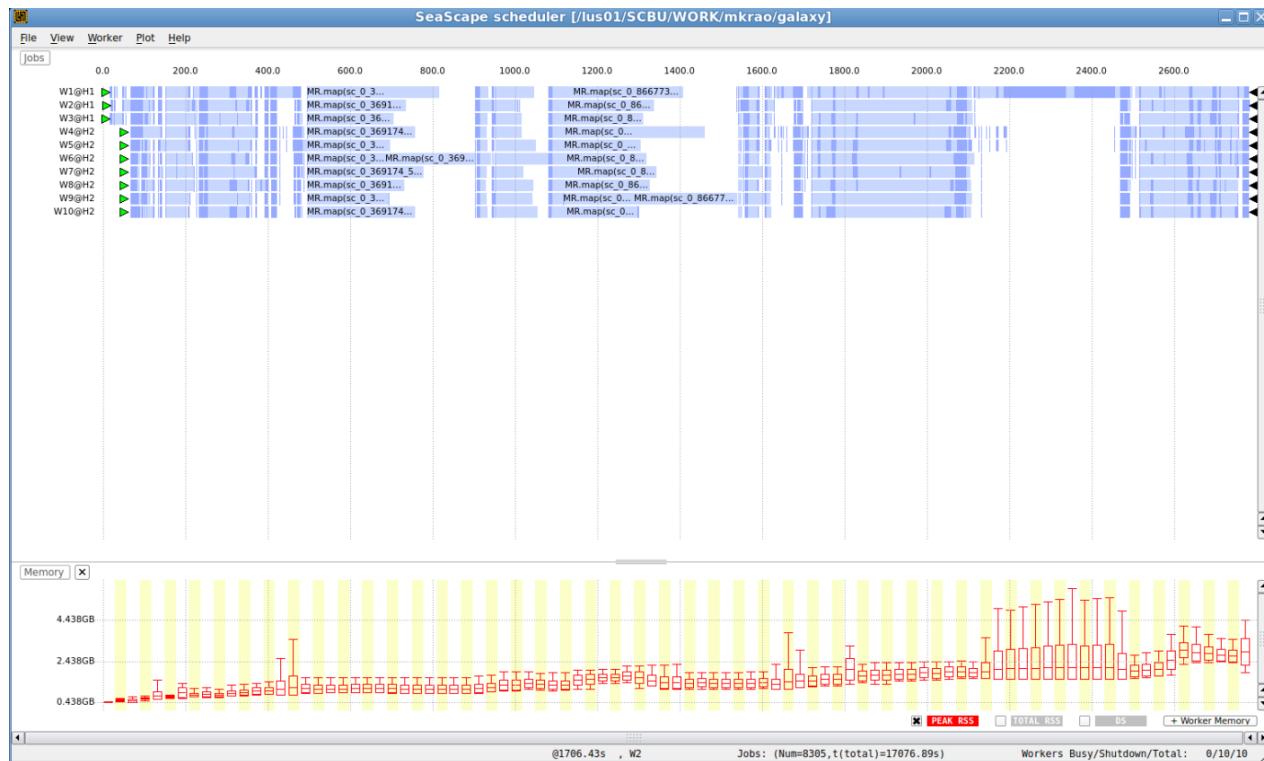
- **Help**

Gets tool help.

For detailed information, see [RedHawk-SC GUI](#) on page 463.

2.1.10. Launching the Scheduler GUI

The Scheduler GUI provides real-time visualization of workers and their jobs. It displays the jobs executed per worker on the y-axis and the time on the x-axis.



To open the Scheduler GUI for an ongoing or completed run from the command line, use the following command:

```
redhawk_sc -g -l -r <path_to_gp_directory>
```

To open the Scheduler GUI from a Python script, use the following command in the script:

```
open_scheduler_window
```

You can also open the Scheduler GUI from the console. The Scheduler GUI provides the following information:

- Worker information
- Host information
- Highlights jobs using run-time and memory
- Total number of workers and jobs with names
- Worker statistics plots including worker memory, machine load, read rate, total read.
- Identifies network issues using red boxes and plots
- Identifies slow jobs with a small number of active workers using long performance sticks

For detailed information, see [RedHawk-SC Scheduler GUI](#) on page 436.

2.2. Preparing Input Data

The following topics describe how to prepare your input data for power integrity analysis.

2.2.1. Specifying Input Data

You can specify the following input data types to run RedHawk-SC for different types of analysis.

2.2.1.1. Technology Data

The RedHawk-SC technology file contains process specific information about the metal layer stack to extract the design. This file also contains electromigration rules for the metal stack.

2.2.1.2. Design Data

Design data includes the following input data files.

Layout (DEF File)

A Design Exchange Format (DEF) file includes the physical description of instances, power and ground network, and other circuit elements. lefdef 5.8 and earlier versions are supported. For multiple hierarchies, the .def files from each hierarchy should be specified.

DEF files can be defined as a list of strings in the input file definition Python commands as follows:

```
def_files = [
    '/design_path/def/design_top.def.gz',
    '/design_path/def/design_blockA.def.gz',
    '/design_path/def/design_blockB.def.gz',
]
```

The DEF files are used to create the DesignView.

Timing File

The timing file contains instance-specific minimum and maximum transition times and defines the timing windows and clock network data. This file is required for signoff accuracy. The timing file:

- Influences the clock roots for the clock network.
- Influences the frequency of each instance.
- Determines the switching time for flops/sequential instances.
- Loops up the input transition time and output transition times for instances.

Timing files can be defined as a dictionary (dict) in the input file definition Python commands as follows:

```
tw_files = [{  
    'instance_name': '',  
    'file_name': '/path/cell_name.sta.gz'}]
```

The timing file is used to create the TimingView.

If you have performed STA with the Synopsys PrimeTime tool, use the `write_rh_file` command to write out the timing file. For other STA tools, contact the vendor for the timing file script.

Pad Location Model File

The pad location file (.ploc) provides the list of power and ground sources (connections, such as `NET_NAME`), x and y locations, and layer assignments that the tool identifies during network extraction.

```
ploc_file='/path/top_level.ploc'
```

The format of the .ploc file is:

```
# pad location file format 5-6 columns
<unique name> <x-loc> <y-loc> <metal> <NET_NAME> <spice_port>
vdd1 6808 11091 metal7 VDD
vss1 7038 11612 metal7 VSS
...
```

To read in the pad location file, you need to create a `ModifiedDesignView`.

Package Model File

To import and prepare package model files:

- A user Python script, `package.py`, should have all the necessary functions for preparing the package models and creating the necessary wrapper to connect the package to the die when reading in a model from Sentinel or any package model containing the CPP header.
- For a chip package analysis (CPA) model, you only need to generate an ASCII padfile (you can reformat the ASCII .ploc file generated by CPA) and directly read in both files.
- N-port, 2-port, and lumped RLC values are supported.
- The prepared package data is read in while creating the `create_simulation_view` stage.

2.2.1.3. Library Data

RedHawk-SC requires the physical and electrical models along with functional characteristics for all library elements in a design. The library data required by RedHawk-SC are described in the following sections.

LEF Model

Physical description of library cells is defined in the Library Exchange Format (LEF) models. lefdef 5.8 and earlier versions are supported. One of the .lef files must contain the technology layer and via information.

LEF files can be specified as a list in the input Python script for RedHawk-SC as follows:

```
lef_files = [
    '/path/mergedTech.lef',
    '/path/stdcellsA.lef',
]
```

The LEF files are directly used to create the DesignView for the design.

Liberty Models

The Synopsys Liberty (.lib) files contain various characteristics of the libraries that get used in a design. RedHawk-SC requires Liberty files to understand the type and functionality of each cell along with its power and timing characteristics. To perform a static or dynamic voltage drop analysis, RedHawk-SC requires the Liberty files for all standard cells. Liberty files can be specified as a list in the input Python script for RedHawk-SC as follows:

```
lib_files = [
    '/path/stdcellsA.lib.gz',
    '/path/stdcellsB.lib.gz',
]
```

User-defined custom .lib files can be specified in a similar list format in the input Python script for RedHawk-SC.

```
lib_files_custom = [
    '/path/custom.lib',
]
```

The Liberty files are used to create the LibertyView for the design.

APL Model

The Ansys Power Library (APL) for cell characterization is used to create accurate switching current waveforms (profiles), output-state dependent decoupling capacitance (intrinsic decoupling capacitance), equivalent power circuit resistance called Effective Series Resistance (ESR), switching delay, and leakage current. These are created for multiple conditions of the desired set of library cells. This data is required for accurate RedHawk-SC dynamic analysis.

APL files can be specified as a dict in the input Python script for RedHawk-SC as follows:

```
apl_files = [
    {'file_name': '/path/cellAND3X.spiprof'},
    {'file_name': '/path/cellNAND2X.spiprof'},
]
```

The APL files are used to create the LibertyView for the design.

For characterizing APL views, see the help for *Characterization Using ANSYS Power Library*.

Switch Model

In low power designs, header and footer switches (also known as power gates) are characterized using the *aplsw* utility and a switch model file through SPICE simulation. Non-ideal piecewise linear control pin inputs can be accommodated in the current modeling. The model from *aplsw* can be loaded in RedHawk-SC and used.

The *aplsw* model is one of the arguments in creating a LibertyView for the design..

The following example shows how to read in the *aplsw* model file in RedHawk-SC.

```
switch_files=[{'file_name': '../design_data/apl/switch.model', 'temperature': 110}]
lv = db.create_liberty_view(liberty_file_names = lib_files,
                            apl_file_names = apl_files,
                            apl_switch_file_names = switch_files,
                            tag = 'lv', settings=settings, options = options)
```

RedHawk-SC can also derive on-state resistance for a switch cell from a CCS power library if no *aplsw* model is available.

2.2.1.4. Macro Data

RedHawk-SC can handle multiple formats for macros/IPs:

- GDS2RH models for macro cells/IPs

Generated using the *GDS2RH* utility that reads in a GDS file along with a layer map file and a LEF (optional) to create a DEF, LEF, and a power distribution ratio file.

- CMM (cell/mmx view) models for macros/IPs

Generated from a transistor-level analysis of the macro/IP in Totem.

This example defines some custom models in RedHawk-SC:

```
model_files = ['<path>/gds2rh_model_macro1', '<path>/cmm_model_macro2',
'<path>/cmm_model_macro3']
# Here the directories containing the LEF, DEF, pratio, apl data etc for the
# models can be given in a list
macro_view = db.create_macro_view(model_details=model_files, tag='macro',
options=None)
```

2.2.1.5. Input Data Specification Example

This example shows a typical Python snippet with different input data specifications.

```
def_files = [design_data_path + '/defs/Galaxy.def', ...]
lef_files = [design_data_path + '/lefs/nangate_stdcell.lef', ...]
lib_files = [design_data_path + '/libs/switch.lib', ...]
tech_file = design_data_path + '/tech/GENERIC.tech'
tw_files = [design_data_path + '/timing/Galaxy.timing.gz']
switch_files = [{'file_name':design_data_path + '/switch_model/aplsw.out',
'temperature':125}]
ploc_file = design_data_path + '/ploc/design.ploc'
pkg_file = design_data_path + '/package/galaxy.pkg'
edited_ploc_file = design_data_path + 'ploc/edited.ploc'
spef_files = [design_data_path + '/spef/Galaxy.temp1_25.spef', ...]
apl_files = [{'file_name': design_data_path + '/APL/VG/Nangate45_FF.current'}]
```

2.2.2. Creating Base Views

A SeaScape view that is created by directly consuming one or more input files is called a base view. See [figure](#). Base views are required views to run power integrity flows.

To check data integrity of input files, you must first create the base views. This example shows the Python command syntax to create base views.

```
include('setup_env.py')
lv = db.create_liberty_view(**lv_args)
nv = db.create_tech_view(**nv_args)
dv0 = db.create_design_view(tech_view=nv, lib_views=lv, **dv0_args)
dv = db.create_modified_design_view(dv0, **dv_args)
```

```

ev = db.create_extract_view(dv, nv, **ev_args)
sv = db.create_simulation_view(ev,
package=package.parse_ploc_spice_func(pkg_file, dv=dv), **sv_args)
evx = db.create_extract_view_from_files(dv, **evx_args)
tv = db.create_timing_view(dv, **tv_args)
...
...

```

Each view creation command has arguments to call the input files. For modularity, these arguments are defined in a different file shown in the following example. For more information, see [Working With Python Scripts](#) on page 22.

```

lv_args = dict(
    liberty_files=lib_files,
    apl_switch_files=switch_files,
    apl_files=apl_files,
    tag='lv',
    settings=settings, options=options)

dv0_args = dict(
    def_files=def_files,
    lef_files=lef_files,
    top_cell_name='Galaxy',
    tag='dv0',
    settings=settings, options=options)

dv_args = dict( eco_commands=package.parse_ploc_func(ploc_file),
    tag='dv',
    settings=settings, options=options)

tv_args = dict(
    timing_window_files=tw_files,
    tag='tv',
    settings=settings, options=options)

nv_args = dict(
    tech_file_name=tech_file,
    tag='nv',
    settings=settings, options=options)

ev_args = dict(
    tag='ev',
    settings=settings, options=options)

evx_args = dict(
    input_files=spref_files,
    tag='evx',
    settings=settings, options=options)

sv_args = dict(
    tag='sv',
    settings=settings, options=options)

```

2.2.3. RedHawk-SC Data Integrity Checks

The tool performs the following library-level and design-level data integrity checks.

Table 1: Library-Level Data Integrity Checks

Check type	Importance
APL/CCS current check	RedHawk-SC gets the cell PG current waveforms from APL or CCS-power data. APL is generated using the Ansys APL utility. In the absence of APL or CCS current data, RedHawk-SC creates a triangular current waveform using Liberty nonlinear power model (NLPM) data. Analysis using NLPM-based current profiles are usually pessimistic than APL or CCS based analysis.
APL/CCS capacitance check	RedHawk-SC gets the cell intrinsic capacitance information from APL/CCS capacitance data. APL capacitance is generated using the APL utility. APL/CCS capacitance data contains the effective series capacitance (ESC) and effective series resistance(ESR) values for each PG pin of the cell. In the absence of APL/CCS capacitance data, RedHawk-SC does not use any cell intrinsic capacitance values in the simulation. So, analysis gives pessimistic IR drop values.
Liberty check	RedHawk-SC uses Liberty for calculating the leakage and internal power for the cell. In the absence of Liberty, RedHawk-SC does not include these instances in power calculation. The timing, functionality, pgarc information of the cell is not available in absence of Liberty files.
LEF check	Instances without LEF cannot be hooked up to power grid, so RedHawk-SC excludes these cells in analysis. So, LEF coverage issue is critical.
Macro check	RedHawk-SC expects GDS2RH or Totem- CMM data for all hard macros in the design. GDS2DEF model provides the physical representation of power grid geometries inside the macro. Without this, the macros get black-boxed and affect the accuracy of analysis.

Table 2: Design Data Integrity Checks

Check type	Importance
DEF check	DEF data can have multiple issues, such as shorts, unconnected instances, unconnected wires, unconnected vias, and missing vias
SPEF check	In static analysis, RedHawk-SC uses SPEF to compute the switching power. In dynamic analysis, the tool needs the SPEF for picking the correct current waveform from the APL/CCS lookup tables. Absence of SPEF causes incorrect power or current waveform calculation.

Check type	Importance
STA check	In static analysis, RedHawk-SC uses frequency and transition time information from the STA file. In dynamic analysis, the tool also uses timing window information from the STA file. The STA file is also used to identify clock instances in the design. In the absence of an STA file, the tool is capable of using the propagation flow.
VCD check	For VCD based runs, gate or RTL, checks for full coverage for all nets (gate VCD) or registers and primary inputs (RTL VCD).

2.2.4. Checking Data Integrity

After creating the base views, you should check the data integrity of input files to resolve any required data integrity issues before proceeding with further analysis. The following example shows the commands to check data integrity of input files, specify the directory to store reports, and specify and print summary report files.

```
di_reports = data_integrity_reports
reports_dir = 'reports/data_integrity/'
if not os.path.exists(reports_dir):
    os.makedirs(reports_dir)
dv_summary = di_reports.create_dv_di_reports(dv, detailed_reports=True,
                                             detailed_reports_directory=reports_dir)
evx_summary = di_reports.create_spf_data_integrity_reports(evx,
                                                          detailed_reports=True,
                                                          detailed_reports_file_name=reports_dir+'/spf_detailed_reports')
tv_summary = di_reports.create_timing_view_data_integrity_reports(tv,
                                                                  per_instance_report=True,
                                                                  file_name={'timing_window': reports_dir+'inst_no_tw.rpt',
                                                                 'slew': reports_dir+'inst_no_slew.rpt'})
write_to_file(reports_dir+'dv_di_summary.rpt', pprint.pformat(dv_summary.get()))
write_to_file(reports_dir+'evx_spf_di_summary.rpt', pprint.pformat(evx_summary.get()))
write_to_file(reports_dir+'tv_di_summary.rpt', pprint.pformat(tv_summary))
```

The tool checks the integrity of input data and generates both summary and detailed reports. For multiple views of the same type, such as multiple design views, the tool generates multiple reports.

By default, a summary report is named <view name>_di_summary.rpt, such as dv_di_summary.rpt, tv_di_summary.rpt, and evx_spf_di_summary.rpt. The detailed report names include the view and the check, such as dv_apl_cap_check.rpt, inst_no_slew.rpt, and spf_detailed_reports.spf_check. Some typical report example snippets are as follows:

tv_di_summary.rpt

```
{'ch_data': <gp.ChunkedData object at 0x2ab0402f0b10>,
 'slew_summary': defaultdict(<type 'int'>, {'Sequential': 70726, 'Macro': 8, 'Combinational': 283055}),
 'total_count': defaultdict(<type 'int'>, {'Sequential': 72072, 'Macro': 8, 'Combinational': 305402}),
 'tw_summary': defaultdict(<type 'int'>, {'Sequential': 70649, 'Macro': 8, 'Combinational': 290388}}}
```

evx_spf_di_summary.rpt

```
{'di_data': <gp.ChunkedData object at 0x2b18f1750d50>,
 'summary': {'spf_check': {'Combinational': {'uncovered_instances': 9890},
                           'Sequential': {'uncovered_instances': 70736},
                           'total': {'total_instances': 380028,
                                     'uncovered_instances': 80626}},
              'zero_cap_check': {'total': {'total_instances': 380028}}}}
```

dv_apl_cap_check.rpt

```
{'apl_cap_check': {'Clock': {'covered': 0, 'total': 0},
                     'Combinational': {'covered': 67, 'total': 67},
                     'Decap': {'covered': 0, 'total': 0},
                     'Filler': {'covered': 0, 'total': 0},
                     'ICG': {'covered': 0, 'total': 0},
                     'MBFF': {'covered': 0, 'total': 0},
                     'Macro': {'covered': 1, 'total': 1},
                     'Power_Gate': {'covered': 0, 'total': 1},
                     'Sequential': {'covered': 8, 'total': 8}},
   'apl_current_check': {'Clock': {'covered': 0, 'total': 0},
                         'Combinational': {'covered': 67, 'total': 67},
                         'Decap': {'covered': 0, 'total': 0},
                         'Filler': {'covered': 0, 'total': 0},
                         'ICG': {'covered': 0, 'total': 0},
                         'MBFF': {'covered': 0, 'total': 0},
                         'Macro': {'covered': 1, 'total': 1},
                         'Power_Gate': {'covered': 0, 'total': 1},
                         'Sequential': {'covered': 8, 'total': 8}}},
```

inst_no_tw.rpt

#	Instance	Failing_pins
	ZBUF_314_inst_65462	['Z']
	ZBUF_2_inst_65247	['Z']
	ZBUF_603_inst_65321	['Z']
	ccd_drc_inst_65432	['Z']
	ccd_drc_inst_65433	['Z']
	ccd_drc_inst_65434	['Z']
	ccd_drc_inst_65435	['Z']
	core2.regfile_data_memory.NAND2_X1_17	['ZN']
	core2.regfile_data_memory.NAND2_X1_1898	['ZN']
	core2.regfile_data_memory.NAND2_X1_1861	['ZN']
	core2.regfile_data_memory.NAND2_X1_1899	['ZN']
	core2.regfile_data_memory.NAND2_X1_1936	['ZN']
	core2.regfile_data_memory.NAND2_X1_1998	['ZN']
	core2.regfile_data_memory.NAND2_X1_2002	['ZN']
	core2.regfile_data_memory.NAND2_X1_2100	['ZN']
	core2.regfile_data_memory.NAND2_X1_2137	['ZN']
	core2.regfile_data_memory.NAND2_X1_2139	['ZN']
	core2.regfile_data_memory.NAND2_X1_2140	['ZN']
	core2.regfile_data_memory.NAND2_X1_2145	['ZN']
	core2.regfile_data_memory.NAND2_X1_2159	['ZN']
	core2.regfile_data_memory.NAND2_X1_2160	['ZN']
	core2.regfile_data_memory.NAND2_X1_2186	['ZN']
	core2.regfile_data_memory.NAND2_X1_2192	['ZN']
	core2.regfile_data_memory.NAND2_X1_2193	['ZN']

For more details about the available data integrity checks, use the `help` command at the tool command prompt:

```
help(data_integrity_reports)
```

3: RedHawk-SC Views

A SeaScape **View** is a container format that holds data related to various design characteristics. These views can represent physical, electrical, logical, graph and simulation type data. Each view has its own input requirements and dependencies. A sequence of views can be used to perform different types of analysis inside RedHawk-SC. This chapter describes the RedHawk-SC commands to create important views and includes the following topics.

- [SeaScape Views](#) on page 37
 - [DesignView](#) on page 39
 - [ModifiedDesignView](#) on page 41
 - [ExtractView](#) on page 42
 - [TimingView](#) on page 43
 - [ScenarioView](#) on page 44
 - [AnalysisView](#) on page 46
 - [ElectromigrationView](#) on page 47
 - [ThermalView](#) on page 48
 - [LibertyView](#) on page 49
 - [MacroView](#) on page 52
 - [TechView](#) on page 52
 - [ValueChangeView](#) on page 53
 - [SwitchingActivityView](#) on page 54
 - [PowerView](#) on page 55
 - [SimulationView](#) on page 56
-

3.1. SeaScape Views

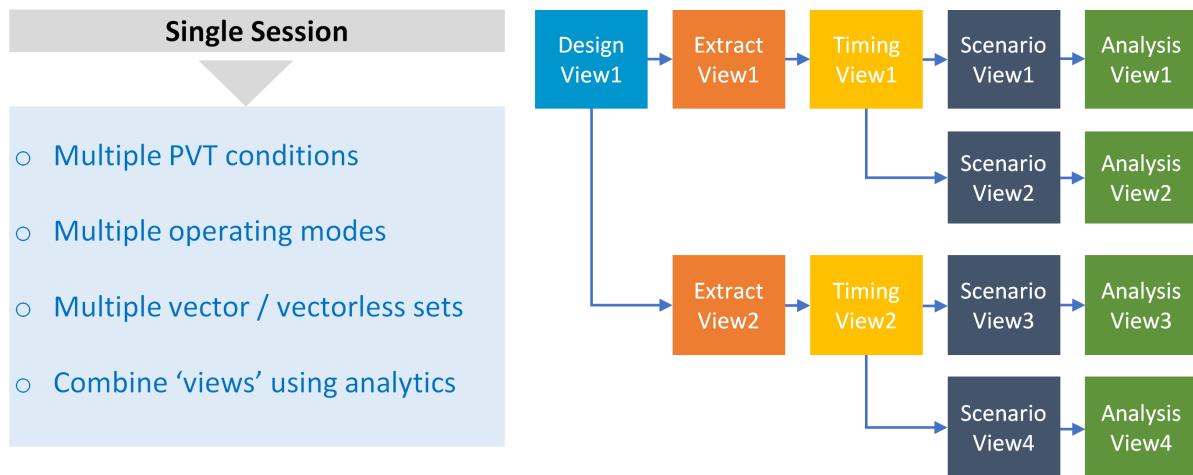
SeaScape views have the following characteristics:

- Each view is stored in a Unix directory under the main SeaScapeDB directory.
- A **tag** defines the name of the view that is also used as the Unix directory name in the SeaScapeDB for view-related data storage.
- Automatic versioning occurs when the **tag** is reused for a view. The tag/directory will have the suffix #v1, #v2... appended (for `tag='tv'`, the result on disk is `./<SeaScapeDB>/tv#v1`).
- A view is written once but can be read many times. Multiple views can be open simultaneously.
- You can restart analysis from any saved view for faster turnaround time and avoid recreating the views that are already completed.
- Since views are objects, they also can be queried.
- You can also combine data from multiple views for data analytics.

There are five main views and multiple auxiliary views in the SeaScapeDB.

Figure 9. Main Views

You can combine views for better analytics.

Figure 10. Single Session, Combined Views for Analytics

Mandatory options Syntax for `create_<type>_view` Commands

You must specify the `options` argument with all `create_<type>_view` commands. Results differ when you do not specify `options` and when you specify `options` by using the `get_default_options()` command. This helps to catch mistakes in your scripts. You should specify `options`, as shown in the following example:

```
options = get_default_options()
create_liberty_view(lib_files=lib_files, options=options)
```

Omitting `options`, such as in, `create_liberty_view(lib_files=lib_files)`, causes a FATAL error.

settings Dict for `create_<type>_view` Commands

In the RedHawk-SC tool command syntax, each `create_<type>_view` command has a `settings` dict as an argument. Most syntax (other than input views, input files, tag, and options) are specified as keys within the `settings` dict.

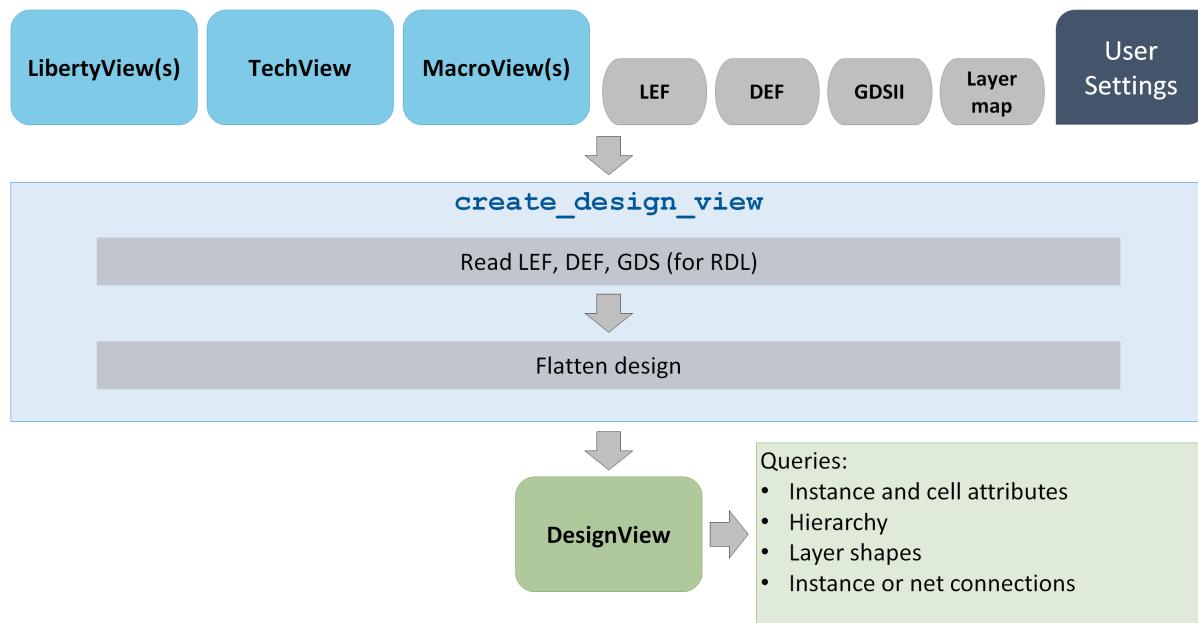
The following topics describe how different views are used in a generic flow.

3.2. DesignView

The DesignView represents the logic netlist and the physical layout of the design and includes the electrical models of cells used in the design.

The DesignView is the root parent of most views in the SeaScape database structure. One DesignView can drive many types of analysis such as static IR drop, dynamic voltage drop, and electromigration. These analyses can be for different PVT conditions and switching scenarios. The DesignView is a mandatory view for any type of analysis in RedHawk-SC.

Figure 11. DesignView



The DesignView contains the following information:

- Main physical information of the design inputs, DEF and LEF files.
- RDL overlay of GDS.
- Ansys Power Library (APL) and Liberty data.
- Physical macro model views.

The following topics describe how to use the `create_design_view` command:

- [Checking Maskshift Sanity](#) on page 39
- [Defining Current Fractions for Different Shapes of a Signal Pin](#) on page 40
- [Specifying Non-Tristate Driver Criteria](#) on page 40
- [Dependencies](#) on page 40

Checking Maskshift Sanity

The tool offers the flexibility to ignore component maskshift sanity checks and continue the DesignView run. To do so, use the `component_maskshift_sanity_check_level` key. The key can take one of the following values:

- **warning** (default): The tool enables the sanity check to issue suitable warnings on check failure and continues with DesignView creation. For example,

```
dv_settings = {
    'component_maskshift_sanity_check_level' : 'warning', ...
}
dv = db.create_design_view(..., settings=dv_settings)
```

- **off**: The tool disables the sanity check.
- **fatal**: The tool enables the sanity check to issue suitable errors on check failure and terminates.

Defining Current Fractions for Different Shapes of a Signal Pin

By default, the tool evenly distributes the current among the different shapes of a signal pin. For improved current distribution during signal net electromigration analysis, you should define custom phantoms while creating DesignView and specify the current fractions for the different shapes of a signal pin under the `custom_cell_phantoms` key. See [Specifying Current Distribution Fractions for Signal Pin](#) on page 272.

Specifying Non-Tristate Driver Criteria

A net that connects to both an inverter output pin and the output (or inout) pin of a bump, through-silicon via (TSV), or a pad cell typically has multiple strong drivers, that is, non-tristate drivers. Counting the input port of top cell and output pins of leaf cells for load division across pins is useful in improving the load-sharing in multidriver nets.

By default, the tool uses the following criteria to determine the number of strong (non-tristate) drivers in multidriver nets:

```
strong_driver_criteria = {'input_port':'on', 'output_pin' : 'on',
                           'inout_port' : 'off', 'inout_pin':'if_liberty_nldm'}
```

This is because:

- An input port is typically connected to an external strong driver and it should not be tristate, so the default is `on`.
- An output pin is generally a normal driver cell, so the default is `on`.
- An inout port is typically connected to an external tristate device, so the default should be `off`.
- An inout pin is generally used in a bidirectional pad cell that can be tristate, so the default is `if_liberty_nldm`.

To change the default `strong_driver_criteria` setting, use the following syntax:

```
strong_driver_criteria = {'input_port' : <value1>, 'output_pin': <value2>,
                           'inout_port' : <value3>, 'inout_pin': <value4>}
                           dv_settings['strong_driver_criteria'] = strong_driver_criteria
                           db.create_design_view(..., settings=dv_settings ,...)
```

Value of both `input_port` and `inout_port` is `on` or `off`. For leaf cell pins, the value of both `output_pin` and `inout_pin` is `on` or `if_liberty_nldm`.

Dependencies

A DesignView has dependencies on the following views.

- LibertyViews for different corners. See [LibertyView](#) on page 49.
- MacroView that contains physical macro models. See [MacroView](#) on page 52.

- TechView that contains LEF and DEF layer maps. See [TechView](#) on page 52.

For more details, use the `help` command at the tool command prompt:

```
help(SeaScapeDB.create_design_view)
```

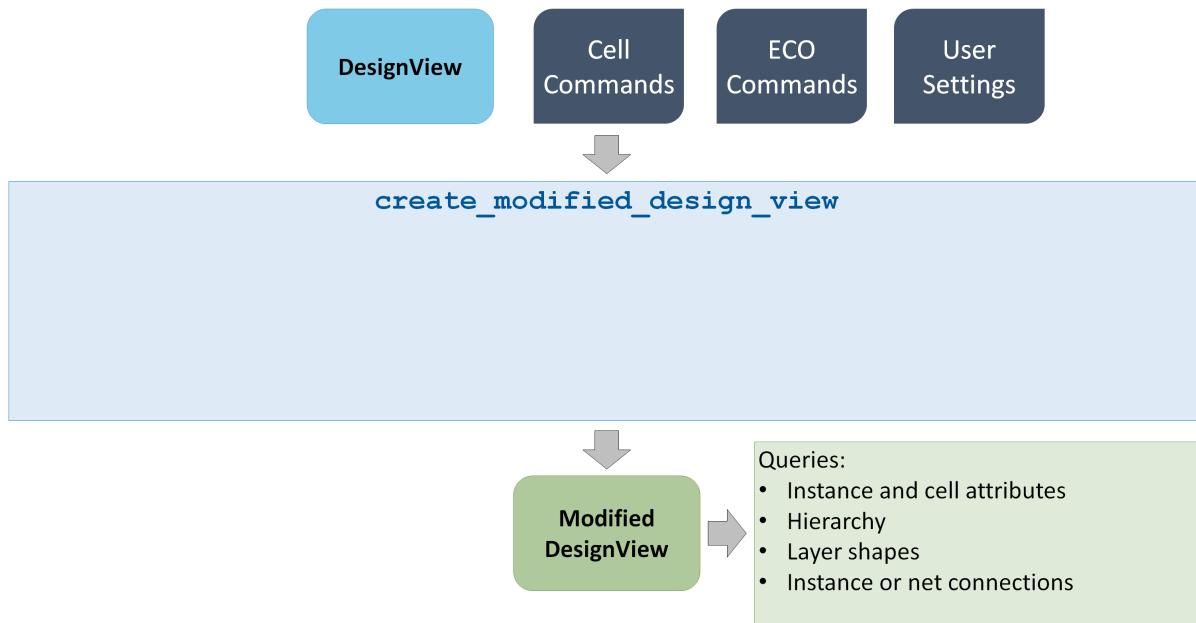
3.3. ModifiedDesignView

A ModifiedDesignView enables you to make small changes to an existing DesignView, such as adding dummy RDL patterns to a design. It is also the primary mechanism to read PG source or bump locations.

The ModifiedDesignView can store the following information.

- The PG voltage source locations or to import a ploc file.
- Modify geometries using by adding/deleting metals and vias or create dummy RDL patterns.
- Information to reference the main DesignView.

Figure 12. ModifiedDesignView



Dependencies

A ModifiedDesignView has dependency on the following view:

DesignView that contains the logic, electrical, and physical representations of the design. See [DesignView](#) on page 39.

For more details, use the `help` command at the tool command prompt:

```
help(SeaScapeDB.create_modified_design_view)
```

3.4. ExtractView

The ExtractView holds the parasitic data for the design, resistors, capacitors and inductances, as a result of processing all the geometries within the design and computing all the RC and, optionally, L components. It also stores the analyzed shorts, disconnects, and Shortest Path Resistance (SPR) [least resistive path from any point to bump]. Custom probes can be added for voltage probing or to create current sinks. In addition, specific reduction techniques are provided for advanced technology nodes to produce a smaller netlist and yet maintain accuracy.

There are two basic variants of ExtractView.

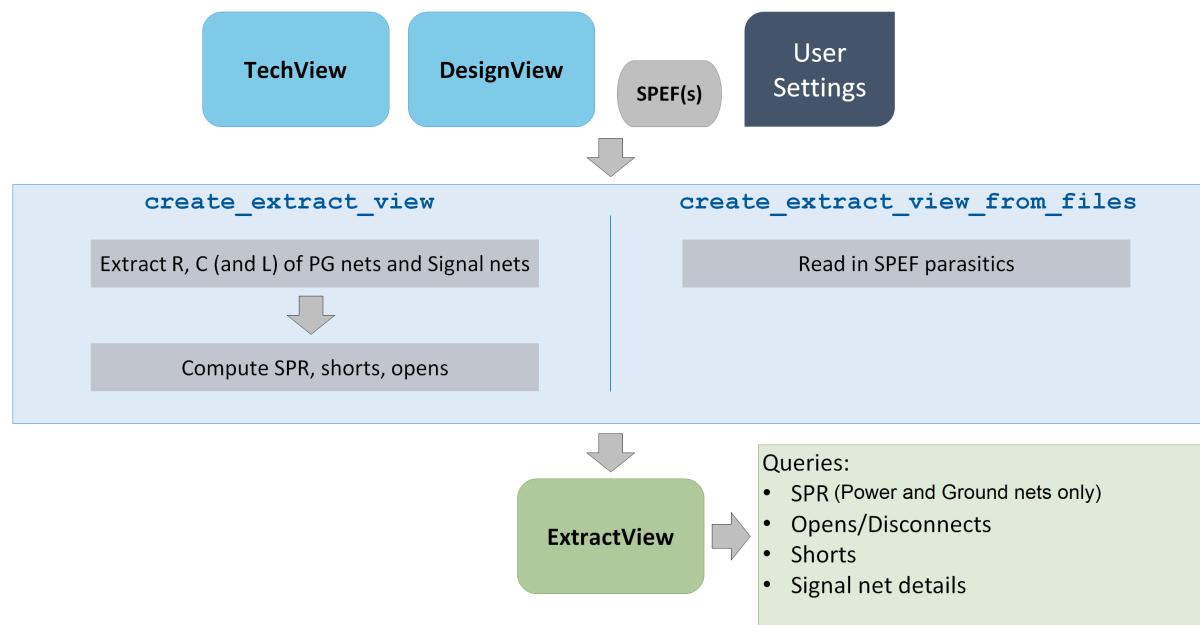
- From the results of performing parasitic extraction directly on the design data.

This ExtractView can store parasitic netlist for PG nets and signal nets. It also stores information for detected net shorts, disconnects, and shortest path resistance (SPR) information. The stored data can be queried from the view.

- From importing a SPEF input file of the signal nets.

By importing from Standard Parasitic Exchange Format (SPEF) file, the parasitic information for the net is stored and post-processed for scenario creation, power estimation and signal electromigration analysis.

Figure 13. ExtractView



Dependencies

An ExtractView has dependencies on the following views when performing extraction of the geometric data.

1. DesignView that contains all the geometric information to be extracted. See [DesignView](#) on page 39.
2. TechView that contains technology information about the process to enable extracting parasitics. See [TechView](#) on page 52.

For more details, use the `help` command at the tool command prompt:

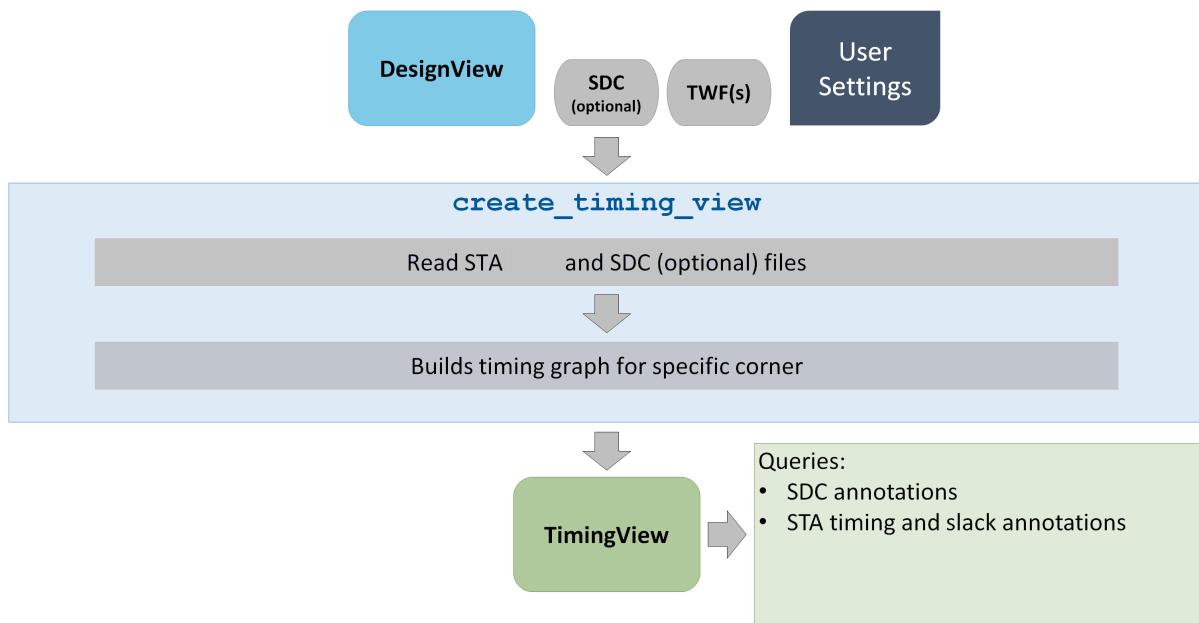
```
help(SeaScapeDB.create_extract_view)
help(SeaScapeDB.create_extract_view_from_files)
```

3.5. TimingView

The TimingView holds all external timing constraints read from SDC files and timing information, slews, timing windows, clocks and slack, read from the output of STA files. It annotates all of the information onto the design as it builds a timing graph of the entire design based on the chosen process corner.

A TimingView is created for each process corner to be analyzed.

Figure 14. TimingView



The following topics describe how to use the `create_timing_view` command:

- [Specifying Arrival Time Delay Per Block on page 43](#)
- [Specifying Different Transition Time Policies for Different Blocks with Same STA on page 44](#)
- [Ignoring Cells Without PG Pins on page 44](#)
- [Dependencies on page 44](#)

Specifying Arrival Time Delay Per Block

You can set a time delay value per block for arrival times. This is useful where the same STA file is reused for multiple instantiations, and a phase shift is present between two successive instantiations. Use the `time_delay` key, as shown in the following example:

```
tw_files = [
    {'instances':[Instance('blockU8'), Instance('blockU12')],
     'file_name': sta_path + 'twf_block.timing', 'time_delay': 2e-9},
    {'instances':[Instance('blockU3')],
```

```
'file_name': sta_path + 'twf_block.timing', 'time_delay': 3e-9}]

tv = db.create_timing_view(..., timing_window_files=tw_files)
```

Specifying Different Transition Time Policies for Different Blocks with Same STA

To set different transition time policies for different blocks that share the same static timing analysis (STA) file, use the `transition_time_policy` key as shown in the following example:

```
tw_files = [
    {'instance_name': '', 'file_name': 'top.timing', 'transition_time_policy': 'max'},
    {'instance_name': 'core0', 'file_name': 'block.timing', 'transition_time_policy': 'min'},
    {'instance_name': 'core1', 'file_name': 'block.timing', 'transition_time_policy': 'avg'}]

tv = db.create_timing_view(..., timing_window_files=tw_files)
```

The `transition_time_policy` key defines how the tool considers transition time values from a range present in the input STA file. Valid values are `min`, `max`, or `avg`.

Ignoring Cells Without PG Pins

To drop cells without PG pins from logic graphs, set the `ignore_cells_with_no_pg_pin` key under `settings['logic_graph']` dict as shown in the following example. The default is `False`.

```
tv = db.create_timing_view(hv, timing_window_files=tw_files, tag = 'tv',
    settings = {'logic_graph': {'ignore_cells_with_no_pg_pin' : True}},
    options=options)
```

Dependencies

The TimingView has dependencies on the following view:

DesignView that contains the design connectivity netlist and also pointers to specific library corners and LibertyViews. See [DesignView](#) on page 39.

For more details, use the `help` command at the tool command prompt:

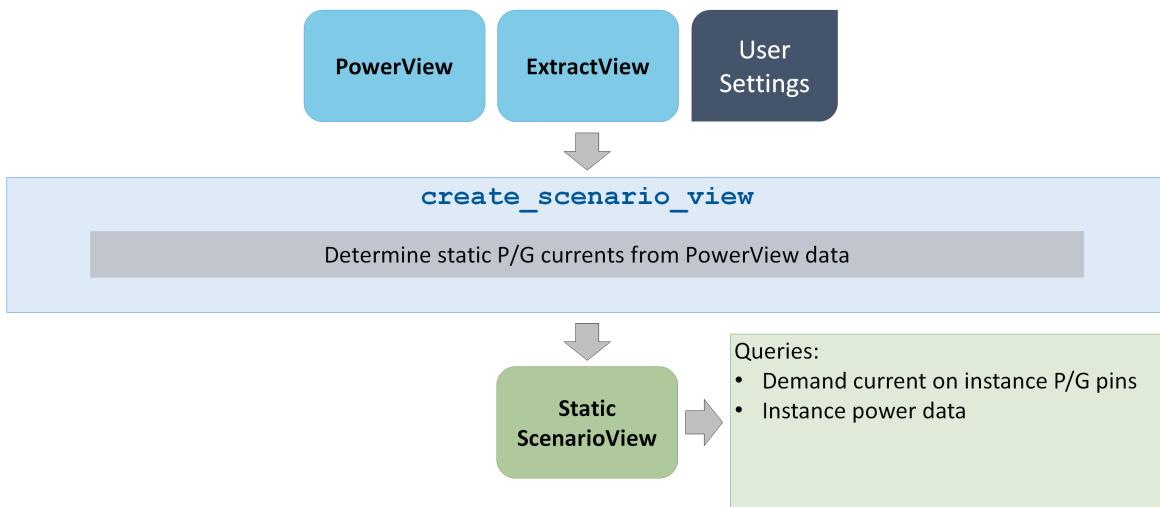
```
help(SeaScapeDB.create_timing_view)
```

3.6. ScenarioView

The ScenarioView is used to determine instance demand currents for specific conditions (such as design mode, PVT corner, specific simulation vectors). This defines the behavior of all the instances in the design and how leaf cells switch.

The calculations made during the view creation determine the design behavior. The queries to instance-specific stored data help you to debug IR drop and electromigration issues. There are two types of scenarios:

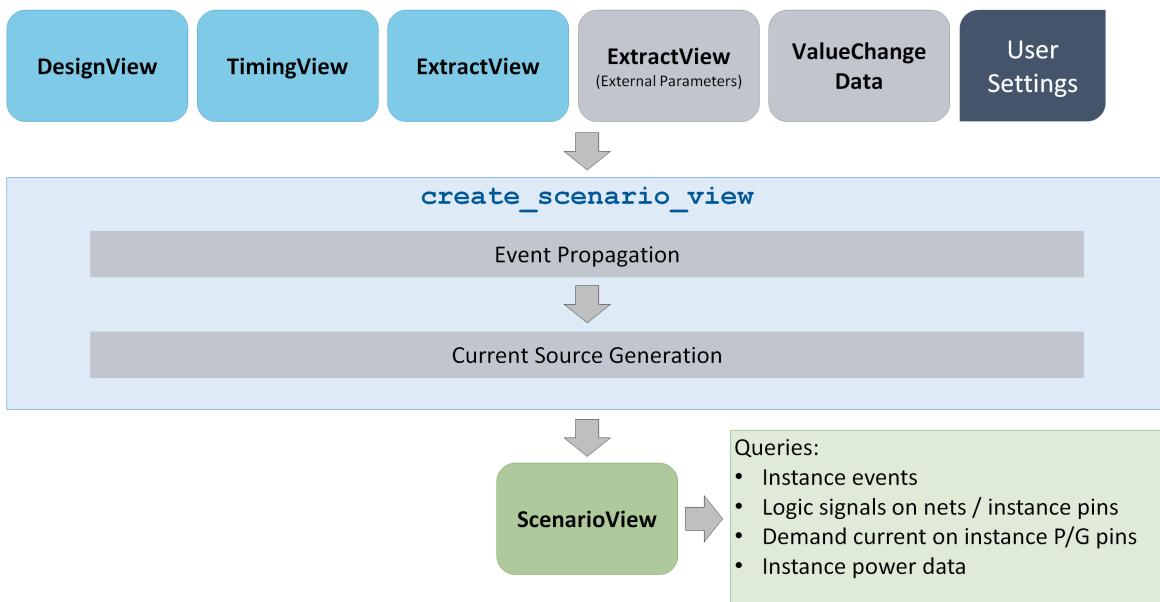
1. Static Scenario where a static current is computed on each instance's PG pin using data from a PowerView.

Figure 15. Static ScenarioView

- 2.** Dynamic Scenario where each instance's PG pin demand current is calculated based on the instance events.

A RedHawk-SC dynamic scenario can be of two types. The no-propagation vectorless scenario is faster and less computationally-intensive.

The logic-propagation scenario uses event propagation with or without power constraints. This is similar in concept to a gate-level simulator with calculated delays. At each flop or macro output pin, activity is determined by random choice given a user-specified activity level. Events propagate through combinational logic using cell delay (or STA timing windows) and logic function information. This makes the activity more realistic and takes into account logical and temporal correlation. Where VCD/FSDB is provided (RTL or gate-level), the vector information is applied, with event propagation filling any gaps.

Figure 16. Dynamic Logic Propagation ScenarioView

Dependencies

A ScenarioView has dependencies on the following views.

1. ExtractViews that contain signal net loading information. The external SPEF takes precedence.
2. TimingView that contains the process corner information, pointers to the appropriate electrical libraries, and timing constraints and timing information.

For more information about how to create static and dynamic ScenarioViews, see:

- [Generating DC Current per PG Pin from Instance Power](#) on page 125
- [Creating No-Propagation Vectorless \(NPV\) Scenarios](#) on page 145
- [Creating Logic Propagation Scenarios](#) on page 163

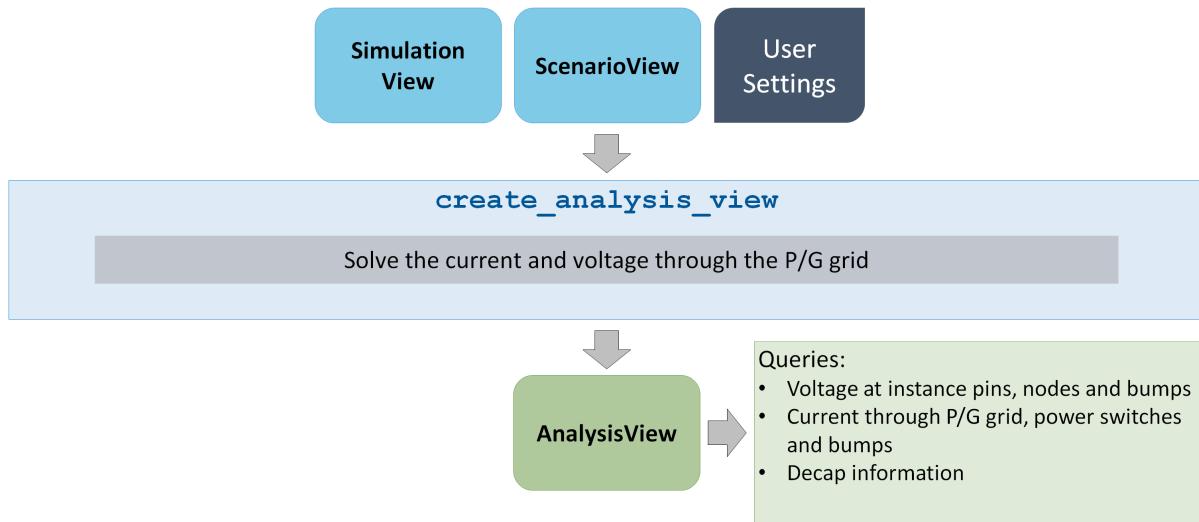
For more details, use the `help` command at the tool command prompt:

```
help(SeaScapeDB.create_no_propagation_scenario_view)
help(SeaScapeDB.create_scenario_view)
```

3.7. AnalysisView

The AnalysisView contains the results of solving the matrices based on the RC (L) of the power grid networks, optionally including a package model, input from the SimulationView, and the current demand of all the leaf cells coming from the ScenarioView. All the IR drop information about the design is available from this view, as well as debug capability to understand the cause of the symptoms, where current is flowing, and how the voltage degrades across the design. Finally, all the currents flowing in the grid are available for post-processing to find ElectroMigration issues by using the `create_electromigration_view` command (see [ElectromigrationView](#) on page 47).

Figure 17. AnalysisView



Dependencies

An AnalysisView has dependencies on the following views.

1. SimulationView (that contains parasitics prepared for a matrix solve, where the bumps/pads are located with optional package/board information), see [SimulationView](#) on page 56.
2. ScenarioViews (that contain the instance currents that will pull currents through the power grid), see [ScenarioView](#) on page 44.

For more information about how to create AnalysisViews for different analysis types, see:

- [Computing Static Voltage Drop for PG Pins](#) on page 126
- [Performing Dynamic Voltage Drop Analysis](#) on page 210

For more details, use the `help` command at the tool command prompt:

```
help (SeaScapeDB.create_analysis_view)
```

3.8. AnalysisComboView

An AnalysisComboView is created as the final step in the RedHawk-SC rampup analysis flow. The view is used to study the impact of rampup current when combined with other sources of activity in an SoC, such as always-on blocks.

AnalysisComboView creation is based on the principle of superposition of multiple rampup AnalysisViews on a base AnalysisView. For each rampup AnalysisView, you specify the time at which current values are passed to the base AnalysisView. This enables you to generate IR drop, heatmaps, and voltages statistics for rampup analysis.

For more information about how to create and use an AnalysisComboView for rampup analysis, see [AnalysisComboView](#) on page 301.

Dependencies

An AnalysisComboView has dependencies on the following views.

1. AnalysisViews (the base AnalysisView object and a list of rampup AnalysisView objects that are superposed on the base AnalysisView). See [AnalysisComboView](#) on page 301.
2. ScenarioView of any type with vectorless or vector-based activity of always-on instances and macros.

For more details, use the `help` command at the tool command prompt:

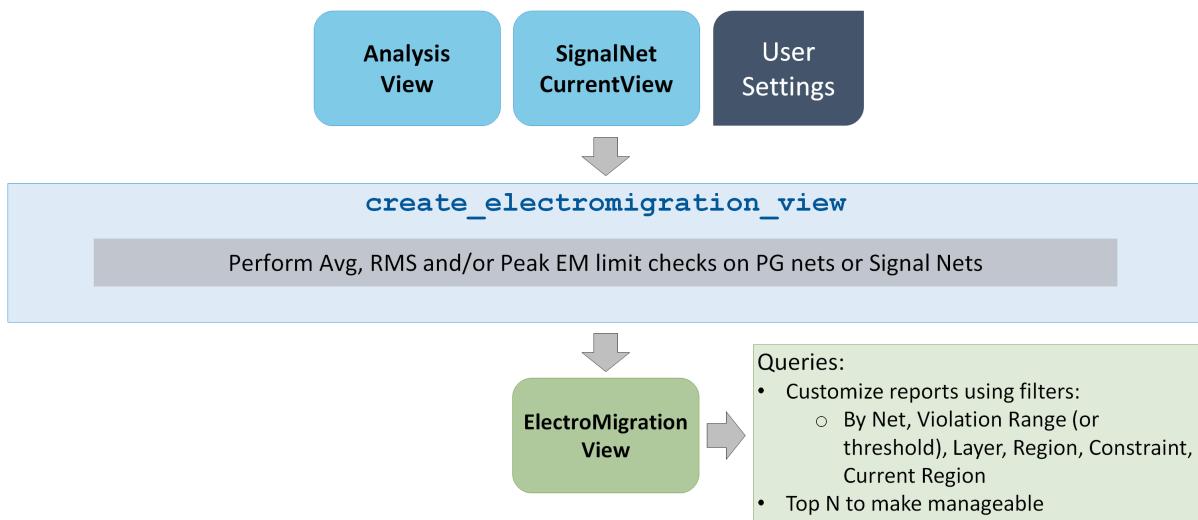
```
help (SeaScapeDB.create_analysis_combo_view)
```

3.9. ElectromigrationView

The ElectromigrationView stores the results of electromigration analysis based on the currents through power and ground nets and signal nets.

For power and ground (PG) net DC electromigration check, the current values from a static AnalysisView are used. For PG root mean square (RMS) and peak checks, currents from a dynamic AnalysisView are used. For signal net electromigration check, the current values are obtained from the SignalNetCurrentView.

These are then compared with the DC, RMS, or peak limits that are loaded from the TechView that stores technology file data.

Figure 18. ElectromigrationView

For more information about how to perform electromigration analysis, see:

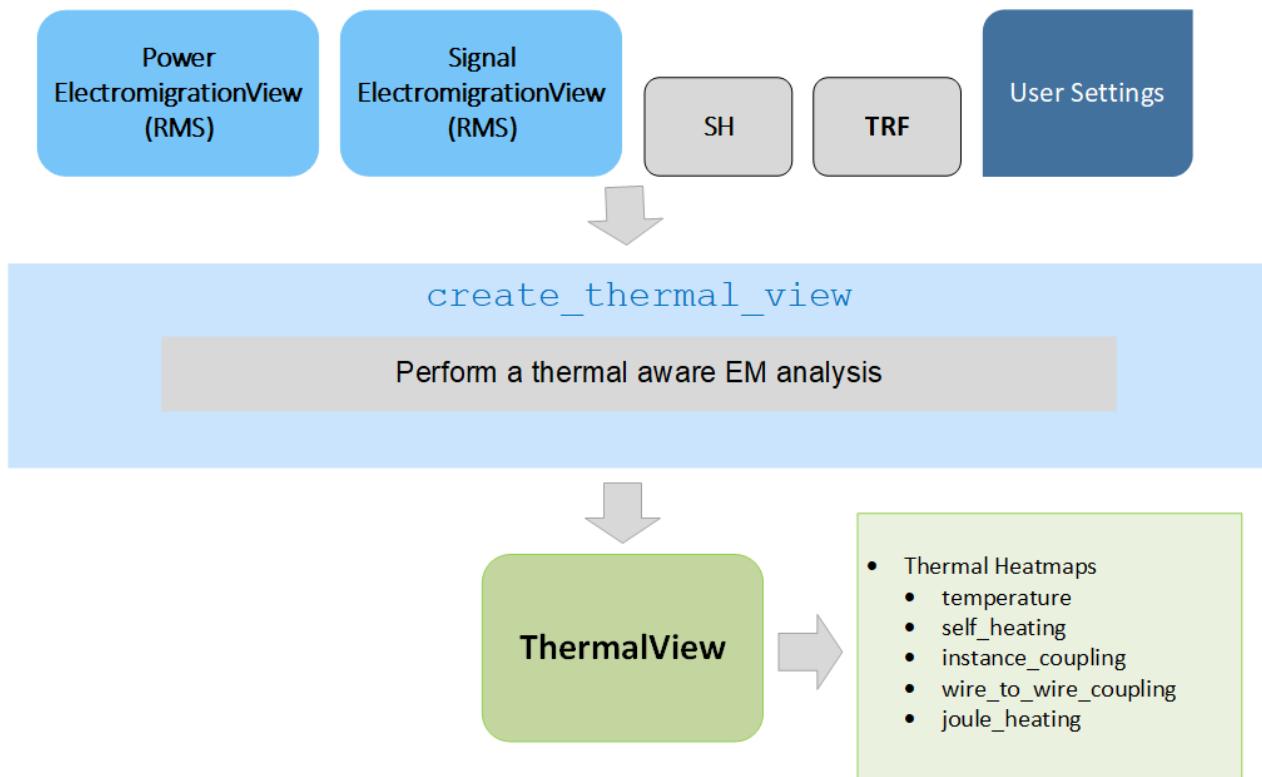
- [Static Electromigration Analysis](#) on page 138
- [Dynamic PG Electromigration Analysis](#) on page 229
- [Signal Electromigration Analysis](#) on page 263

For more details, use the `help` command at the tool command prompt:

```
help(SeaScapeDB.create_electromigration_view)
```

3.10. ThermalView

The ThermalView stores the results of SelfHeat analysis based on the temperature details of instances, power and ground edges, or signal edges.



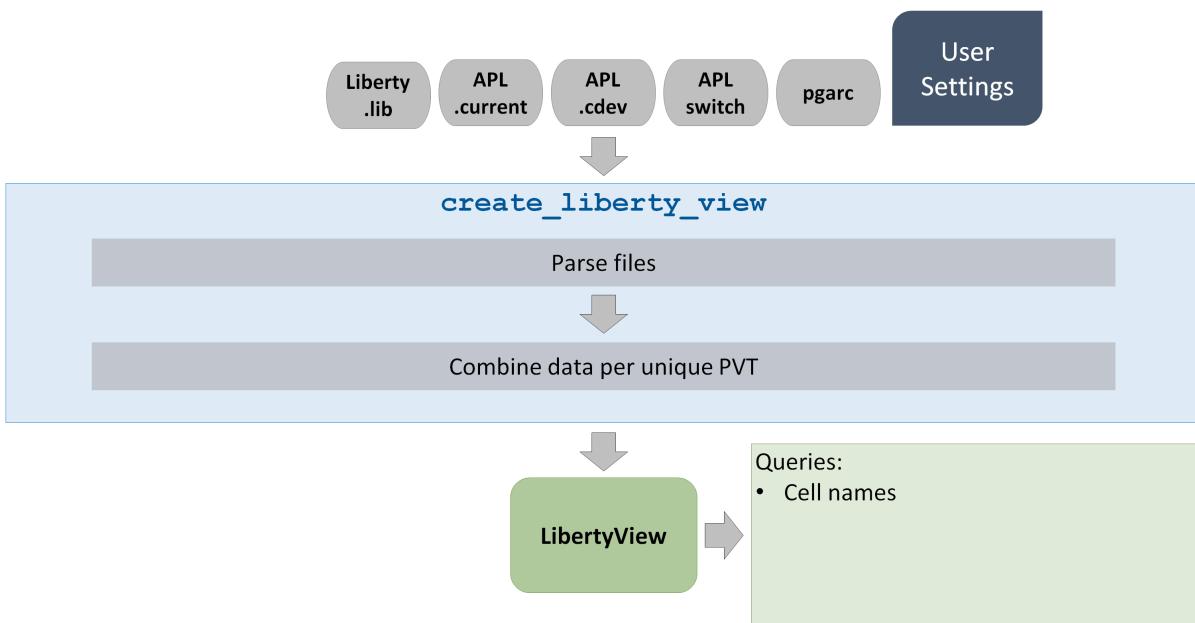
For more information about how to perform thermal analysis, see [SelfHeat Analysis](#) on page 391.

For more details, use the `help` command at the tool command prompt:

```
help(SeaScapeDB.create_thermal_view)
```

3.11. LibertyView

The LibertyView stores all the electrical and logic models of the primitive cells from both Liberty and APL files.

Figure 19. LibertyView

The data stored in LibertyView includes the following:

- Timing arcs
- Nonlinear delay model (NLDM) tables used to calculate delays and transition times (can also store composite current source (CCS) timing models)
- Nonlinear power model (NLPM) energy
- Leakage power or current
- CCS power: power and ground pin current waveform for specific event
- CCS timing: for signal electromigration analysis
- Ansys Power Library (APL): power and ground pin current waveforms and decap models
- APLSW or Liberty: Power gate models
- Modes
- Logic function
- Related ground pin for each power pin

When creating a LibertyView, it is recommended that for PVT storage, for a given P, all V & T variants are read in together. For example, when reading for the 'slow', 'typical', or 'fast' corner, all voltage and temperature variants are all read during the `create_liberty_view` run for that corner. Later, when calculating currents or power, you set the actual desired voltage and temperature, as all the necessary variants are available to enable any voltage and temperature interpolation needed at runtime.

The following topics describe how to use the `create_liberty_view` command:

- [Loading LDO Models](#) on page 51
- [Overriding Liberty Attributes](#) on page 51
- [Scaling Cell and Pin Based Leakage Power](#) on page 51
- [Dependencies](#) on page 51

Loading LDO Models

LibertyView supports static models of on-chip low drop-out (LDO) regulator cells. To generate the LDO models, the APLDO utility is used.

To input LDO models, use the `apl_ldo_files` argument with the `create_liberty_view` command. For example,

```
ldo_model_files = ['cell1.mdl', 'cell2.mdl']
lv = db.create_liberty_view(..., apl_ldo_files=ldo_model_files)
```

An LDO instance is used as a resistor during simulation and is marked with the `is_ldo_cell` attribute.

Overriding Liberty Attributes

You can override the Liberty cell-level `is_decap_cell` attribute value. To do so, use the `is_decap_cell` key under the `settings` argument of the `create_liberty_view` command. For example,

```
lv_settings = {'override_attributes' : [
    'cell_pattern' : ['ICCCAP128*'],
    'cell_attributes' : {'is_decap_cell' : True}, ]
lv = db.create_liberty_view(lib_files, settings=lv_settings, options=options,
tag='lv')
```

Valid value of the `is_decap_cell` key is `True` or `False`.

Scaling Cell and Pin Based Leakage Power

During LibertyView creation, you can scale leakage power numbers from the input Liberty file. To enable per cell and per power pin leakage power scaling, use the `leakage_scale_factor` key under `cell_values` and `pin_values` in the `settings` dict of the `create_liberty_view` command. For example,

```
lv_settings = {
    'cell_values' : [ {
        'patterns' : 'cellA1B2',
        'leakage_scale_factor' : 1.34}, ],
    'pin_values': [
        {'cell_pattern':'cellA3B4',
         'pin_pattern':'vdd1' ,
         'leakage_scale_factor':0.87},
        {'cell_pattern':'cellA3B4',
         'pin_pattern':'vdd2' ,
         'leakage_scale_factor':0.2}]}
lv = db.create_liberty_view(..., settings = lv_settings, ...)
```

Dependencies

A LibertyView has no dependencies on other views.

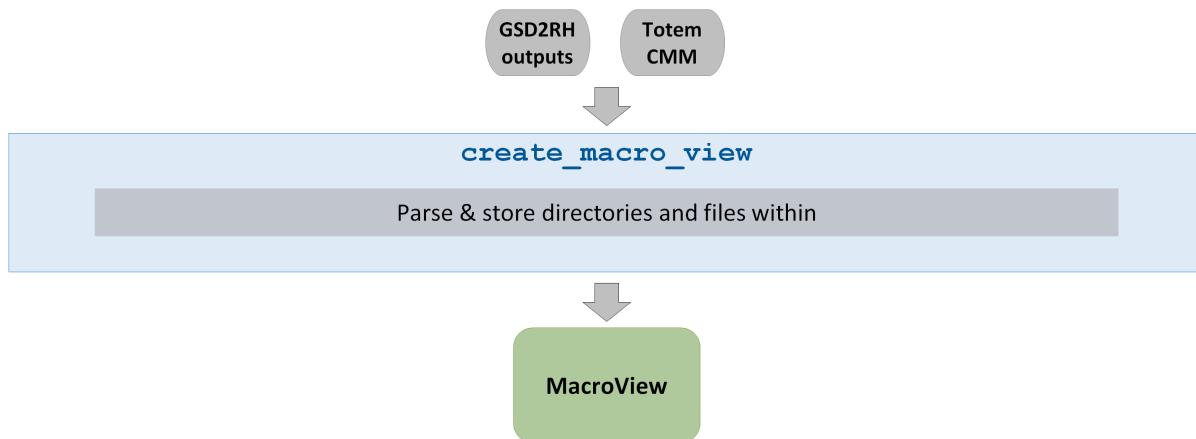
For more details, use the `help` command at the tool command prompt:

```
help(SeaScapeDB.create_liberty_view)
```

3.12. MacroView

The MacroView stores the physical models of macros and IPs generated using either GDS2RH or output from Totem as CMM models, MMX, and cell views. When creating the MacroView, you need to provide a list of directories containing the model data and pick out the relevant subset of needed files from the directories, `<>_adsgds.def`, `<>_adsgds.lef`, and `<>_adsgds.pratio`, for example, from the GDS2RH directories.

Figure 20. MacroView



Dependencies

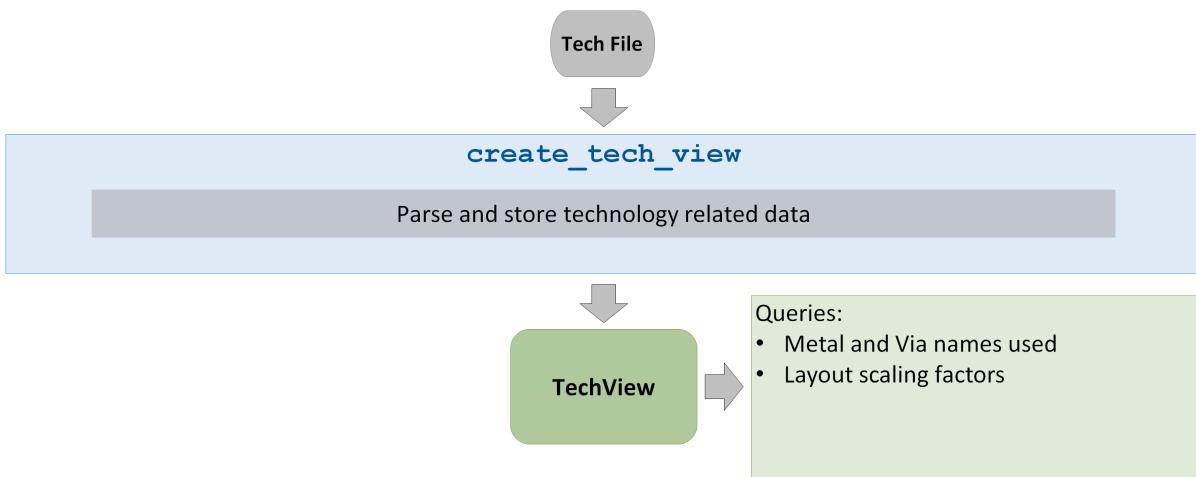
A MacroView has no dependencies on other views.

For more details, use the `help` command at the tool command prompt:

```
help(SeaScapeDB.create_macro_view)
```

3.13. TechView

The TechView is a holding view storing the technology information for a specific extraction corner for subsequent RC(L) extraction as well as the ElectroMigration rules (see). The most common input file is the Apache technology file. However, ITF format and iRCX format are also supported.

Figure 21. TechView

In rare cases, a more accurate capacitance extraction might be needed. You can use the RedHawk-SC tool to precharacterize the information for detailed capacitance extraction and save into an internal format, and then load the generated file with that format. Contact a RedHawk-SC specialist AE for this flow.

Dependencies

A TechView has no dependencies on other views.

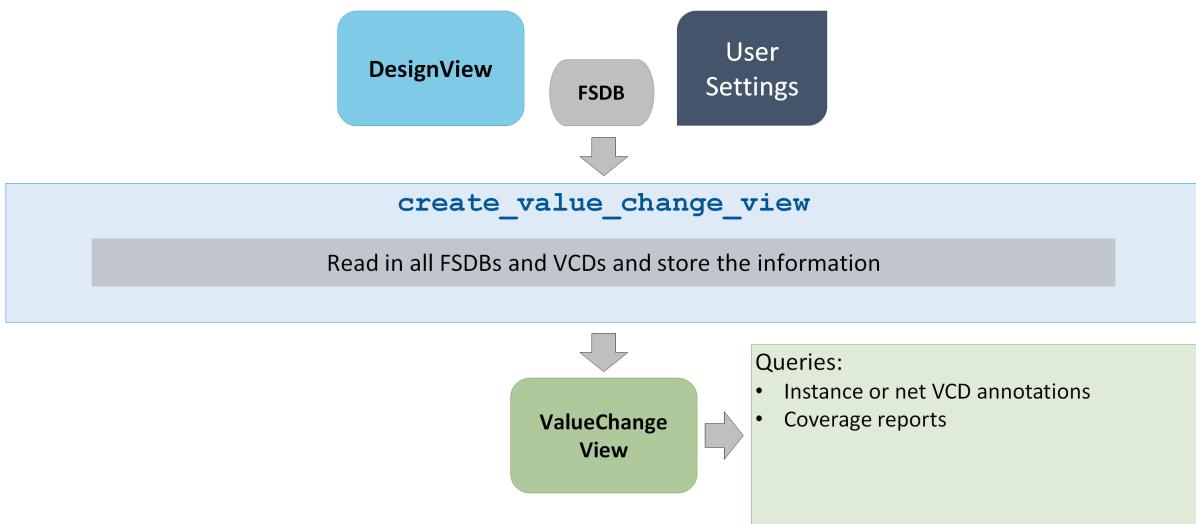
For more details, use the `help` command at the tool command prompt:

```
help(SeaScapeDB.create_tech_view)
```

3.14. ValueChangeView

The ValueChangeView imports all the FSDB/VCD files needed for the design. It provides a very flexible storage of the vectors with the ability to store multiple time slices per FSDB, to be used for multiple simulations or different time slices for different instances in the design that share the same FSDB/VCD file input.

There are also RTL mapping options for user-defined mapping methods from RTL to the gate-level instances in the design.

Figure 22. ValueChangeView

Dependencies

A `ValueChangeView` has dependencies on the following view:

`DesignView` that contains the design connectivity netlist. See [DesignView](#) on page 39.

For more information about `ValueChangeView`, see [ValueChangeView](#) on page 238.

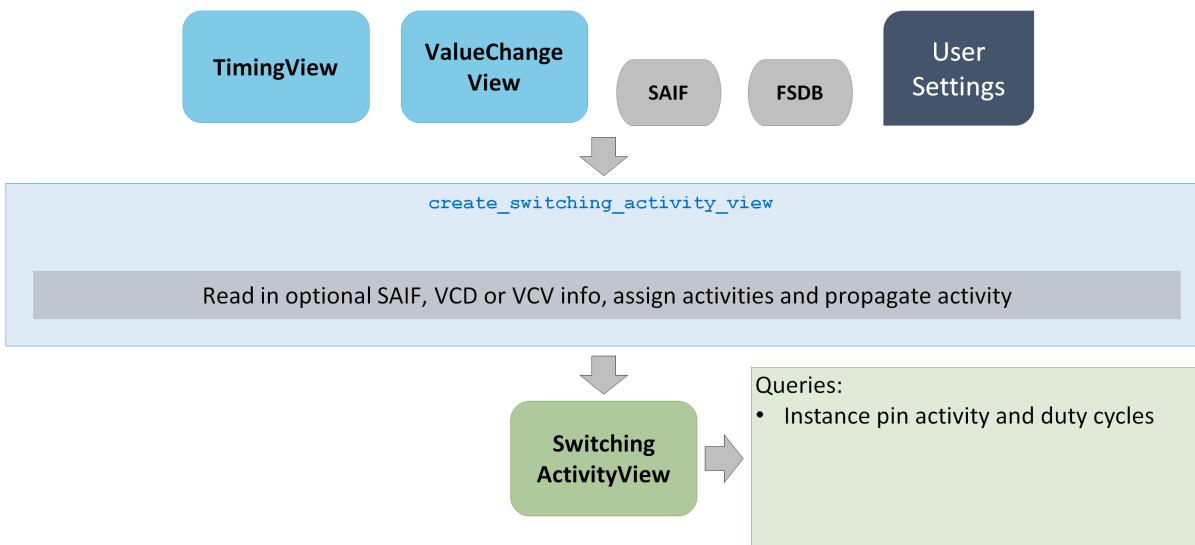
For more details, use the `help` command at the tool command prompt:

```
help(SeaScapeDB.create_value_change_view)
```

3.15. SwitchingActivityView

The `SwitchingActivityView` assigns user-defined toggle rates and duty cycles and, optionally, propagates activity through the combinational logic between register outputs. Inputs to control activities can be imported from gate-level SAIF, FSDB/VCD, or global design and block-specific settings.

The primary use of the `SwitchingActivityView` is to feed into static IR drop and DC electromigration analysis. However, the toggle rates can also be input to vectorless dynamic simulation as probabilities of switching rather than the `toggle_rate` settings of the `create_scenario_view` command.

Figure 23. SwitchingActivityView

Dependencies

A `SwitchingActivityView` has dependencies on the following views:

1. `TimingView` that contains the process corner information and clock data information.
2. `ValueChangeView` that contains imported `FSDB/VCD` vectors.

For more information about how to set switching activity, see [Setting Switching Activity](#) on page 105.

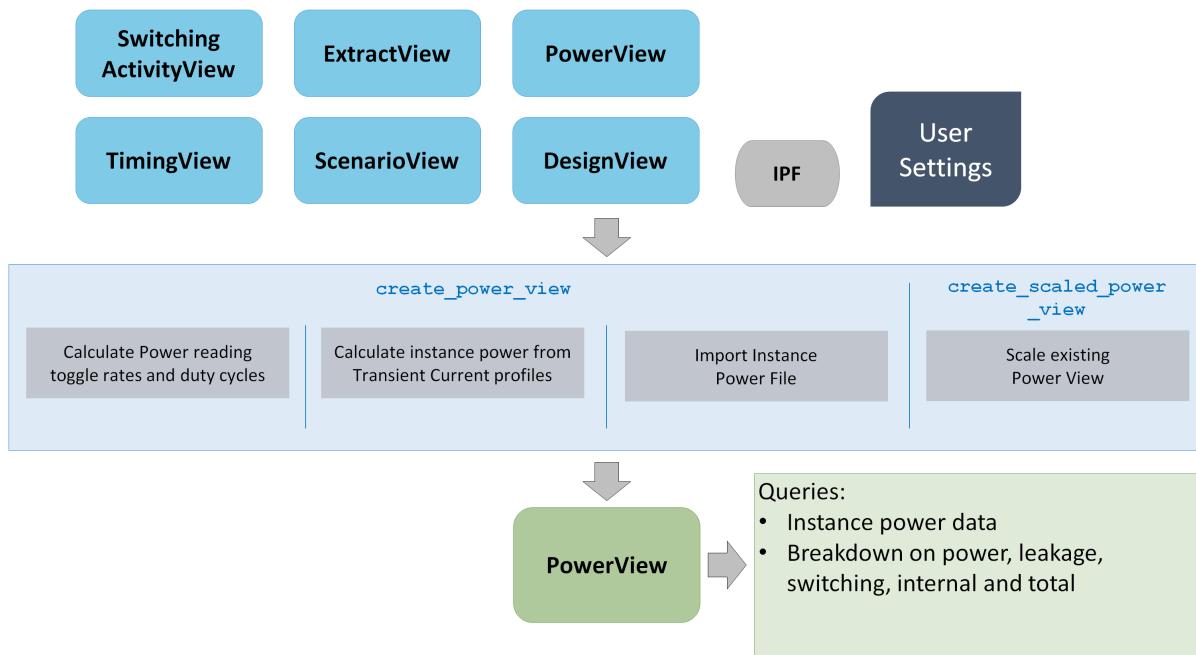
For more details, use the `help` command at the tool command prompt:

```
help(SeaScapeDB.create_switching_activity_view)
```

3.16. PowerView

The `PowerView` stores the power for each instance of the design. There are several methods to create a `PowerView` depending on the source of the data.

- It can be used to import and store a user-defined instance power file (.ipf file format).
- It can be used to represent the power from an existing `ScenarioView`.
- It can be used to calculate the power of instances using the toggle rates and duty cycles coming from a `SwitchingActivityView`.
- It can be used to scale an existing `PowerView` using toggle rates or target power settings and controls for scaling clocks or data.

Figure 24. PowerView

Dependencies

A PowerView has varying dependencies on the following views (based on which previously-described source of data is used).

1. DesignView (that contains the design connectivity netlist).
2. SwitchingActivityView (that contains the toggle rates and duty cycles).
3. ExtractViews (that contains the signal net load information).
4. ScenarioView (that contains instance currents).
5. PowerView (that contains instance power).

For more information about how to compute and scale power values, see:

- [Computing Power](#) on page 112
- [Scaling Power](#) on page 120

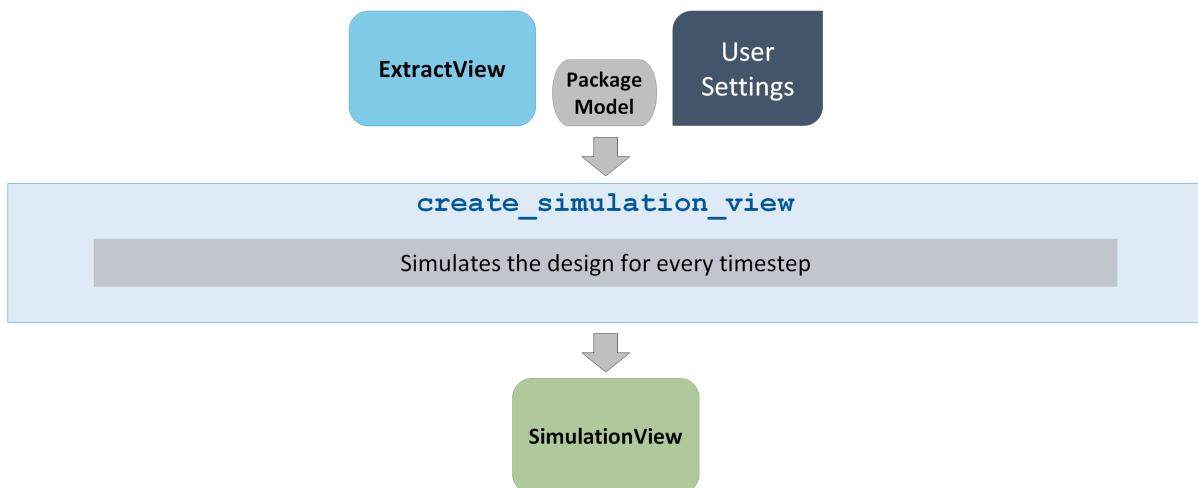
For more details, use the `help` command at the tool command prompt:

```
help(SeaScapeDB.create_power_view)
help(SeaScapeDB.create_scaled_power_view)
```

3.17. SimulationView

The SimulationView is another holding view which creates the distributed data needed for solving the power and ground or signals network systems. For PG rail analysis, it can read in the package (and board) models or apply lumped RLC values.

SimulationView also has settings to enable die-model creation (Chip Power Model, CPM) as well as being able to dump out a Spice netlist of the power and ground network.

Figure 25. SimulationView

Dependencies

A **SimulationView** has dependencies on the following views:

ExtractView that contains the power and ground extracted network or the signals extracted networks. See [ExtractView](#) on page 42.

For more details, use the `help` command at the tool command prompt:

```
help(SeaScapeDB.create_simulation_view)
```


4: Early Grid Analysis

The RedHawk-SC tool can identify problems in PG structures early in the design cycle, as soon as the early power-grid is ready.

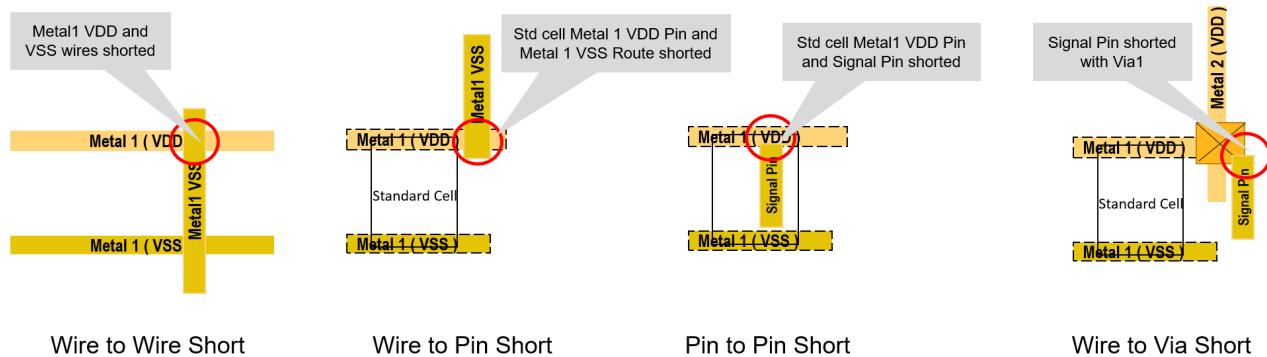
By default, the tool checks for shorts, disconnected instances and wires in ExtractView. You can view the heatmaps in the GUI and generate text reports. In addition to the default checks, you can also perform build quality metric (BQM), shortest path resistance (SPR), effective resistance (Reff), and missing via checks.

The RedHawk-SC tool supports the following methods and checks for early grid analysis:

- [Reporting Shorts](#) on page 59
- [Reporting Disconnected Instances and Wires](#) on page 62
- [Performing Build Quality Metric \(BQM\) Analysis](#) on page 65
- [Tracing Shortest Path Resistance \(SPR\)](#) on page 74
- [Computing Effective Resistance](#) on page 84
- [Reporting Missing Vias](#) on page 90
- [Creating Statistical Heatmaps for Multiple Parameters](#) on page 92

4.1. Reporting Shorts

A short is created when two different nets touch each other. The following figure shows the different short types.



By default, the tool checks for shorts in ExtractView between:

- Two different power nets
- Two different ground nets
- A power net and a ground net
- A power or ground net, and a signal net

The following topics describe how to report and view shorts.

4.1.1. Reporting Shorts Summary

The following example shows how to use the `get_shorts_summary` command to generate the shorts summary report:

```
ev.get_shorts_summary()

{'pg_net_shorts': {('Net('VSS'), Net('core3/VDD_INT'), Layer('metal1')): 39},
 'total_num_shorts': 135, 'total_num_shorts_between_pg_nets': 39}
```

The summary informs you that the total number of shorts in the design of any type are 135 and a total of 39 shorts are found on metal1, between net VSS and core3/VDD_INT.

For more details about the `get_shorts_summary` command, use the `help` command at the tool command prompt:

```
help(ExtractView.get_shorts_summary)
```

4.1.2. Reporting Shorts Details

The following example shows how to use the `write_short_reports` command to generate the detailed shorts report:

```
reports_dir = 'reports/grid_robustness_checks/'
data_integrity_reports.write_short_reports
(ev, output_file=reports_dir+'shorted_nodes.rpt')
```

By default, the detailed shorts report has the following directory structure:

`reports/grid_robustness_checks/shorted_nodes.rpt`.

The following example shows the snippet of a detailed shorts report:

#Net_1	Net_2	Location	Layer
VDD	VSS	13.340, 64.000	metal10
VDD	VSS	13.340, 64.000	metal11
VDD	VSS	13.340, 156.000	metal10
VDD	VSS	13.340, 156.000	metal11
VDD	VSS	13.340, 61.600	metal9
VDD	VSS	13.340, 66.900	metal9
VDD	VSS	13.340, 66.900	metal11
VDD	VSS	13.340, 154.000	metal9
...			
VSS	HFSNET_4	1105.600, 708.100	metal10
VSS	ZBUF_2_4	1107.605, 751.800	metal1
VSS	dma_din	1008.000, 984.000	metal10
VSS	_1827_	1073.035, 981.400	metal3
...			

For more details about the `write_short_reports` command, use the `help` command at the tool command prompt:

```
help(data_integrity_reports.write_short_reports)
```

4.1.3. Querying Custom Shorts Data

To report custom shorts data, use the `get_net_shorts` command. You can use the command to report shorts between two domains, in a specific layer or region, such as report only PG shorts.

```
ExtractView.get_net_shorts(from_nets=None, to_nets=None, layers=None,
region=None, pg_short_only=True)
```

Note: The `get_net_shorts` command requires launching additional workers.

The following example shows how to use the `get_net_shorts` command to report shorts:

```
shorts_data = ev.get_net_shorts(pg_short_only=False)
```

You can use `shorts_data` to generate shorts summary and detailed shorts report:

```
shorts_summary = shorts_data.get_summary()
```

`shorts_summary` is a list that shows the number of shorts between different power, ground, or signal nets.

For example,

```
shorts_summary[0]
```

```
(Net('VDD'), Net('core1.regfile_program_memory._9581_'), Layer('metal10'), 1)
```

To generate shorts report from `shorts_data`, use the following command:

```
shorts_data.write_to_file('shorts.rpt')
```

The following example shows the snippet of a detailed shorts report:

# Location	Layer	Net_1	Net_2
13.340, 64.000	metal10	VDD	VSS
13.340, 64.000	metal11	VDD	VSS
13.340, 156.000	metal10	VDD	VSS
13.340, 156.000	metal11	VDD	VSS
13.340, 61.600	metal9	VDD	VSS
13.340, 66.900	metal9	VDD	VSS
13.340, 66.900	metal11	VDD	VSS
13.340, 154.000	metal9	VDD	VSS
1105.600, 708.100	metal10	VSS	HFSNET_4803
1107.605, 751.800	metal1	VSS	ZBUF_2_4136
1008.000, 984.000	metal10	VSS	dma_din[7]
1073.035, 981.400	metal3	VSS	_1827_
...			

For more details about the `get_net_shorts` command, use the `help` command at the tool command prompt:

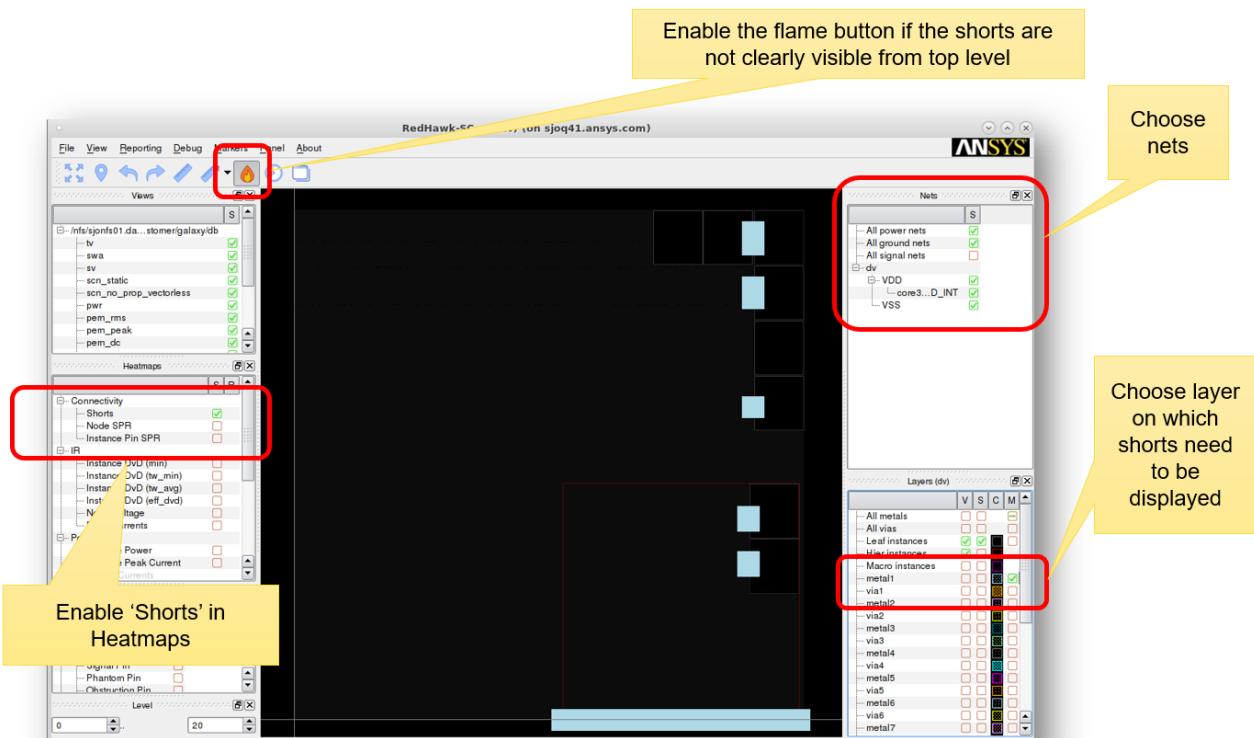
```
help(ExtractView.get_net_shorts)
```

4.1.4. Viewing Shorts

Based on the shorts from the ExtractView, the layout GUI displays a prepopulated shorts heatmap as shown in the following figure. To view these shorts in the GUI, follow these steps:

1. Select **Shorts** under **Heatmaps**.
2. Select the nets under **Nets**.
3. Select the layer under **Layers**, where the shorts are to be displayed.

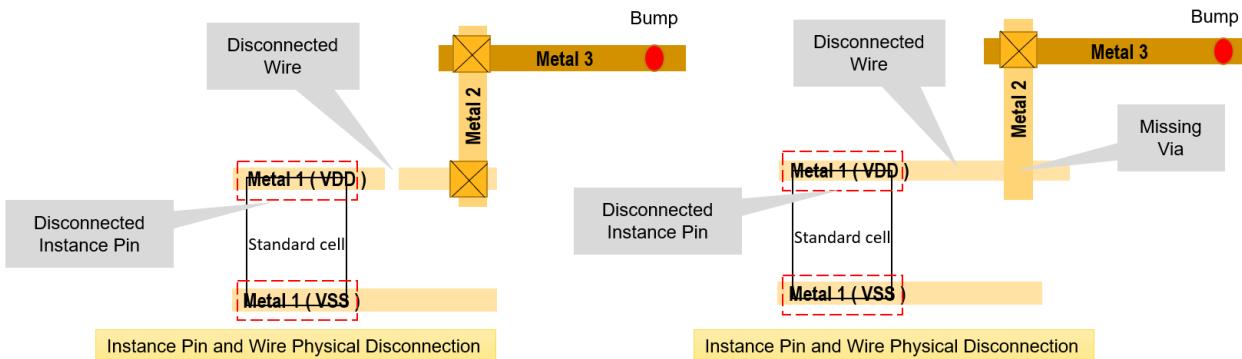
To magnify the shorts view, enable the Flame icon ().



4.2. Reporting Disconnected Instances and Wires

ExtractView checks for and stores logically and physically disconnected instances, and physically disconnected nodes of the design.

A logical disconnect occurs when an instance pin connection to the top level net (such as VDD, VSS in the following figure) is not defined or missing in the design netlist. A physical disconnect occurs when a physical pin or a wire node is disconnected from the bump or pad location as shown in the following figure.



Note: When an instance is logically disconnected, the tool does not check the physical disconnects for this instance.

The following topics describe how to report disconnects of a design.

4.2.1. Reporting Disconnected Pins

To generate a report of instance PG and package pins that are disconnected logically and physically, use the `write_unconnected_pg_pins_report` command:

```
data_integrity_reports.write_unconnected_pg_pins_report
```

The `write_unconnected_pg_pins_report` command is used to create a formatted report of the disconnected instances in the design. The following examples show how to use the command with different arguments.

- When you specify only the required `extract_view` argument, a default report is generated at the following location:

`reports/grid_robustness_checks/unconnected_instance_pins.rpt`.

```
data_integrity_reports.write_unconnected_pg_pins_report(ev)
```

Note: The `extract_view` argument (`ev` in the example) is required. All other arguments are optional.

The generated output file shows the instance, cell, pin, and net names and indicators for you to determine whether the disconnect is physical or logical. An example snippet of the output file is as follows:

#loc_x	loc_y	cell_name	pin	net	logical_disconnect	physical_disconnect	Instance
1215.62	222.6	INV_X4	VDD	VDD	0	1	ZINV1_INST1
1215.62	222.6	INV_X4	VSS	VSS	0	1	ZINV1_INST1
1216.57	222.6	INV_X4	VDD	VDD	0	1	ZINV1_INST2
1216.57	222.6	INV_X4	VSS	VSS	0	1	ZINV1_INST2
1215.62	85.4	INV_X4	VDD	VDD	0	1	ZINV2_INST1
1215.62	85.4	INV_X4	VSS	VSS	0	1	ZINV2_INST1
...							

- When you use the `sort` and `sort_order` arguments, the values in the report are first sorted based on instance names and then net names.

```
data_integrity_reports.write_unconnected_pg_pins_report
(ev, max_lines=None, sort=True, sort_order="ascending",
output_file=reports_dir+'unconnected_instance_pins_sortedAscending.rpt')
```

- When you use the `sort_column` argument, the values in the report are sorted based on the specified value, such as `cell_name`.

```
data_integrity_reports.write_unconnected_pg_pins_report
(ev, max_lines=None, sort=True, sort_order="ascending", sort_column=["cell_name"],
output_file=reports_dir+'unconnected_instance_pins_sort_by_cell_name.rpt')
```

For more details about the `write_unconnected_pg_pins_report` command, use the `help` command at the tool command prompt:

```
help(data_integrity_reports.write_unconnected_pg_pins_report)
```

4.2.2. Querying Disconnected Instances

To query the list of disconnected instances of the design, use the following command:

```
ev.get_disconnected_instances(limit=10)
```

`ev` is the ExtractView to query. Using the `limit` argument returns only the specified number of disconnected instances. In the following example, the `get_disconnected_instances` command returns only ten disconnected instances though the actual number of disconnected instances is much larger.

```
>>> ev.get_disconnected_instances(limit=10)
WARNING<XTR.141> There were a total of 5419 physical disconnected instances but only the first 10 were returned
. Set a higher limit option if you want to see more.
{'logical': [],
'physical': [Instance('xofiller!FILLCELL_X1!x12262600y2744000'),
Instance('xofiller!FILLCELL_X1!x12264500y2744000'),
Instance('xofiller!FILLCELL_X1!x12266400y2744000'),
Instance('xofiller!FILLCELL_X32!x12156200y2758000'),
Instance('xofiller!FILLCELL_X16!x12217000y2758000'),
Instance('xofiller!FILLCELL_X8!x12247400y2758000'),
Instance('xofiller!FILLCELL_X1!x12262600y2758000'),
Instance('xofiller!FILLCELL_X32!x12156200y2744000'),
Instance('xofiller!FILLCELL_X16!x12217000y2744000'),
Instance('xofiller!FILLCELL_X8!x12247400y2744000')]}
```

For more details about the `get_disconnected_instances` command, use the `help` command at the tool command prompt:

```
help(ExtractView.get_disconnected_instances)
```

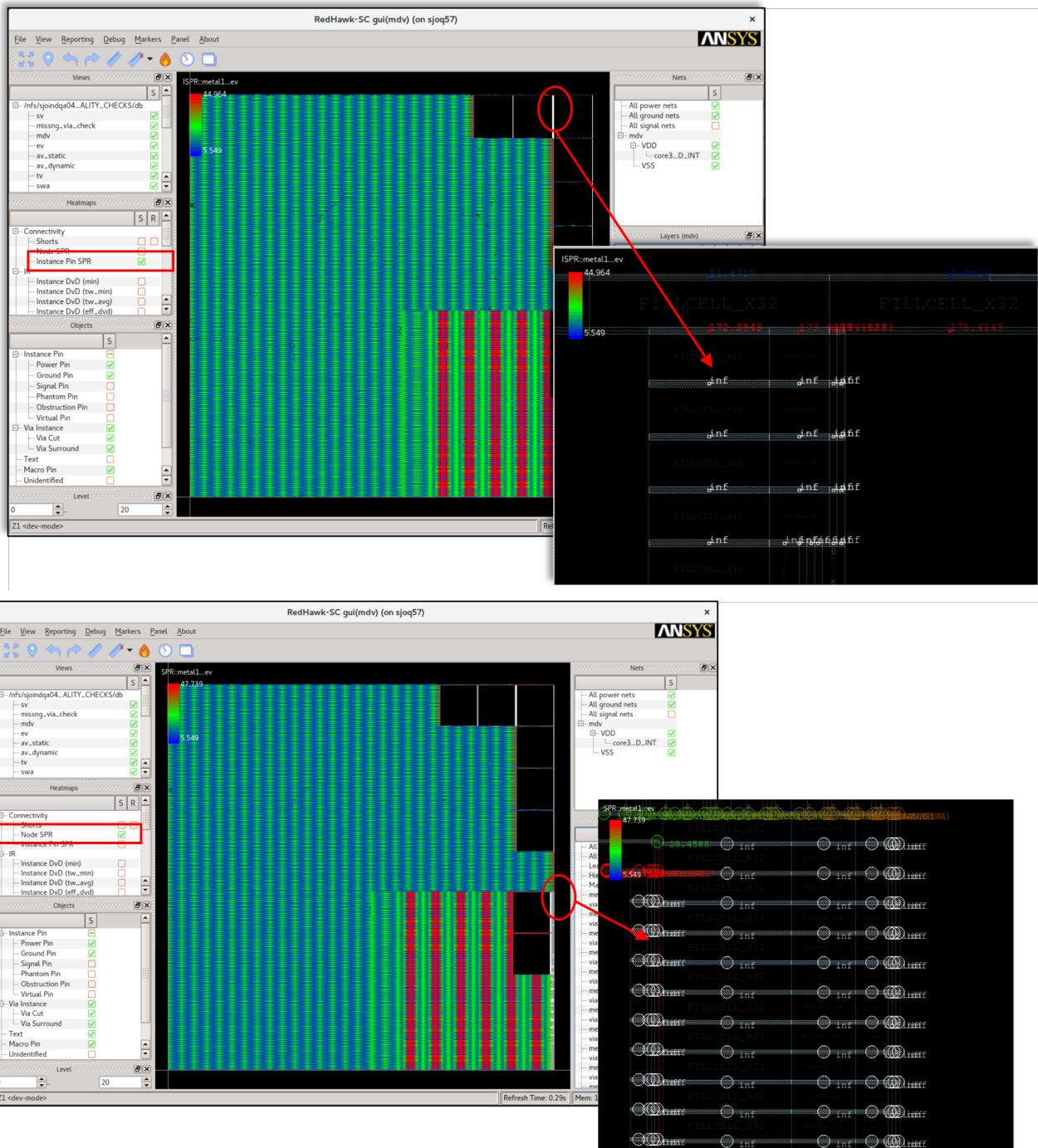
4.2.3. Viewing Disconnects in GUI

You can review instance pin and wire disconnections in GUI using shortest path resistance (SPR) heatmaps. To view these in GUI, appropriately select **Instance Pin SPR** or **Node SPR** under **Heatmaps**.

If the instance pin or any routing is disconnected from the grid, corresponding SPR value is infinity.

[Instance Pin Disconnects](#) shows an instance pin SPR heatmap with disconnected instance pins and [Wire Disconnects](#) shows a node SPR heatmap with disconnected wire nodes.

The disconnects are displayed in white color and represented in text as **inf**.



4.3. Performing Build Quality Metric (BQM) Analysis

Build quality metric (BQM) checks the reliability of a power grid using static simulation by normalizing the voltage drop across each node of the design.

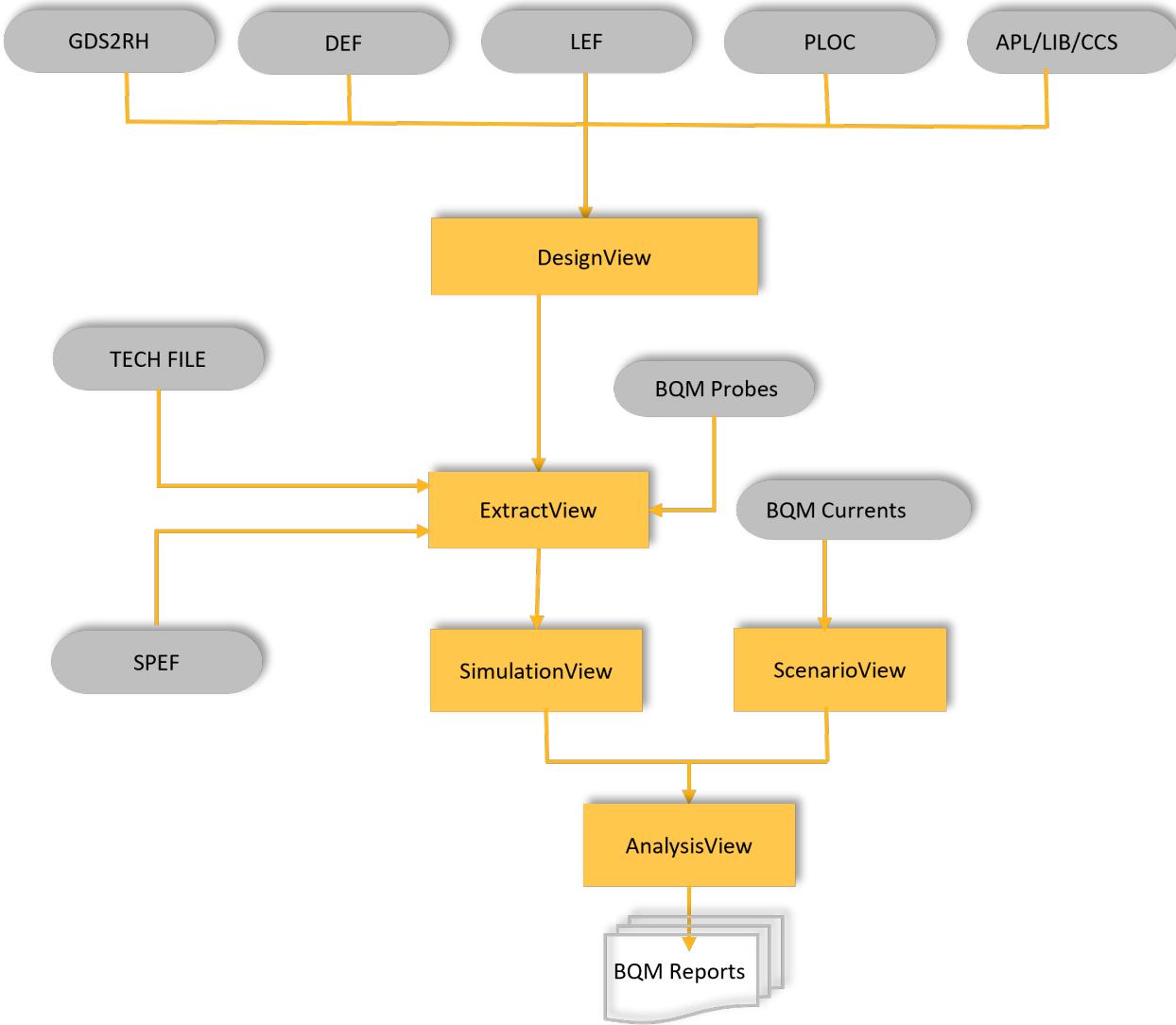
The BQM technique has the following advantages over other such techniques:

- Finds resistive connection faults that cannot be detected using either SPR or effective resistance techniques.

- Finds weaknesses of the entire grid independent of cell placement. As this technique uses current probes, the results are not limited to the lowest metal layers unlike the resistance-based SPR or effective resistance techniques.
- Efficient to analyze even before placement is performed.

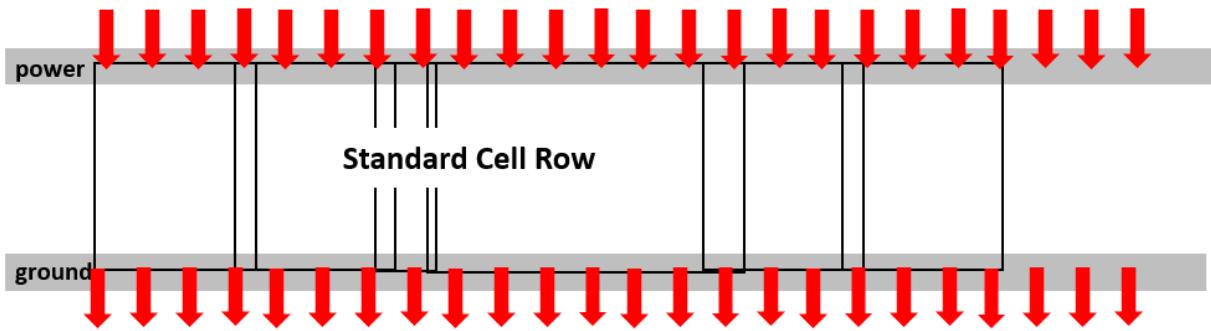
Therefore, it is recommended that you perform BQM checks over [Tracing Shortest Path Resistance \(SPR\)](#) on page 74 and [Computing Effective Resistance](#) on page 84.

The following figure shows the BQM flow. Probes are applied to the design at the ModifiedExtractView stage and the currents are applied at the ScenarioView stage. Static analysis is performed to generate the node and instance heatmaps and reports.



The RedHawk-SC tool performs the following steps for a BQM check:

1. By default, determines the PG segments on the lowest metal layer that connect to the standard cells. You can select a different metal layer.
2. Applies equidistant constant current sources (through probes) on these segments as shown in the following figure:



3. Simulates the static voltage drop.
4. Generates heatmaps to show relative weak areas of the grid.

4.3.1. Generating Probes and Currents

To generate probes and their currents for BQM check, use the `bqm.generate_bqm_current_probe_metal` command:

```
bqm.generate_bqm_current_probe_metal
```

Argument	Description
dv	DesignView (type=BaseView, required=True)
check_layer	Layer on which to place the current probes (type=Layer, required=True)
probe_distance	Distance in microns between each probe on the metal rails found (type=float, default_value=0.2)
hor_ver_ratio	Ratio of horizontal dimension to vertical dimension for which to add probes to the metal (type=float, default_value=10.0)
total_current_for_probes	Total current to distribute to the current probes (type=float, default_value=10.0)
current_per_micron	Current density to distribute to the current probes (type=float, default_value=None)
<p>Note: The current_per_micron argument converts per-micron power specification into equivalent current drawn on the power rail. When you use this argument, BQM results are directly interpreted as voltage.</p>	
add_probes_on_lef_pin	Add probes on the geometries from the LEF along with the DEF routes (type=bool, default_value=False)

The following example shows how to use the `generate_bqm_current_probe_metal` command shown in [BQM flow diagram](#). The command generates equidistant probes separated by a distance of 3µm, and a current source for each probe at the metal1 layer. The probes and their currents are passed to ExtractView

and ScenarioView. The analysis results are stored as node voltage heatmaps. You must review each PG net and each layer to evaluate the relative weaknesses in the grid.

```
#Generate probes at every 3 μm distance
bqm_probes, bqm_currents = bqm.generate_bqm_current_probe_metal
                            (weak_dv,Layer('metall1'),probe_distance=3.0)

#Pass the probes and currents to ExtractView and ScenarioView
ev_bqm_args = dict(
    settings={'probes' : bqm_probes}
    ...)
ev_bqm = db.create_extract_view(weak_dv, nv, **ev_bqm_args)

scn_bqm_args = dict(
    settings={'probe_sources' : bqm_currents}
    ...)
scn_bqm = db.create_scenario_view(design_view=weak_dv, **scn_bqm_args)

sv_bqm = db.create_simulation_view(ev_bqm, **sv_bqm_args)

#Run analysis
av_bqm = db.create_analysis_view(sv_bqm, scn_bqm, **av_bqm_args)
```

In the following example, the `current_per_micron` argument is set to default `None`, and the value specified with the `total_current_for_probes` argument is equally distributed between all the probes.

```
bqm.generate_bqm_current_probe_metal(dv,check_layer,
probe_distance=0.2, hor_ver_ratio=10.0,total_current_for_probes=10.0,
current_per_micron=None, add_probes_on_lef_pin=False)
```

4.3.2. Generating Instance Voltage Drop Heatmaps

By default, BQM analysis results are stored as node voltage heatmaps. To generate heatmaps with corresponding voltage drop across instances, use the `generate_pin_voltage_heatmaps` command:

```
gen_drop_hm.generate_pin_voltage_heatmaps
```

The following example shows how to use the `generate_pin_voltage_heatmaps` command to generate and store BQM reports as shown in [BQM flow diagram](#). Analysis results are stored in UserView. To specify the UserView name, use the `tag` argument with the `create_user_view` command. You can view the heatmaps in GUI.

```
bqm_inst =
gen_drop_hm.generate_pin_voltage_heatmaps(av_bqm,nets=pg_nets,base_level=True)

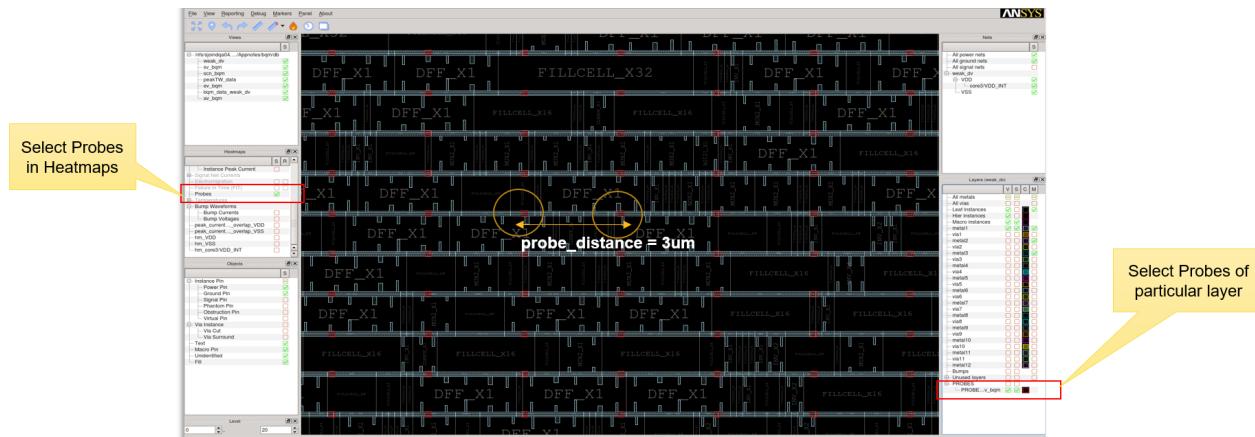
bqm_data = db.create_user_view(tag='bqm_data_weak_dv')
for kk in bqm_inst:
    bqm_data['hm_'+kk.get_name()] = bqm_inst[kk]
    bqm_data['stats_'+kk.get_name()]=
norm.get_heatmap_min_max_median(weak_dv,bqm_inst[kk])
```

4.3.3. Viewing BQM Analysis Results

The following figure shows the BQM probes separated by a distance specified with the `probe_distance` argument of the `generate_bqm_current_probe_metal` command. To view these probes in GUI:

1. Select Probes in Heatmaps.
2. Select the probes of a particular layer under Probes in Layers.

Figure 26. Viewing BQM Probes



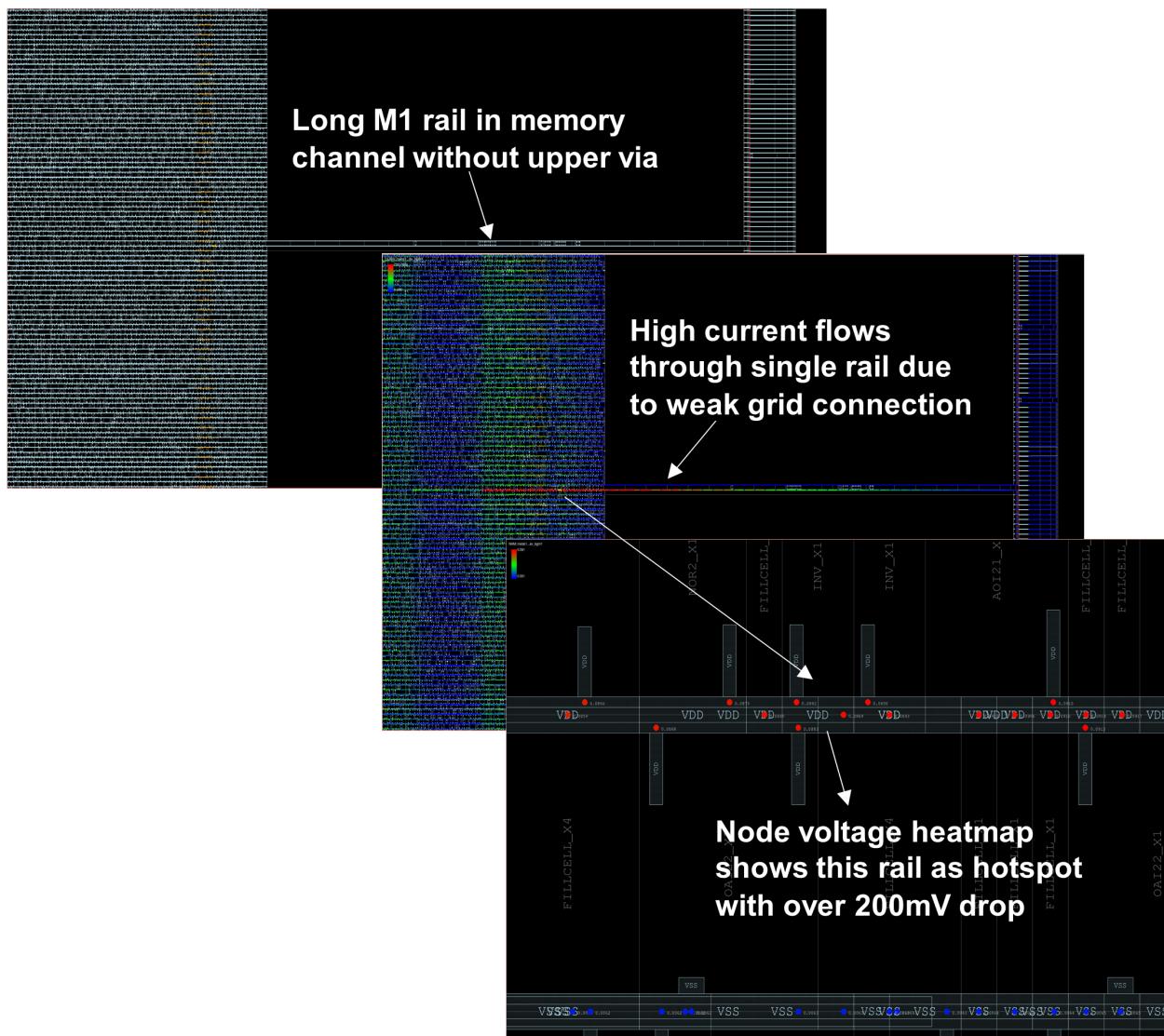
The following figure shows the BQM current values and corresponding probes. In this case, each BQM probe on metal1 consumes a current of 30.7199 μ A.

To view the currents in GUI, select **Edge Currents in Heatmaps**.

Figure 27. Viewing BQM Currents



The following figure shows how to identify a hotspot and perform limit checking in the BQM node voltage heatmap.

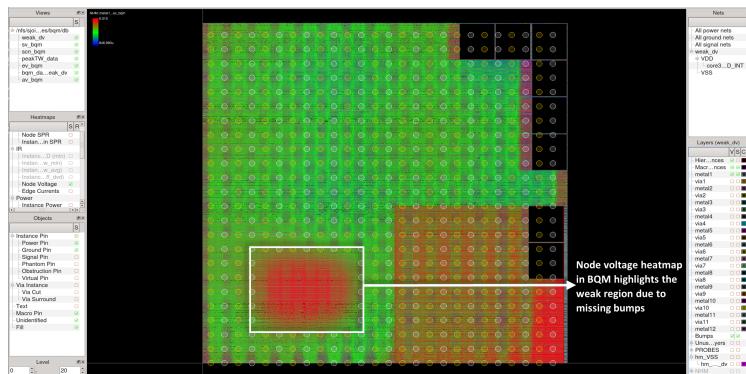
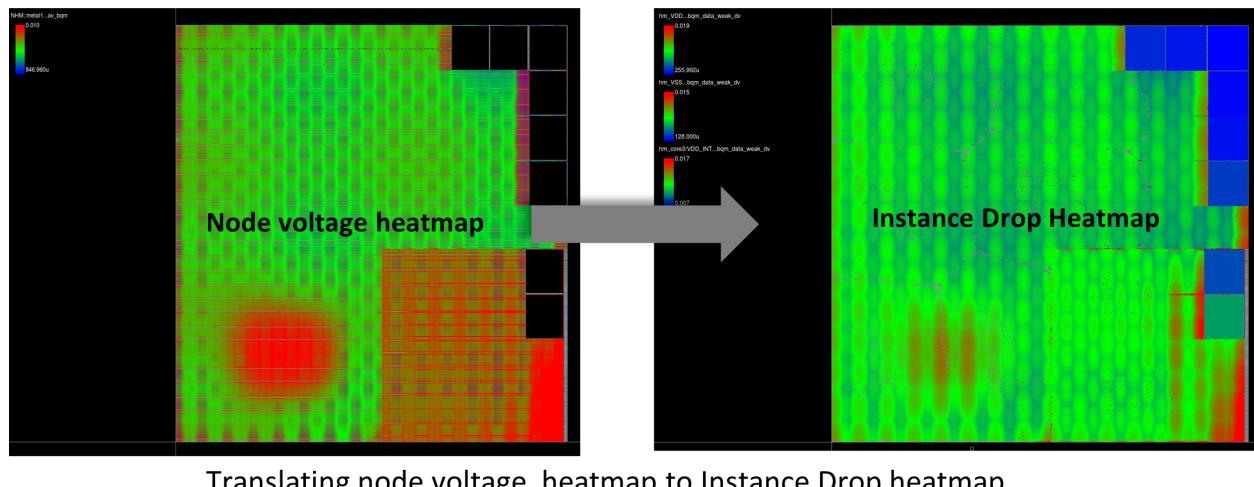
Figure 28. Identifying Hotspots

BQM Instance Voltage Drop Heatmaps

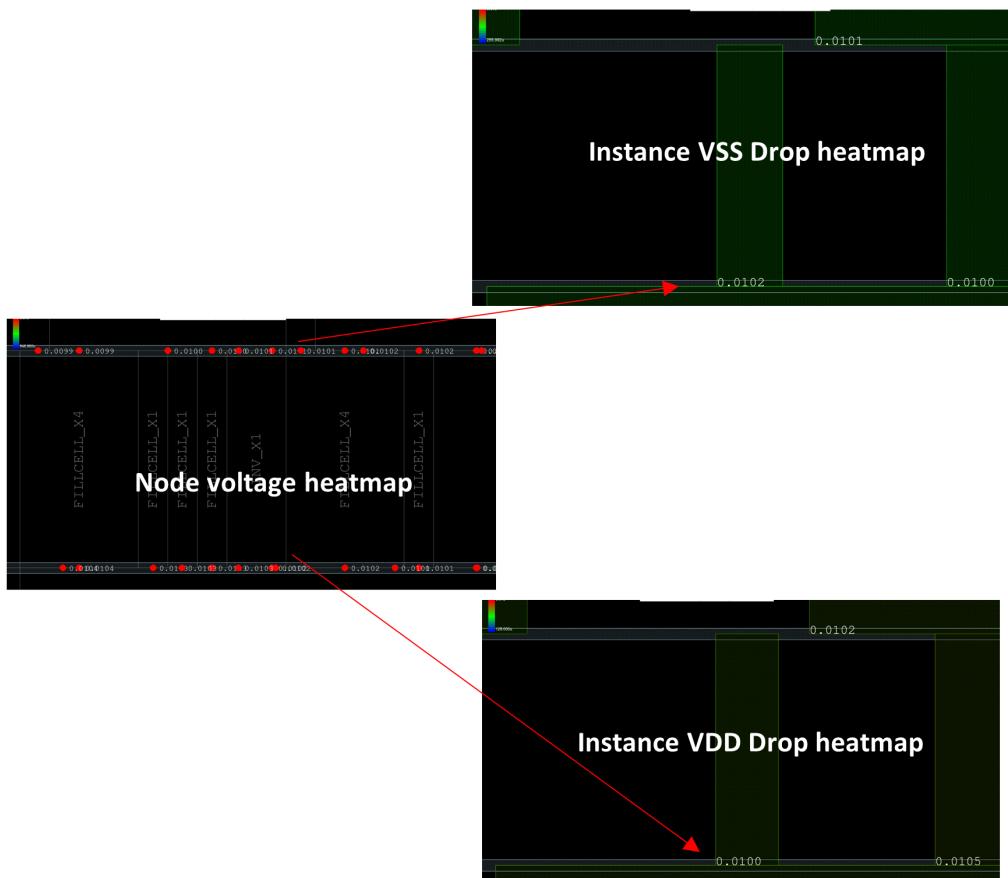
An instance voltage drop heatmap is a representation of the node voltages on the pin nodes of instances.

By default, BQM analysis results are stored as node voltage heatmaps. To view the corresponding voltage drop across instances, use the `generate_pin_voltage_heatmaps` command. See [Generating Instance Voltage Drop Heatmaps](#) on page 68.

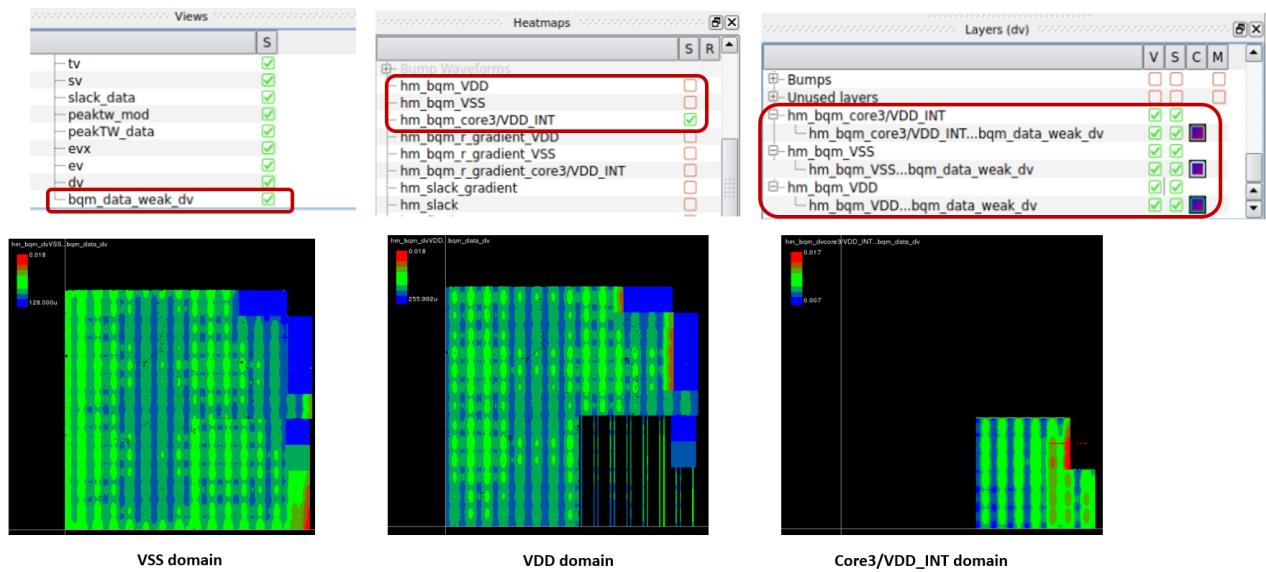
The following figure shows the node voltage heatmap from BQM simulation. The weak region due to missing bumps is highlighted. [Creating Instance Drop Heatmap from Node Voltage Heatmap](#) shows the instance voltage drop heatmap that is generated from the node voltage heatmap.

Figure 29. Node Voltage Heatmap from BQM Simulation**Figure 30. Creating Instance Drop Heatmap from Node Voltage Heatmap**

The following figure shows how the VDD and VSS node voltages translate to voltage drops across instance pins.



To view a BQM instance voltage drop heatmap in GUI, make the required selections under **Views**, **Heatmaps**, and **Layers** as shown in the following figure.



4.3.4. Performing BQM Analysis for Macro Cells

The BQM flow for macro cells is similar to that of standard cells.

To generate probes and their currents for macro BQM analysis, use the `generate_macro_bqm_current_probe_metal` command:

```
bqm_current.generate_macro_bqm_current_probe_metal
```

The current is assigned only to the macro LEF pins.

The following example shows how to use the `generate_macro_bqm_current_probe_metal` command to perform BQM analysis for macro cells. The command generates equidistant probes and a current source for each probe at the lowest metal layer. The probes and their currents are used to generate node voltage heatmaps.

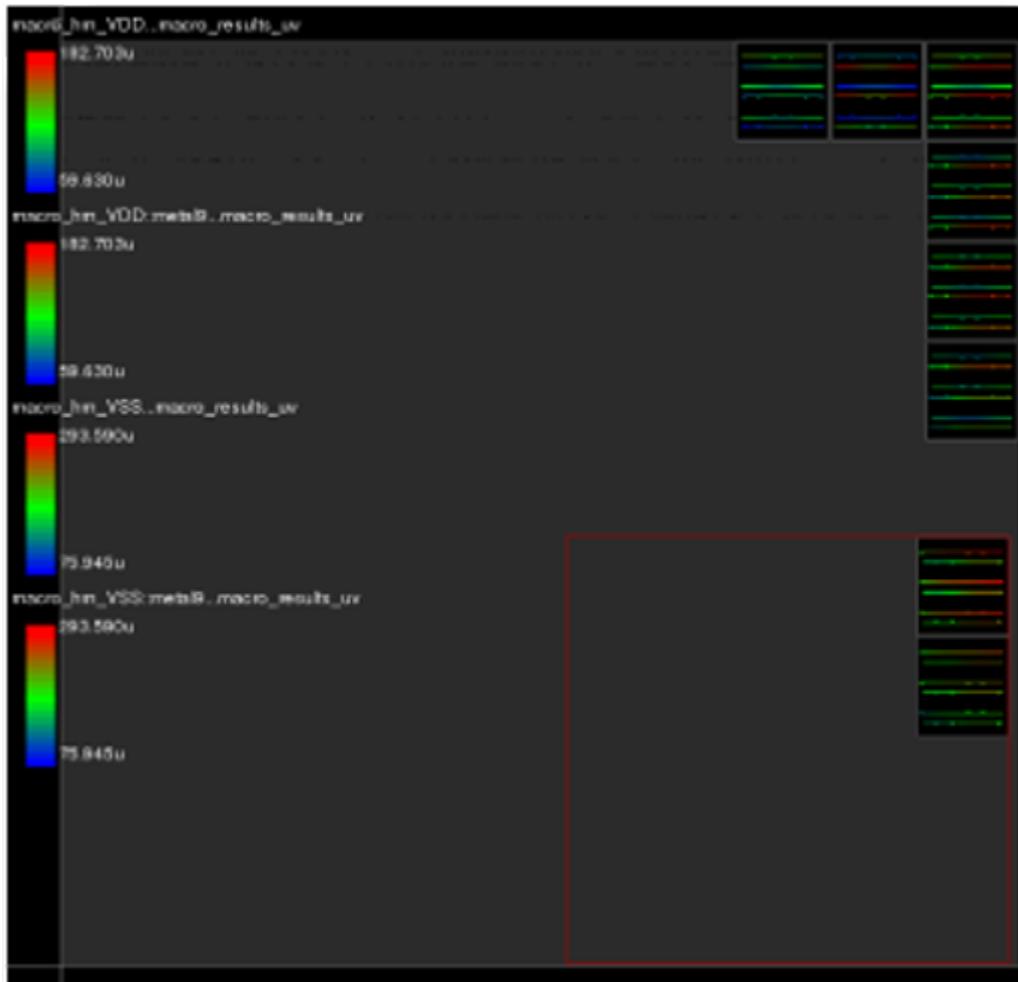
```
import bqm_current

#Generate probe locations and currents for Build Quality Metric for Macros.
bqm_current.generate_macro_bqm_current_probe_metal(dv, check_layer=None,
probe_distance=0.2, hor_ver_ratio=10.0, current_per_micron=1.0, macro_chunk_data=None)

#Storing the results in UserView from 'bqm' AnalysisView
get_macro_bqm_data(bqm, macro_chunk_data, results_user_view)
```

The following figure shows a typical BQM heatmap for macro cells.

Figure 31. BQM Heatmap for Macros

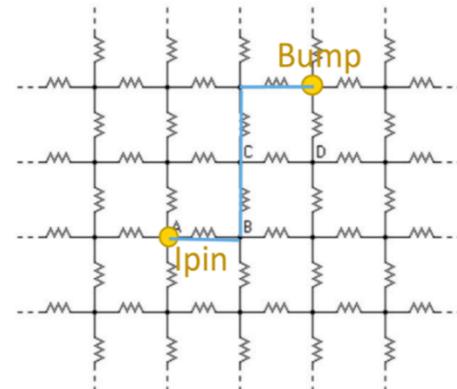
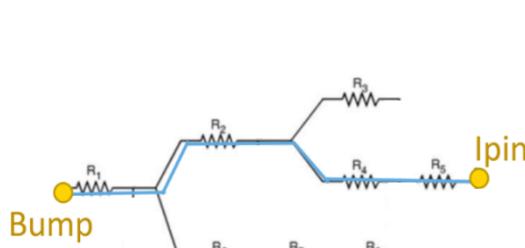


4.4. Tracing Shortest Path Resistance (SPR)

Shortest path resistance (SPR) is the total resistance of the shortest electrical path from a voltage source to an instance pin or any particular location of the grid.

The SPR technique applies only to PG nets. As the lowest metal layer contributes most to SPR, you can use this technique for manual debugging.

Note: Good SPR values do not imply that there are no problems with the power grid.



The following sections describe SPR tracing, querying, and reporting for individual chips. For information about SPR tracing in multichip systems, see [SPR for Multichip Designs](#) on page 369 in the [Multichip Analysis](#) on page 329 chapter.

4.4.1. Generating Shortest Path Resistance Data

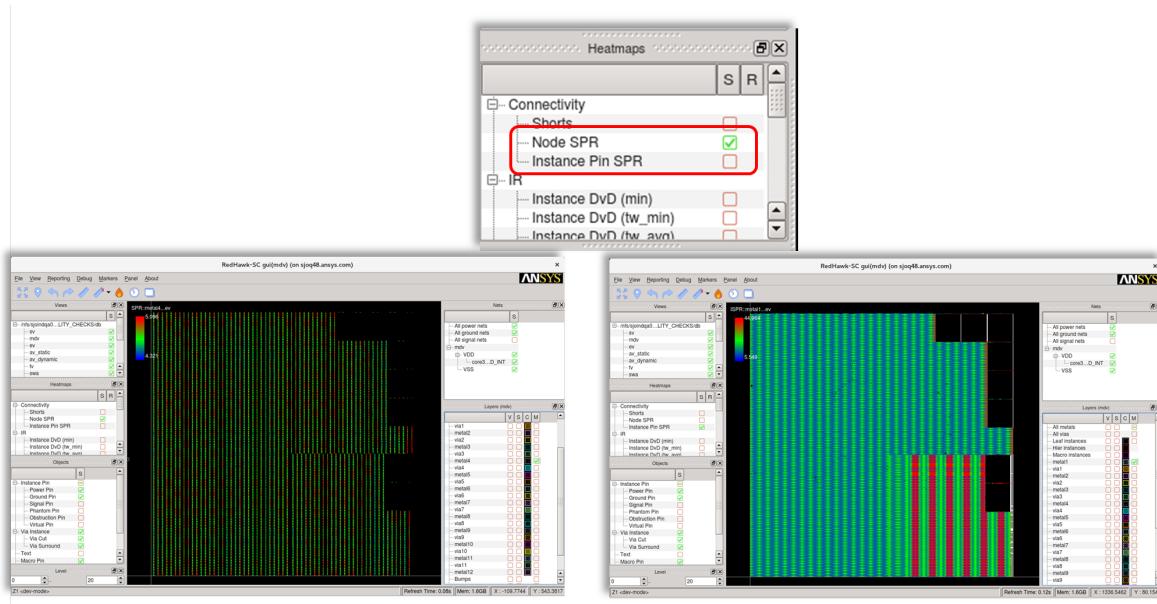
SPR data is not generated by default in the ExtractView. To generate SPR data, set the `calculate_spr` key under the `settings` dict to `True` with the `create_extract_view` command as follows:

```
extraction_settings = {..., "calculate_spr" : True}
ev = db.create_extract_view(..., settings = extraction_settings)
```

If the ExtractView is already generated without specifying `calculate_spr=True`, use the following function to generate SPR data:

```
mev = db.perform_shortest_resistance_path_check(ev, options)
```

Based on the SPR data from ExtractView, the layout GUI has pre-populated SPR heatmaps as shown in the following figure. To view these heatmaps, select **Node SPR** or **Instance Pin SPR** in **Heatmaps**.



Reporting SPR for Signal Nets

To report SPR for signal nets, query the Circuit object in ExtractView by using the `Circuit.perform_shortest_resistance_path_check` function as shown in the following example:

```
loc1={'layer':Layer('metal1'), 'coord':(403.380,953.420)}
loc2={'layer':Layer('metal5'), 'coord':(403.290,986.200)}

net=Net('n12237')
circuit=ev.get_signal_net_circuit(net)
spr=circuit.perform_shortest_resistance_path_check(start_point=loc1,end_points=loc2)
print(spr)
```

The tool calculates the SPR between `start_point` and `end_points`, where `start_point` and `end_points` can either be locations on the net or instance pins.

The following is a typical report example of SPR traced for signal nets.

```
# Traced Shortest Resistance Path
From: Node: RealCoord(403.380000,953.320000) Layer('metal1') Net('n12237') Cost(635.071ohm)
To: Bump RealCoord(403.290000,986.200000) Layer('metal5') Net('n12237') Cost(0.0ohm)

# Location(x,y)      Layer  Cost(ohm)  ResDiff  Length(um)  Width(um)  Drop(mv)  DropDiff   Net  Comment
( 403.380, 953.320)  metal1  635.071  ==       ==        ==        ==        ==        n12237 # --
( 403.380, 953.320)  metal2  611.136  23.935   ==        ==        ==        ==        n12237 # via: VIA1A_DA
( 403.380, 953.320)  metal3  585.071  26.064   ==        ==        ==        ==        n12237 # via: VIA2AP_DA
( 403.380, 954.680)  metal3  563.267  21.804   1.36      0.044    ==        ==        n12237 # wire: metal3
( 403.380, 954.680)  metal4  533.098  30.169   ==        ==        ==        ==        n12237 # via: VIA3A_DA
( 403.290, 954.680)  metal4  531.406  1.693    0.09      0.04     ==        ==        n12237 # wire: metal4
( 403.290, 954.680)  metal5  505.341  26.064   ==        ==        ==        ==        n12237 # via: VIA4AP_DA
( 403.290, 986.200)  metal5  0.000   505.341  31.52      0.044    ==        ==        n12237 # wire: metal5
```

4.4.2. Querying SPR of Instance Pins

To query an SPR value, use the `get_instance_pin_spr` command:

```
ExtractView.get_instance_pin_spr(instance,pin,spr_report_type='worst')
```

The `get_instance_pin_spr` command returns the layer name, SPR value, and the coordinates of the instance pin as shown in the following examples:

```
ev.get_instance_pin_spr(inst,Pin('VDD'),spr_report_type='worst')

{'layer': Layer('metal1'), 'value': 15.770429611206055,
 'coord': RealCoord(508.56,840)}
```

```
ev.get_instance_pin_spr(inst,Pin('VSS'),spr_report_type='worst')

{'layer': Layer('metal1'), 'value': 9.229924201965332,
 'coord': RealCoord(507.8,838.6)}
```

For more details about the `get_instance_pin_spr` command, use the `help` command at the tool command prompt:

```
help(ExtractView.get_instance_pin_spr)
```

4.4.3. Querying Pin SPR Path Data

To query the least resistive path from a given location or an instance pin to the nearest bump, use the `get_shortest_resistance_path` command:

```
ExtractView.get_shortest_resistance_path
```

To query the least resistive path from an instance pin to the nearest bump, specify the `instance` argument with the `get_shortest_resistance_path` command as shown in the following example:

```
spr_path = ev.get_shortest_resistance_path(instance=inst)
print spr_path[(inst,Pin('VDD'))]
```

A typical report of the traced path is as follows:

```
# Traced Shortest Resistance Path
From: Instance('core0.regfile_data_memory.MUX2_X1_2686'):Pin('VDD'):
      RealCoord(58.56,84) Layer('metal1') Net('VDD') Cost(15.77ohm)
To:   Bump Pin(VDD_147) RealCoord(48,83.38) Layer('metal12') Net('VDD')
Cost(0.0ohm)

# Location(x,y) Layer Cost(ohm) ResDiff Drop(mv) DropDiff Net Comment
(58.56, 84.00) metal1 15.770    --    --    --    VDD  #--
(45.14, 84.00) metal1  6.857    8.913   --    --    VDD  #wire: metal1
(45.14, 84.00) metal2  5.924    0.934   --    --    VDD  #via: vial_4_1
(44.94, 84.00) metal2  5.789    0.135   --    --    VDD  #wire: metal2
...
...
```

For more details, use the `help` command at the tool command prompt:

```
help(ExtractView.get_shortest_resistance_path)
```

4.4.4. Viewing Pin SPR Values

SPR node and pin information are stored as heatmaps. When you select **Instance Pin SPR** under **Connectivity** in **Heatmaps** panel and the corresponding pin layer under **M** column in **Layers** panel, the GUI displays the instance pin SPR values.

The following figure shows the instance pin SPR heatmap of all instance pins across multiple layers. [Figure 33: Instance Pin SPR Values](#) on page 78 shows the magnified view where the instance pins are marked with the corresponding SPR values when **Text** in **Objects** panel is selected.

Figure 32. Instance Pin SPR Heatmap

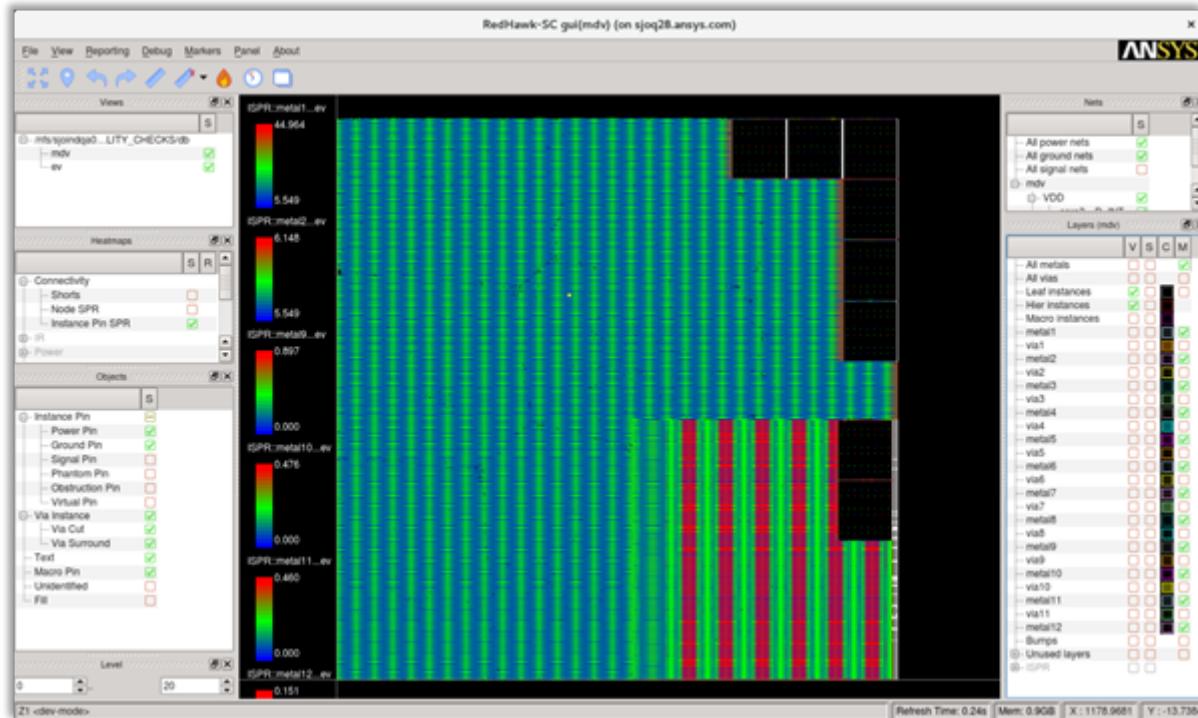
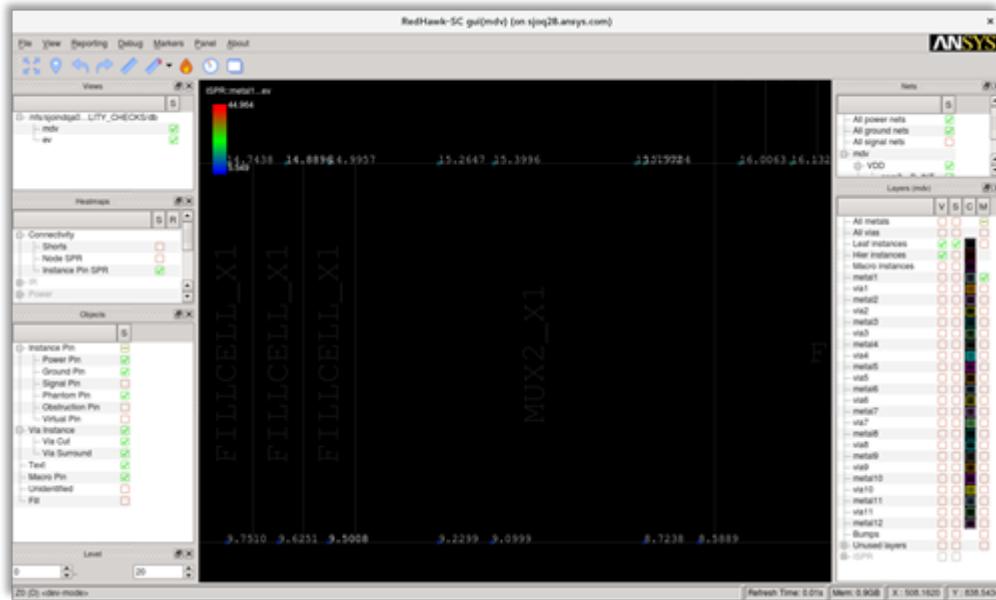
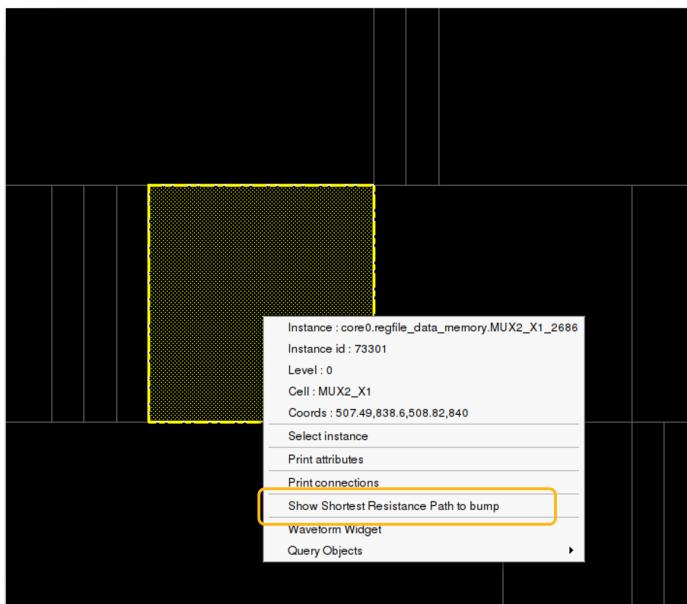


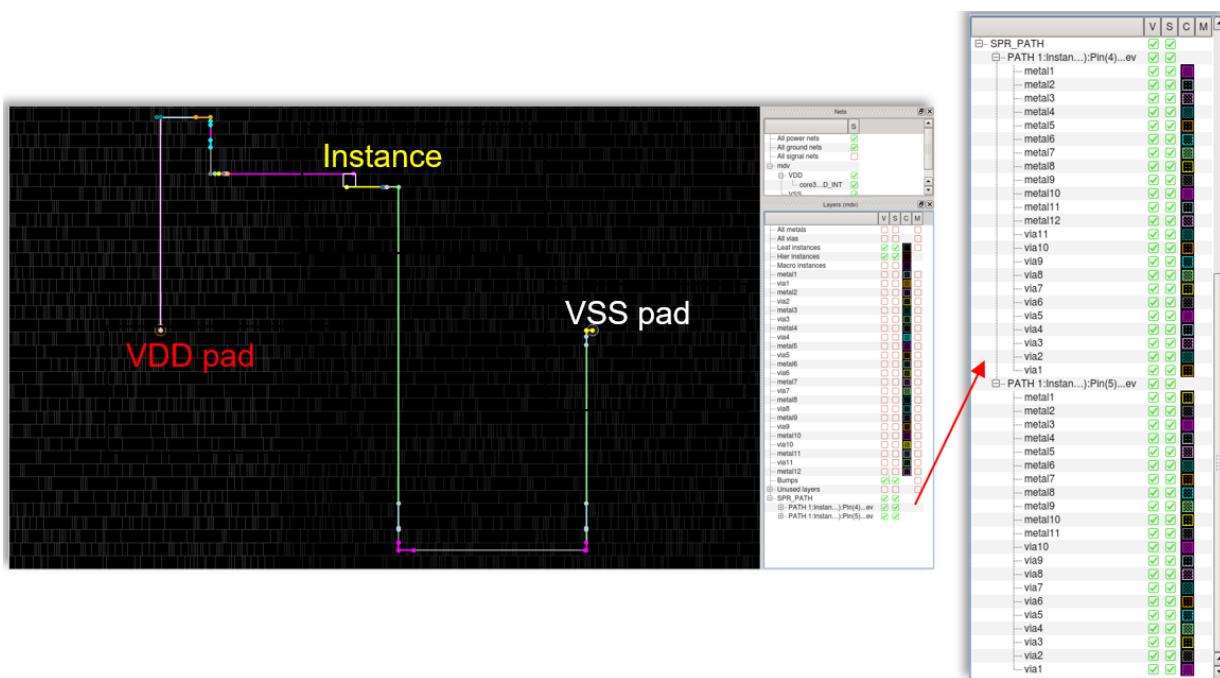
Figure 33. Instance Pin SPR Values

To trace the SPR path for an instance in GUI, follow these steps:

1. Select the instance in the display area.
2. Right-click and select **Show Shortest Resistance Path to bump** as shown:



The GUI displays the SPR path from instance PG pins to bumps:



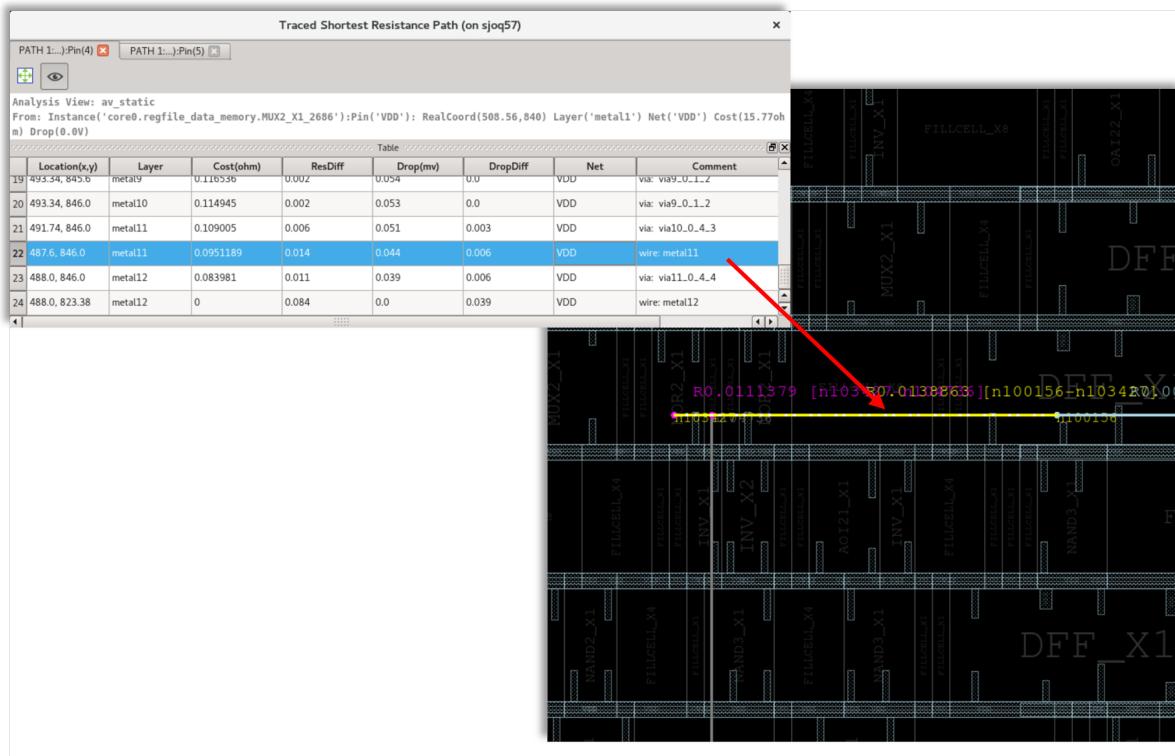
The path is divided into line segments and highlighted in different colors. To control the SPR path layer visibility, select the available **SPR_PATH** options in the **Layers** panel. SPR traces of the same layer in different paths have the same color.

Further, a browser showing instance pin SPR path data in tables is automatically opened.

The browser table data is similar to the SPR path data report generated by the `get_shortest_resistance_path` command. See [Querying Pin SPR Path Data](#) on page 76.

Each row of the table corresponds to a line segment in the traced path. You can double-click any row of the table to zoom the region where the line segment is located.

You can click any column header to sort them based on ascending or descending order. You can also use the search filter at the bottom of the table. The filter supports string, int, and float formats based on the selected column.



4.4.5. Querying SPR From a Location

To query the least resistive path from a given location to the nearest bump, specify `net`, `layer`, and the location coordinates with the `get_shortest_resistance_path` command as shown in the following example. For details of the command syntax, see [Querying Pin SPR Path Data on page 76](#).

```
spr_path=ev.get_shortest_resistance_path(net=Net('VSS'),layer=Layer('metal9'),  
x=886.869,y=1205.33)  
print spr_path
```

A typical report of the traced path is as follows:

```
# Traced Shortest Resistance Path  
From: Node: RealCoord(86.86,15.33) Layer('metal9') Net('VSS') Cost(0.46ohm)  
To: Bump Pin(VSS_258) RealCoord(02,13.39) Layer('metal12') Net('VSS') Cost(0.0ohm)  
  
# Location(x,y) Layer Cost(ohm) ResDiff Drop (mv) DropDiff Net Comment  
(86.87, 15.33) metal9 0.46 -- -- -- VSS # --  
(92.00, 15.33) metal9 0.35 0.11 -- -- -- VSS # wire:m9  
(92.80, 15.33) metal12 0.35 0.00 -- -- -- VSS # wire:m12  
(94.40, 15.33) metal12 0.35 0.00 -- -- -- VSS # wire:m12  
(94.40, 13.39) metal12 0.00 0.34 -- -- -- VSS # wire:m12  
(92.00, 13.39) metal12 0.00 0.00 -- -- -- VSS # wire:m12
```

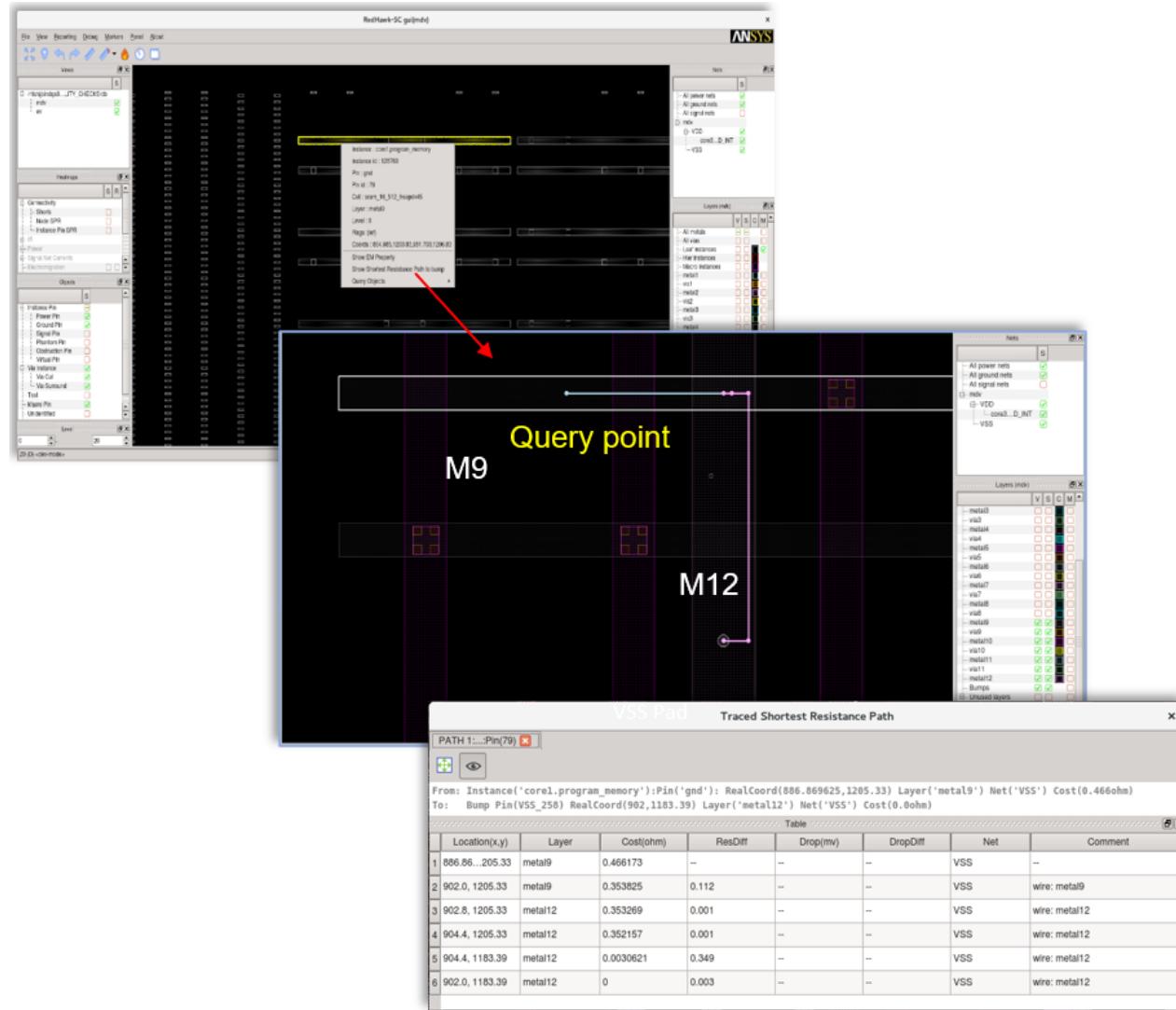
For more details, use the `help` command at the tool command prompt:

```
help(ExtractView.get_shortest_resistance_path)
```

4.4.6. Viewing SPR Path For a Location

With node SPR information shown on PG routes, you can trace SPR path from those nodes. After selecting a PG route, right-click **Show Shortest Resistance Path to Bump** as shown in the figure.

Figure 34. SPR Path from a Node location



4.4.7. Generating SPR Text Report

To generate a text report of the SPR data of instances, use the `report_instance_pin_spr` command:

```
emir_reports.report_instance_pin_spr
```

Use the `ignore_cell_types` argument to control the cell types that are reported. By default, the command skips reporting decoupling capacitance and filler cells. You can specify one or more of the following cell types as a list with this argument.

Table 3: Valid Cell Types

Cell Type	Description
is_bank	Multibit Sequential Cell
is_bump_cell	Bump Cell
is_cmm_macro_cell	CMM Macro Cell
is_decap_cell	Decoupling Capacitor Cell
is_def_cell	DEF Based Cell (Cell definition from DEF)
is_edge_triggered	Edge Triggered Flip Flop
is_edge_triggered_bank	Edge Triggered Multibit Flip Flop
is_fall_edge_triggered	Fall edge triggered Sequential Cell
is_filler_cell	Filler Cell
is_gds_cell	GDS Cell
is_icg_cell	Integrated Clock Cell
is_lef_cell	LEF Based Cell (Cell definition from LEF)
is_level_sensitive	Level Sensitive Latch Cell
is_level_sensitive_bank	Level Sensitive Multibit Latch Cell
is_level_shifter	Level Shifter Cell
is_lib_macro_cell	Liberty Macro Cell
is_macro_cell_view	Cell with Cell View Modes
is_macro_mmx_view	Cell with MMX Model
is_pad_cell	Pad Cell
is_power_gate	Power Gate Cell
is_rise_edge_triggered	Rise Edge Triggered Sequential Cell
is_seq_cell	Sequential Cell
is_shadow_cell	Macro Shadow (inner) Cell
is_shadow_parent_cell	Macro Shadow Parent (Outer) Cell
is_signal_cell	Cell with Signal Pins
is_std_cell	Standard Cells
is_via_cell	Via Cells

Use the `columns` argument to customize the columns that are reported. Here are the list of valid column names for the `columns` argument

Table 4: Valid Column Names

Column Names	Description
pin_name	The pin name of the instance for which SPR is reported
spr_value	The SPR Value of the instance-pin
cell_name	The cell name of the instance
x	Lower Left x coordinate for the instance bounding box
y	Lower Left y coordinate for the instance bounding box
layer	The layer of the instance-pin for which SPR is reported
instance_name	Name of the instance for which SPR is reported

The following example shows how to generate a default instance pin SPR report.

```
emir_reports.report_instance_pin_spr(ev)
```

A snippet of the default report file, ./instance_pin_SPR.rpt, is as follows:

```
#pin_name  spr_value  cell_name   x        y        layer    instance_name
VSS       345.183    INV_X4     1116.895  1.4      metall1 ZINV_11
VSS       343.164    INV_X4     1113.855  1.4      metall1 ZINV_12
VDD       327.482    INV_X4     1116.895  0.0      metall1 ZINV_11
VDD       325.467    INV_X4     1113.855  0.0      metall1 ZINV_12
VDD       219.254    BUF_X8     1167.65   435.565  metall1 ZBUF_11
VSS       211.245    CBUF_X1   1121.46   1098.4   metall1 ZBUF_12
... ... ...
```

The following example shows how to customize the columns to be reported. When you report all the instances in the design by setting `max_lines` to None, you must also set `sort` to False.

```
columns = ['spr_value', 'layer', 'pin_name', 'instance_name']
formats = '{0:10.4} {1:10} {2:10} {3}\n'
header = '{0:10} {1:10} {2:10} {3}\n'.format(*columns)
emir_reports.report_instance_pin_spr(ev, columns=columns, header=header,
formats=formats, max_lines=None, sort=False)
```

A snippet of the report is as follows:

```
spr_value  pin_name  instance_name
11.04      VDD       ZBUF_11
15.34      VSS       ZBUF_12
27.48      VDD       CINV_11
12.46      VSS       CINV_12
... ... ...
```

The following example shows how to report SPR information for the PG arcs of the design.

```
emir_reports.report_instance_pin_spr(ev,
report_pg_arcs={'report_absolute_resistance': True,
'minimal_lines_per_arc': 1000},output_file=reports_dir+'instance_pg_arc_spr.rpt)
```

When you set `report_absolute_resistance` to True, resistance is reported in unit of Ohms. When you set `report_absolute_resistance` to False, resistance is reported in percent. A typical report snippet is as follows:

```
#Maximum resistance: 673.197235 ohms

#Arc1 VDD VSS 100.0000 %
#Arc2 core3/VDD_INT VSS 43.1111 %

{ Power/Ground Arc1(VDD VSS):
#pg_spr vdd_spr vss_spr location_x location_y p_pin g_pin InstanceName

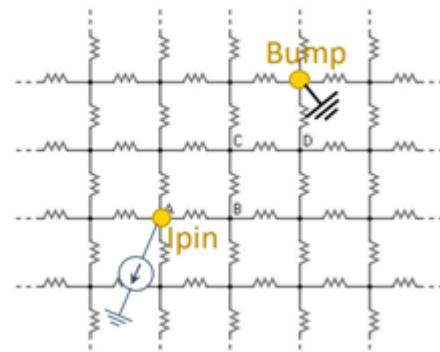
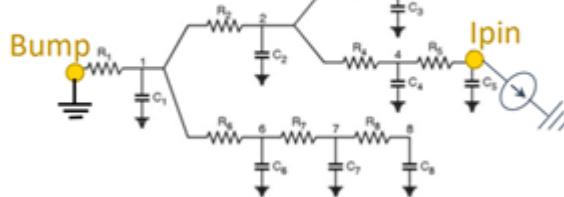
% % %
100.0000 48.6458 51.3542 345.183 1116.895 VDD VSS ZINV_11
99.4008 48.3465 51.0543 343.164 1110.985 VDD VSS ZINV_12
99.1758 48.2342 50.9415 327.482 995.687 VDD VSS ZBUF_11
35.1663 31.3866 3.7797 325.467 1113.855 VDD VSS ZBUF_12
...
{ Power/Ground Arc2(core3/VDD_INT VSS):
...
...
```

For more details, use the `help` command at the tool command prompt:

```
help(emir_reports.report_instance_pin_spr)
```

4.5. Computing Effective Resistance

Effective resistance (R_{eff}) is the resistance computed from an instance pin or probe to all bumps (shorted). As most of the resistance is at the lowest metal layer, you can use this technique to check the relative PG grid strength in early prototyping stages.



Prerequisites

To calculate effective resistance data, you must have already created the following views:

1. A DesignView with LEF, DEF, and GDS2DEF input data.

Note: Liberty input data is not mandatory for effective resistance calculation.

2. An ExtractView to extract the impedance (RLC) models of power delivery network for the input data.
3. A SimulationView to generate a reduced model of the RLC network from ExtractView.

To compute the effective resistance for all phantoms/instance pins in a design, create an EffectiveResistanceView using the `create_effective_resistance_view` command as follows:

```
reff = db.create_effective_resistance_view(simulation_view=sv, **reff_args)
```

`reff_args` is a dictionary of arguments. For example,

```
reff_args = dict(
    tag = 'reff',
    select_settings = {
        'exclude_cells':cm_convert_to_regex(['*FILLCELL*', '*ram*']),
    },
    options = options)
```

By default, the tool skips the computation of effective resistance for pad cells, filler cells, decoupling capacitance cells, and macro cells (both LEF and DEF models). Use the `select_settings` argument to selectively compute effective resistance.

The following example includes filler and decoupling capacitance cells in the effective resistance computation.

```
select_settings={'cell_types': ['is_filler_cell', 'is_decap_cell']}
```

The following example explicitly excludes pad cells from the effective resistance computation.

```
select_settings={'exclude_cell_types': ['is_pad_cell']}
```

The following example computes effective resistance for all instances except for instances with the term, CCAP, in cell names.

```
select_settings = {
    'exclude_cells': '*CCAP*',
}
```

The following example computes the effective resistance for the specified region. The calculation ignores instance pins connected to domain names starting with VSS.

```
select_settings = {
    'region':RealRect(10,10,100,100),
    'exclude_nets':'VSS*',
}
```

The following example computes the effective resistance only for instance names starting with `inst_sw` and cell names starting with `ram`.

```
select_settings = {
    'instances': 'inst_sw*',
    'cells': 'ram*',
}
```

By default, the tool calculates the effective resistance for all phantoms of a design.

Note: For computation purposes, the tool assigns one or more current sink locations in each pin. The location of a current sink is called a phantom.

The EffectiveResistanceView stores the effective resistance of each instance pin.

- For an instance pin with a single phantom, a single value is computed and stored.
- For an instance pin with multiple phantoms, effective resistance for each phantom is computed. However, only the minimum, maximum, and mean values are saved.

For more details, use the `help` command at the tool command prompt:

```
help(SeaScapeDB.create_effective_resistance_view)
```

4.5.1. Computing Effective Resistance for Macro Cells

The following considerations apply to computation of effective resistance for macro cells:

- For macro cells with only LEF models, behavior is the same as that of standard cells.
- For macro cells with GDS2RH models:
 - All pins from DEF are treated separately and values are stored for each of vdd, vdd.gds1, vdd.gds2 pins.
 - Any grouping of DEF pins from the DesignView is honored.
- The following options control effective resistance computation for macro cells.

```
#This option is used to include/exclude shadow parent pins in computation of
effective resistance.
options['effective_resistance_options'].include_shadow_parent_pins = True/False
(Default True)

#This option is used to include/exclude DEF clone pins in computation of
effective resistance.
options['effective_resistance_options'].include_shadow_cells = True/False
(Default True)

#To skip cells if any dimension is larger than max_cell_size in computation
of effective resistance.
options['effective_resistance_options'].max_cell_size = 200 (Default value in
u)
```

4.5.2. Generating Effective Resistance Text Report

To generate a text report of effective resistance of instances, use the `report_effective_resistance` command:

```
emir_reports.report_effective_resistance
```

Argument	Description
reff	The EffectiveResistanceView from which to write the Reports (required)
file_name	The path to the output report file (optional. Default value is './reff_report.rpt')
report_type	One of 'min_res', 'avg_res', 'max_res' or None. Reports minimum, average or maximum resistance among all the pin shapes for an instance-pin. If set to None, reports all three metrics

Argument	Description
max_lines	The number of lines in the output report (optional. Default is 5000)
sort_columns	A list of columns that are used to sort the report. Can be any or multiple of ['min_res', 'avg_res', 'max_res']. (optional. Default is to sort by 'max_res'. This is active only if max_lines is not set to None)

To calculate effective resistance, RedHawk-SC does a Static Solve with current sources connected to each instance-pin shapes separately.

For a pin with multiple shapes, such as in multi-height cells, the effective resistance of each of the shapes is calculated separately. For each instance-pin pair, the tool saves the maximum, minimum, and average effective resistance values. To report only one of the maximum, minimum, or average effective resistance values, use the `report_type` argument.

The following example shows how to generate an effective resistance report of all instance pins in the design.

```
emir_reports.report_effective_resistance(reff)
```

By default, the report is limited to 5000 instance-pin pairs that are sorted by the maximum resistance of each pin. A snippet of the report is as follows:

```
#Reff Report
#min      avg      max      loc_x      loc_y      pin_      net_
#           name     name     name      name      name     name
339.509247 339.760406 340.012726 1117.272500 1.400000  vss       VSS      ZINV_11
337.490479 337.741669 337.993958 1114.232500 1.400000  vss       VSS      ZINV_12
336.731110 336.982269 337.234558 1113.092500 1.400000  vss       VSS      ZINV_13
321.325287 321.574982 321.825745 1117.272500 0.000000  vdd       VDD      ZINV_14
...
...
```

The following example reports the first 50 instances with highest effective resistance values.

```
emir_reports.report_effective_resistance(reff, report_type='max_res',
                                         max_lines=50)
```

For more details, use the `help` command at the tool command prompt:

```
help(emir_reports.report_effective_resistance)
```

4.5.3. Querying Effective Resistance of Instance Pins and Probes

To query the effective resistance of each instance pin, use the `get_effective_resistance` command:

```
reff.get_effective_resistance(Instance(<inst_name>), Pin(<pin_name>))
```

The following example queries the effective resistance of an instance pin with single phantom.

```
data = reff.get_effective_resistance(Instance('U1/U2'), Pin('vdd'))
print data
```

```
Min: 20.482 Max: 20.482 Mean: 20.482 Count: 1
```

By default, the `get_effective_resistance` query command returns the resistance values of all pin nodes. To output the information for each node, use the `sub_pin` argument as shown in the following example:

```
print
reff.get_effective_resistance(Instance('inst1'), Pin('vssfx'), sub_pin =
SubPin(3))

Min: 22.0051 Max: 22.0051 Mean: 22.0051 Count: 1
```

The following example queries the effective resistance of an instance pin with multiple phantoms.

```
data = reff.get_effective_resistance(
    Instance("core1.regfile_program_memory.DFF_X1_3915"), Pin('VDD'))
print data

Min: 6.9808 Max: 8.75468 Mean: 7.83091 Count: 5
```

Probes are inserted by using specific inputs to the `create_modified_extract_view` command.

1. To enable effective resistance measurement for probes, use the `probes` key under the `select_settings` of the `create_effective_resistance_view` command, as shown:

```
select_settings = {'probes':'*'}
reff = db.create_effective_resistance_view(..., select_settings=select_settings)
```

2. To query the effective resistance computed for probes, use:

```
data = reff.get_effective_resistance(probe='<probe_name>')
```

The following example shows how to query the effective resistance computed for probes and the corresponding output.

```
data = reff.get_effective_resistance(probe="VDD.0.1")
print data
```

Output:

```
Min: 20.482 Max: 20.482 Mean: 20.482 Count: 1
```

For more details, use the `help` command at the tool command prompt:

```
help(EffectiveResistanceView.get_effective_resistance)
```

4.5.4. Querying Phantom Layers and Locations

To query the details of layer and location of phantoms associated with the maximum and the minimum effective resistance values, use the `get_effective_resistance_bounds` command:

```
reff.get_effective_resistance_bounds(Instance(<inst_name>), Pin(<pin_name>))
```

For example,

```
data = reff.get_effective_resistance_bounds(Instance
("core1.regfile_program_memory.DFF_X1_3915"), Pin('VDD'))
```

```

print data

{'max': 8.754682540893555, 'max_loc': (RealCoord(1.225000,747.600000),
Layer('metall1')),  

'min_loc': (RealCoord(3.890000,747.600000), Layer('metall1')), 'min':  

6.980804920196533}

```

For more details, use the `help` command at the tool command prompt:

```
help(EffectiveResistanceView.get_effective_resistance_bounds)
```

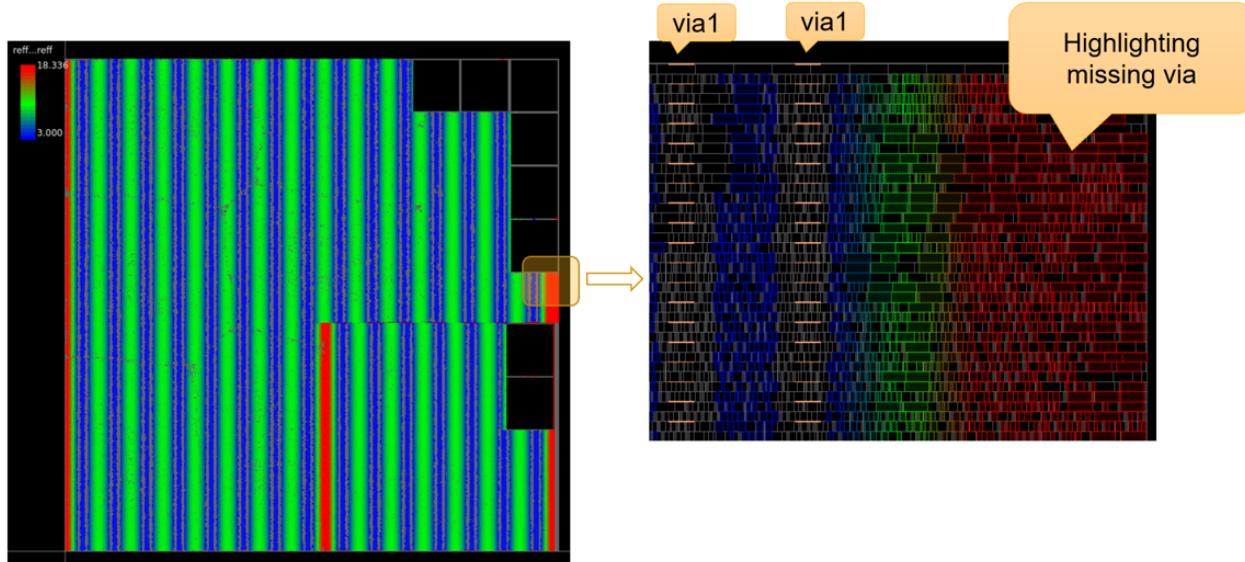
4.5.5. Viewing Effective Resistance Heatmaps

To view effective resistance heatmaps in GUI:

1. Select the effective resistance view name in **Views**.
2. Select **reff** in **Heatmaps**.
3. Select the corresponding layer in **Layers**.
4. Select the VDD or the VSS net in **Nets**.

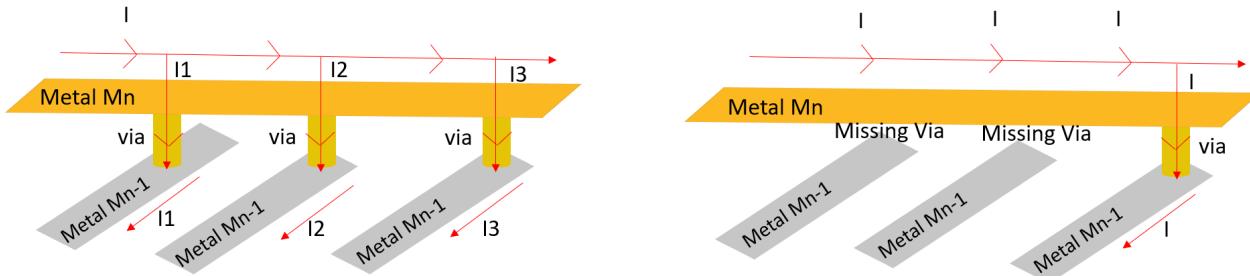
Note: By default, the heatmap shows the mean effective resistance values for either the VDD or the VSS net. If you select both VDD and VSS, the heatmap represents the greater of the two effective resistance values.

The following figure shows a typical heatmap of effective resistance values of a design. The effective resistance values indicate how far an instance pin is from the nearest via cut at the lowest metal layer. You can also use the effective resistance values to estimate the self-switching voltage drop across an instance pin ($voltage_drop = demand_current \times Reff$).



4.6. Reporting Missing Vias

Missing vias cause currents to flow through longer resistive paths causing grid weakness. You check for missing vias to identify the corresponding weak power grid areas.



By default, the tool does not check for missing via locations between two metal layers. To check for missing vias, use the following command after DesignView is created:

```
db.perform_missing_via_check()
```

For example,

```
db.perform_missing_via_check(design_view=mdv, tag='missing_via_check')
```

The check results are stored in a UserView as a heatmap with missing via locations. The default UserView is `missing_via_check`.

Including or Excluding Instances

You can include instances to or exclude instances from the missing via check.

To do so, use the `settings` dict with the `perform_missing_via_check` command as shown in the following examples. Each key accepts a list. Each element of the list is a string (cell or instance name), object, or CMRegex pattern.

- `exclude_cells`: Excludes all instances of specified cells from the missing via check.

```
db.perform_missing_via_check(..., settings={..., exclude_cells:
[Cell("abc"), "bc*", CMRegex("b.")], ...}, ...)
```

- `exclude_instances`: Excludes specified instances from the missing via check.

```
db.perform_missing_via_check(..., settings={..., exclude_instances:
[Instance("inst_abc"), "inst_1*", CMRegex("ins.")], ...}, ...)
```

- `include_cells`: Includes the geometries of only specified cells to the missing via check.

```
db.perform_missing_via_check(..., settings={..., include_cells:
[Cell("abc"), "bc*", CMRegex("b.")], ...}, ...)
```

- `include_instances`: Includes the geometries of only specified instances to the missing via check.

```
db.perform_missing_via_check(..., settings={..., include_instances:
[Instance("inst_abc"), "inst_1*", CMRegex("ins.")], ...}, ...)
```

Note: Do not specify `include_cells` and `include_instances`, or `exclude_cells` and `exclude_instances` together. Otherwise, the tool issues an error.

For more details, use the `help` command at the tool command prompt:

```
help(db.perform_missing_via_check)
```

4.6.1. Generating Missing Via Locations Text Report

To generate a text report of missing via locations, use the `report_missing_vias` command:

```
heatmap_utils.report_missing_vias()
```

The following example shows how to generate a missing via report and the output file.

```
heatmap_utils.report_missing_vias(missing_via_view=missng_via_check,file_name='./missing_via_locations.rpt',redhawk_style=True)
```

```
Net Layer1 Layer2 X,Y
VDD metal12 metal11 1040,18
VDD metal12 metal11 1040,110
VDD metal12 metal11 1040,202
VSS metal12 metal11 1178,64
VSS metal12 metal11 1178,156
...
```

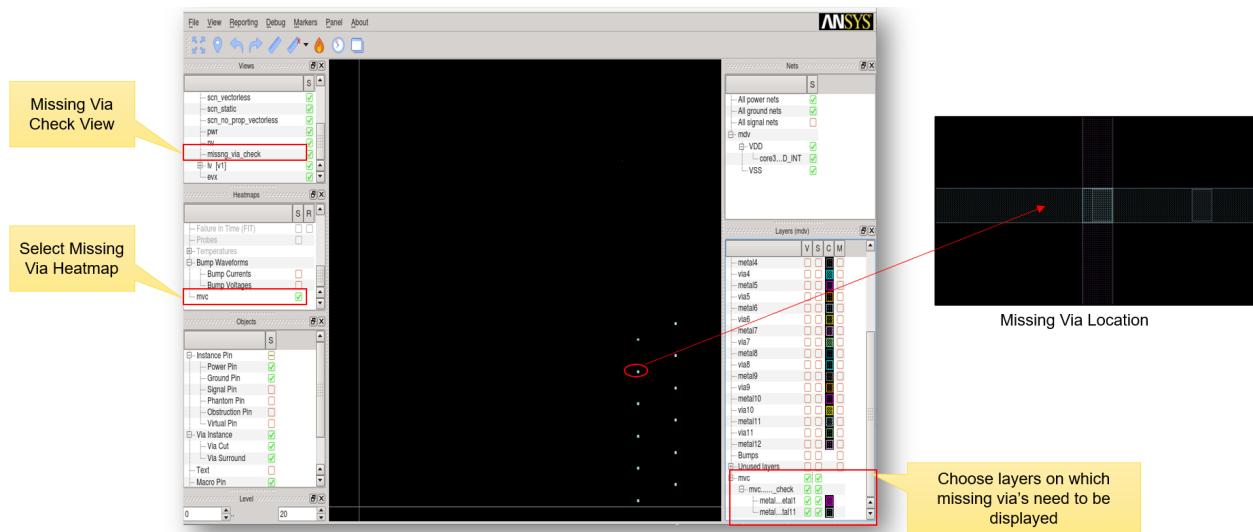
For more details, use the `help` command at the tool command prompt:

```
help(heatmap_utils.report_missing_vias)
```

4.6.2. Viewing Missing Vias

To view missing vias in GUI, follow these steps:

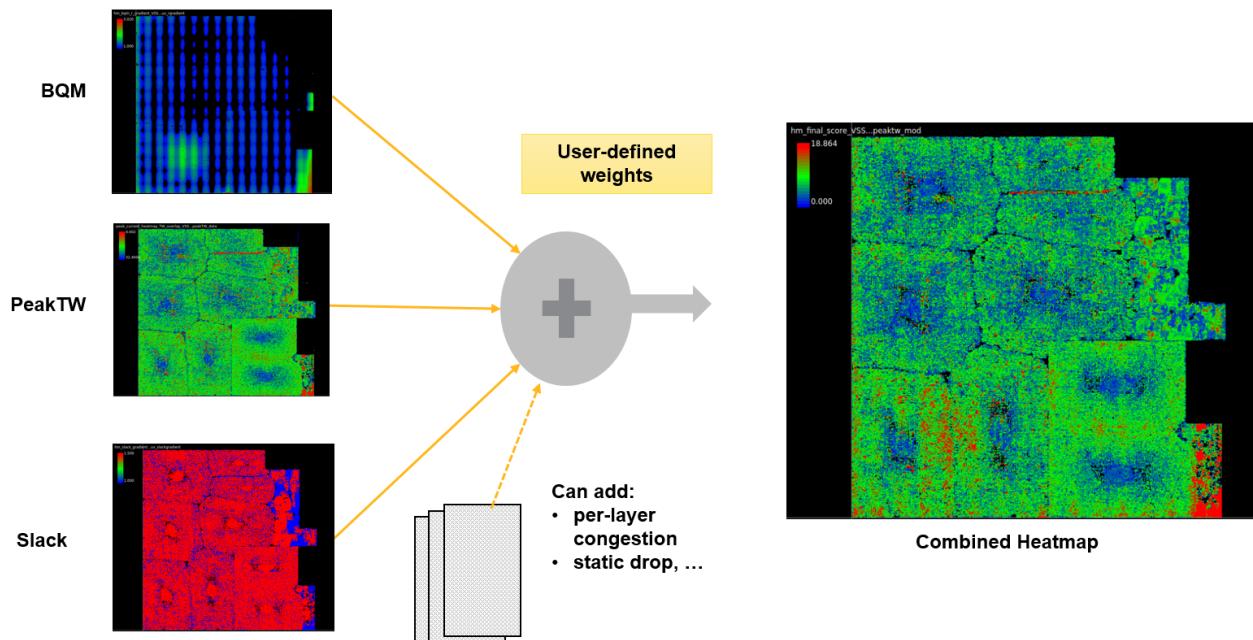
1. Select **missing_via_check** in **Views**.
2. Select **mvc** in **Heatmaps**.
3. Select the layers that have missing vias in **Layers**.

Figure 35. Missing Via Locations in GUI

4.7. Creating Statistical Heatmaps for Multiple Parameters

Multiple parameters, such as simultaneous switching (peakTW), slack, and per-layer congestion, contribute to power grid weakness. Statistical techniques that combine the impact of these parameters with the resistive grid checks (such as BQM) enable you to analyze grid weakness.

The following example shows generating a single heatmap from BQM, simultaneous switching (peakTW), and slack data.



The following topics describe how to generate a single custom heatmap from multiple parameter heatmaps, and use the heatmap to analyze grid weakness.

- [Computing Resistance Gradient from BQM Data](#) on page 93
- [Analyzing PeakTW Currents](#) on page 94

- [Analyzing Slack](#) on page 97
- [Combining Heatmaps](#) on page 99

4.7.1. Computing Resistance Gradient from BQM Data

To compute the resistance gradient from BQM data, follow these steps:

1. Generate BQM probes and currents. See [Generating Probes and Currents](#) on page 67.
2. Generate and store instance voltage drop heatmaps. See [Generating Instance Voltage Drop Heatmaps](#) on page 68.
3. Run the following example script to compute the resistance gradient for PG nets of the design.

```

gnd_count = list()
for agnd in dv.get_nets('ground'):
    if not bqm_inst.get(agnd) is None:
        gnd_count.append((agnd,bqm_inst[agnd].get_stats()['0'].get_count()))

# Find Median from BQM data for largest ground net (called basis_ground)
basis_ground = max(gnd_count,key=operator.itemgetter(1))[0]

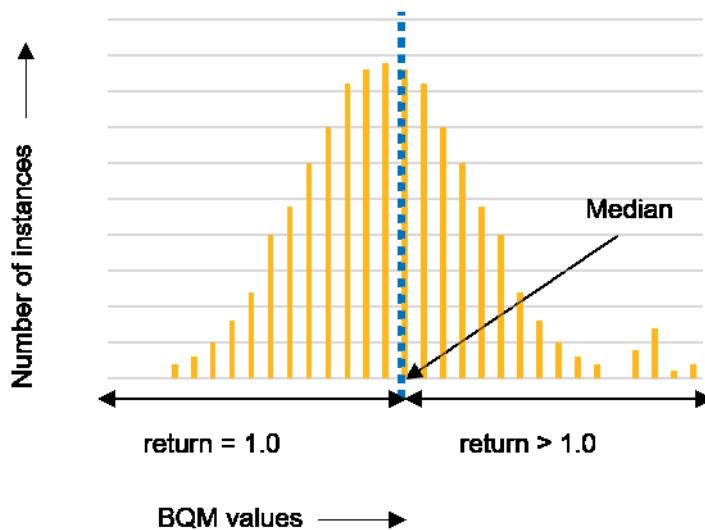
# Return 1 if value < median OR return Rgradient if value > median
def _mod_by_median(values):
    if values[0] < bqm_data['stats_'+basis_ground.get_name()]['median']:
        return 1.0
    else:
        return values[0]/(bqm_data['stats_'+basis_ground.get_name()]['median'])

# Generate R Gradient Heatmaps, Rgradient = resistance_score_instance_pin / 
median_score_basis_ground
uv_rgradient = db.create_user_view(tag='uv_rgradient')
for a_pgnet in bqm_inst:
    uv_rgradient['hm_bqm_r_gradient_'+a_pgnet.get_name()] =
        hm_oper.do_instance_heatmap_custom_operations(dv,[bqm_inst[a_pgnet]],_mod_by_median)
    uv_rgradient[a_pgnet.get_name()+'_stats'] =
        norm.get_heatmap_min_max_median(dv,uv_rgradient['hm_bqm_r_gradient_'+a_pgnet.get_name()])

```

The `_mod_by_median()` function assigns a weight to each BQM value for an instance. The function assigns a weight of 1.0 to each BQM value that is less than the median value, and a weight greater than 1.0 to BQM values greater than the median. The resistance gradient heatmap is generated from the weights and the BQM heatmap, and stored in UserView as `uv_rgradient`.

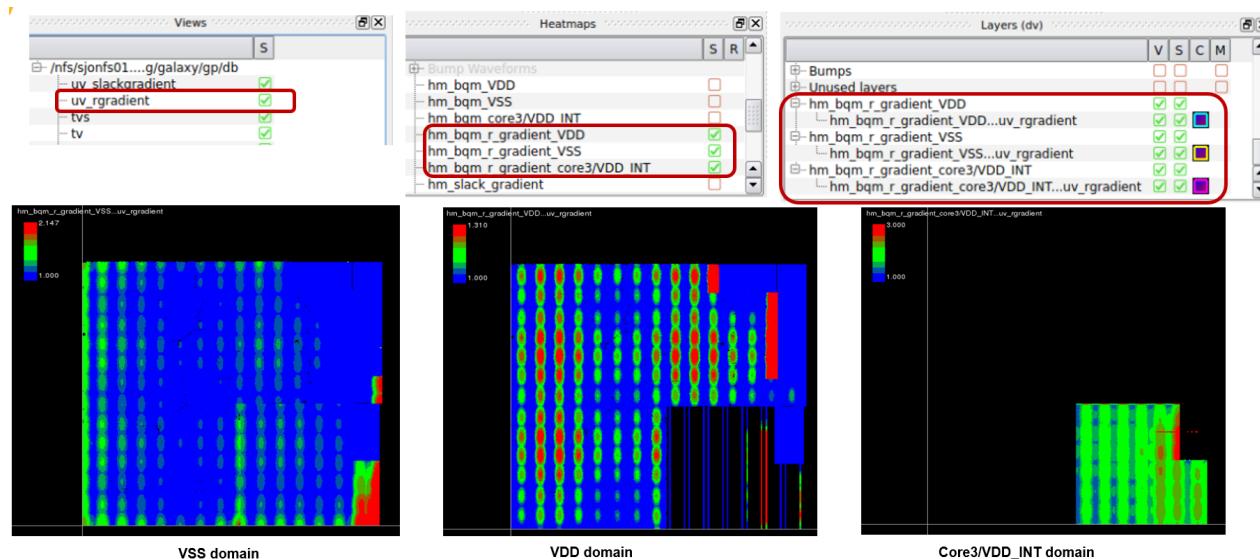
The following figure shows an example BQM spread of the design instances about the median.



4.7.1.1. Viewing Resistance Gradient Heatmaps

A resistance gradient heatmap is a pictorial representation of weighted scores of instance BQM data, and therefore, better represents grid weakness in terms of pin node voltages.

For information about how to view a BQM instance voltage drop heatmap in GUI, see [Viewing BQM Analysis Results](#) on page 68.

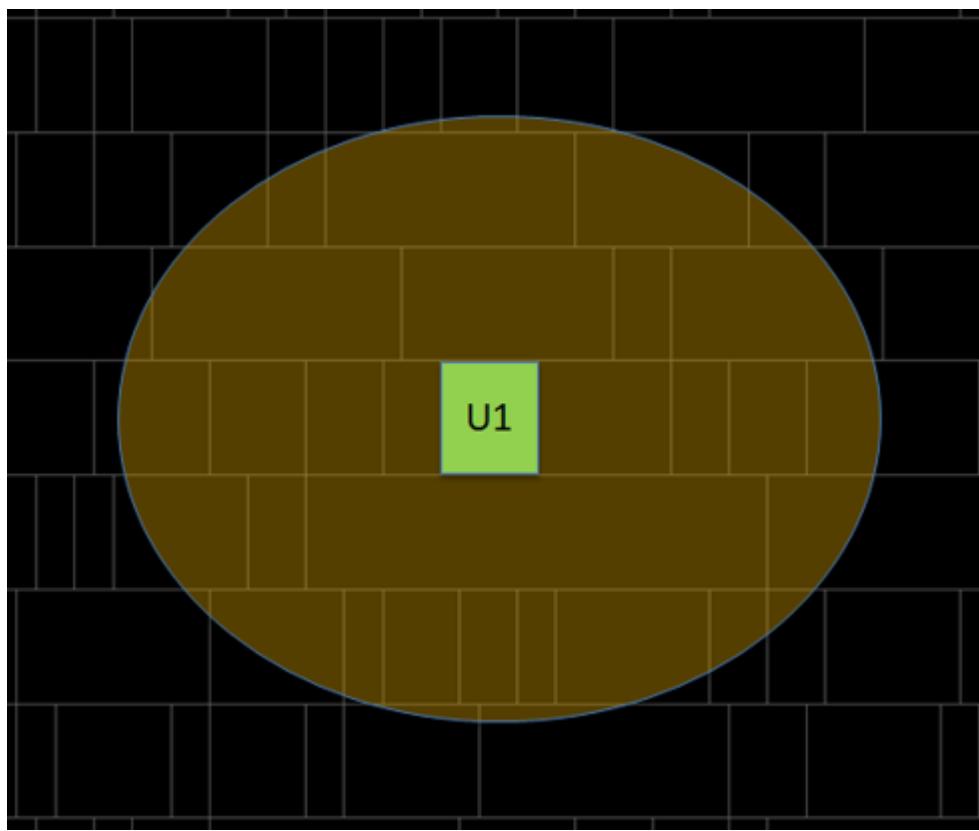


4.7.2. Analyzing PeakTW Currents

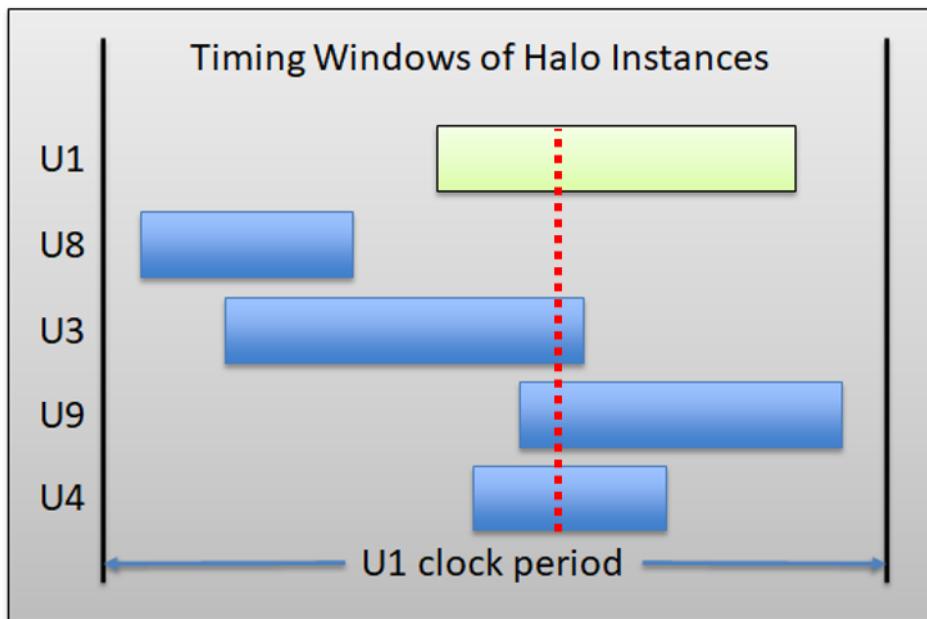
The method of PeakTW analysis uses instance peak current values to determine grid weakness. Currents are calculated for all instances without using switching scenarios and represented as heatmaps.

When multiple instances of a design switch together, they affect the voltage drop across one another. To estimate the impact of switching of other instances on each switching instance of the design, the tool considers a halo around each instance.

The following figure shows an instance, U1, with its halo. The instances in this halo are considered to impact the voltage drop across U1 when it is switching.



However, not all instances in the halo switch at the same time as U1. The switching time periods (or timing windows) of some instances overlap with that of U1. For example, the following figure shows that switching timing windows of instances U8 and U1 do not overlap. The switching timing windows of instances U3, U9, and U4 overlap with that of U1. The timing window information is taken from STA input files. STA files with good coverage are important for accuracy of results.



The tool performs the following steps during PeakTW analysis:

- 1.** Determines the peak current waveform for each instance of the design based on the slew, load, and voltage for each instance.

These are taken from the APL or CCS dynamic power models. If these models are not available, the peak current value is determined from the Liberty energy values.

- 2.** Identifies a halo region around each instance of the design.
- 3.** Determines the overlap of timing windows for all instances in the halo. The timing window information is taken from STA input files.
- 4.** Adds the peak currents of the instances with timing window overlap in the halo. This step determines the halo current of each instance of the design.

Understanding PeakTW Score

- The peak current heatmap of timing window overlap represents the simultaneous switching score for all instances of the design. Simultaneous switching is the primary cause of local DVD peak current. The score is not based on simulation, but is a probability score.
- The peakTW score is driven by the STA file. So, you can create a different peakTW heatmap for each functional mode of the design.
- The heatmap is independent of the power grid design.

4.7.2.1. Creating PeakTW Heatmaps

To generate current heatmaps representing simultaneous switching of instances, use the `generate_peak_power_views` command:

```
generate_peak_power_views()
```

The following example shows how to use the `generate_peak_power_views` command to generate the halo current for each switching instance of a design and store these values as a heatmap. A heatmap is generated for each power rail and stored as `peakTW_data` in the `UserView` (using the `create_user_view` command). The heatmap represents the simultaneous switching score for each instance of the design. A different heatmap is created for each timing mode.

```
import generate_peak_power_views as peakTW
peaktw_dict = peakTW.generate_peak_power_views(timing_view=tv,
                                                external_parasitics=evx,
                                                per_domain_maps=True)

peaktw_data = db.create_user_view(tag='peakTW_data')
for map_type in peaktw_dict:
    for pgnet in peaktw_dict[map_type]:
        if map_type == 'peak_current_heatmap_TW_overlap':
            peaktw_data[map_type+'_'+pgnet.get_name()] =
peaktw_dict[map_type][pgnet]
```

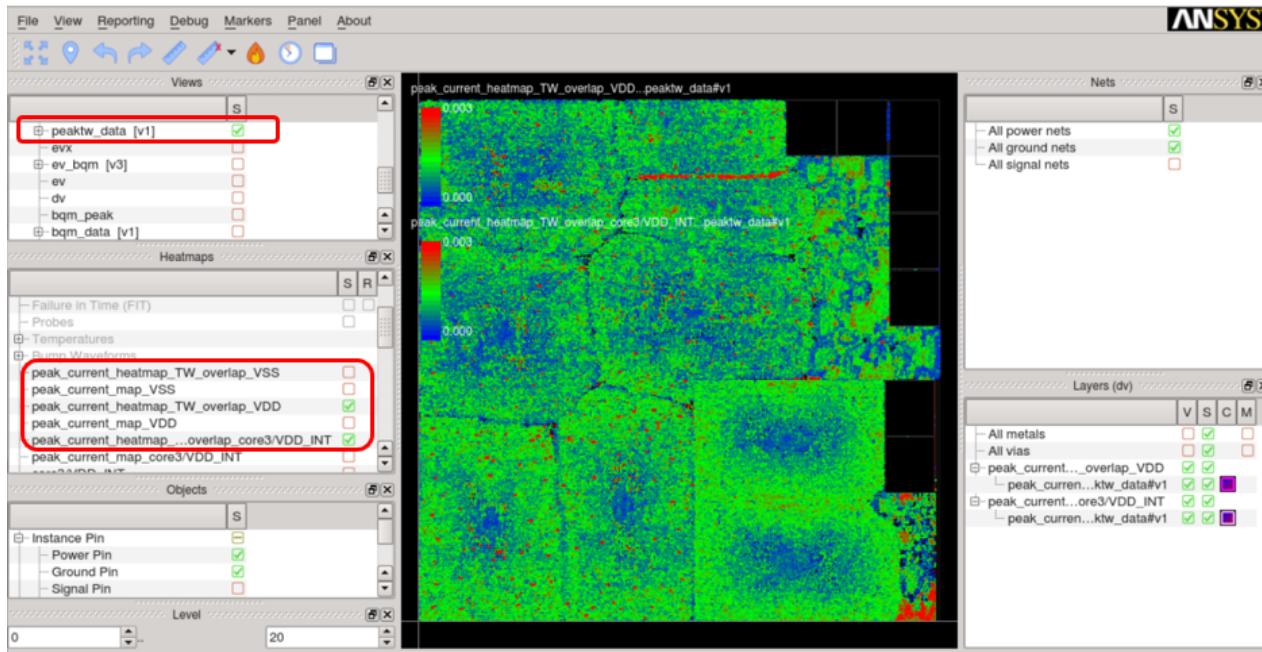
For more details, use the `help` command at the tool command prompt:

```
help(generate_peak_power_views)
```

4.7.2.2. Viewing PeakTW Heatmaps

To view the peakTW heatmap in GUI, follow these steps:

1. Select the tag specified with the `create_user_view` command (**peakTW_data**) in **Views**.
2. Select the heatmap for a particular rail (**peak_current_heatmap_TW_overlap_VDD**) in **Heatmaps**.
3. Select the layers to view in **Layers**.



4.7.3. Analyzing Slack

To perform slack analysis, follow these steps:

1. Read in the input slack data from timing files using the `create_timing_view` command as shown in the following example:

```
tv_slack = db.create_timing_view(dv, **tv_slack_args)
```

2. Run the following example script to generate the slack heatmap and the gradient heatmap.

```
hm_slack=create_slack_hm_tv.create_inst_slack_hm(tv_slack)
slack_data = db.create_user_view(tag='slack_data')
slack_data['hm_slack'] = hm_slack
slack_data['stats']= norm.get_heatmap_min_max_median(dv,hm_slack)

def _slack_mod_by_median(values):
    if values[0] < slack_data['stats']['median']:
        return 1.5
    else:
        return 1.0
uv_sgradient = db.create_user_view(tag='uv_slackgradient')
uv_sgradient['hm_slack_gradient'] =
hm_oper.do_instance_heatmap_custom_operations

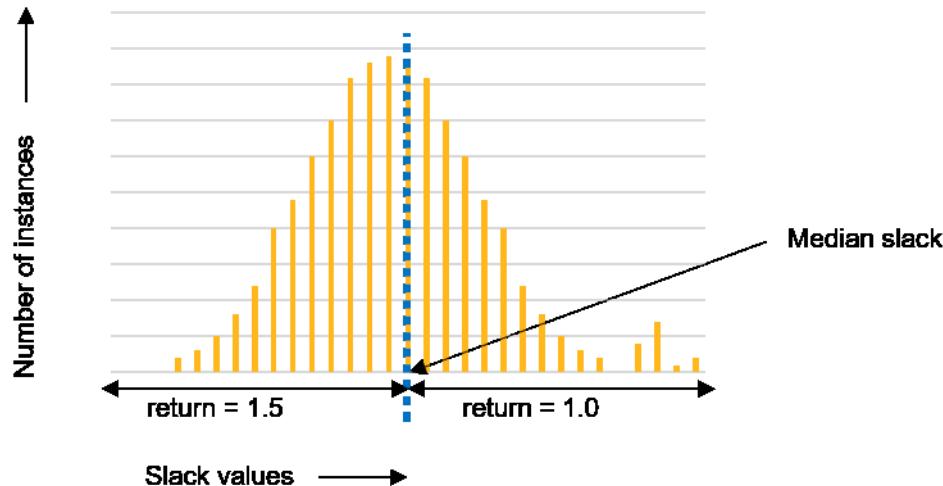
(dv,[slack_data['hm_slack']],_slack_mod_by_median)
```

```
uv_sgradient['stats'] =
norm.get_heatmap_min_max_median(dv,uv_sgradient['hm_slack_gradient'])
```

In the script, the `create_inst_slack_hm` command generates the slack heatmap for each design instance by reading the data from TimingView. The slack heatmap is stored in UserView as `slack_data`.

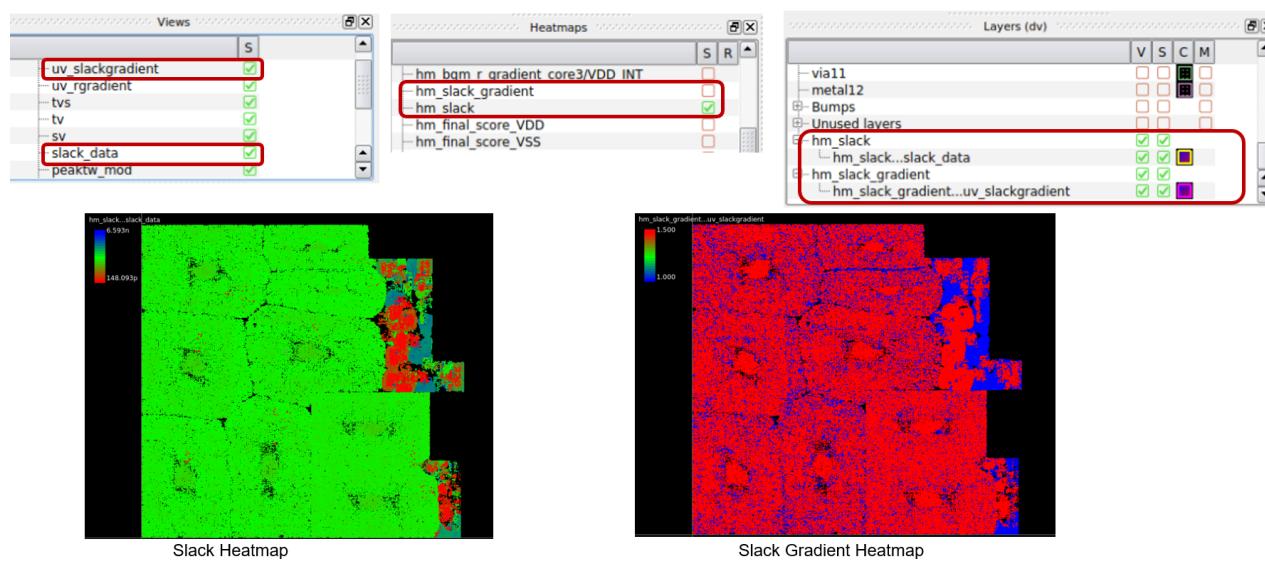
The `_slack_mod_by_median()` function assigns a weight to each instance slack. The function returns a weight of 1.5 for each instance slack that is less than the median slack, and a weight of 1.0 for each instance slack that is greater than the median slack. The slack gradient heatmap is generated from the weights and the slack heatmap, and stored in UserView as `uv_slackgradient`.

The following figure shows an example slack spread of the design instances about the median slack.



4.7.3.1. Viewing Slack Gradient Heatmaps

A slack gradient heatmap represents weighted scores of instance slacks, and therefore, shows better convergence for instances with low slacks than the slack heatmap.



4.7.4. Combining Heatmaps

You can use heatmap operations to consolidate grid check (such as BQM), local simultaneous switching (peakTW), and slack data, to statistically analyze grid weakness. You have the flexibility to customize scripts for the consolidated score, and resistance and slack gradients.

To consolidate the multiple heatmaps, follow these steps. This flow enables you to find grid weakness in early design stages by combining BQM, peakTW, and slack data.

1. Determine the final score per instance by multiplying the peakTW, resistance gradient, and slack gradient data.
2. Normalize the final score to between 0 and 100.
3. Store the per voltage rail normalized score heatmap in UserView.

The following example shows a typical script to consolidate multiple heatmaps.

```
#Custom Heatmap Operations (Multiply peakTW value, BQM rgradient, slack
gradient)
def _combine_peaktw_r_slack(values):
    return values[0]*values[1]*values[2]
peaktw_mod = db.create_user_view(tag='peaktw_mod')
mod_dict = dict()
for uvkey in uv_rgradient.keys():
    if 'stats' in uvkey:
        continue
    peaktw_key = 'peak_current_heatmap_TW_overlap_'+uvkey
    peaktw_key = peaktw_key.replace('hm_bqm_r_gradient_','')
    mod_uvkey = 'hm_final_score_'+uvkey.replace('hm_bqm_r_gradient_','')
    mod_dict[mod_uvkey] = hm_oper.do_instance_heatmap_custom_operations

(dv, [peaktw_data[peaktw_key],uv_rgradient[uvkey],uv_sgradient['hm_slack_gradient']],

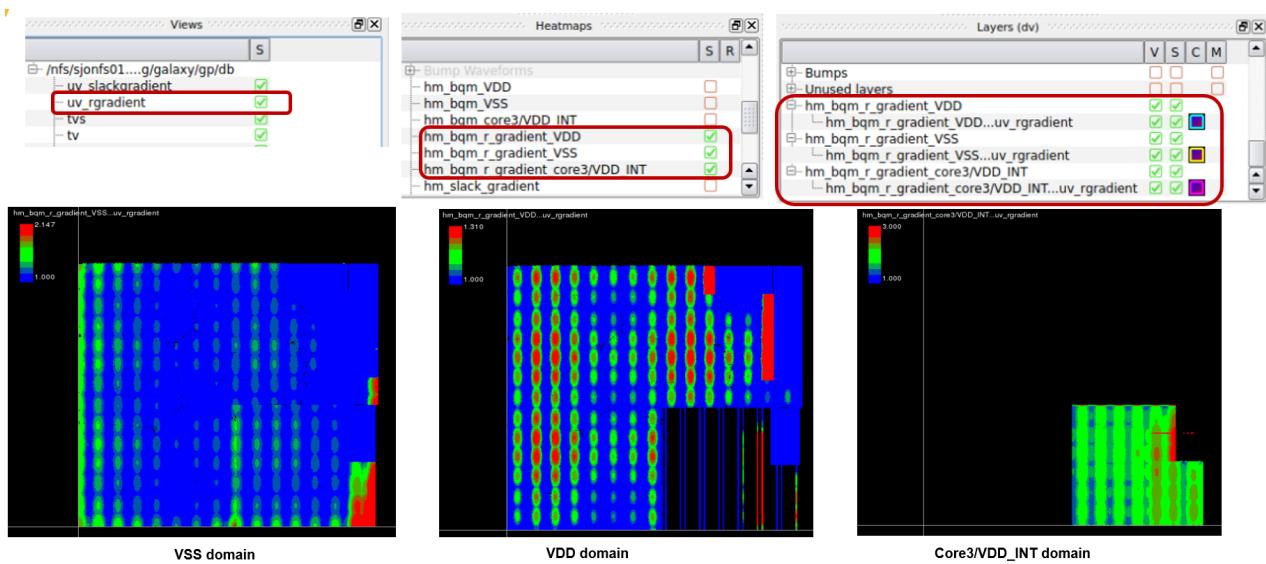
    _combine_peaktw_r_slack)

#Normalized Heatmaps (Normaize data from 0->100)
norm_mod_dict = norm.normalize_heatmaps(dv=dv, raw_heatmap_dict=mod_dict,
                                         norm_range=100.0)
for mod_key in norm_mod_dict:
    peaktw_mod[mod_key] = norm_mod_dict[mod_key]
    peaktw_mod[mod_key+'_stats'] =
        norm.get_heatmap_min_max_median(dv,norm_mod_dict[mod_key])
```

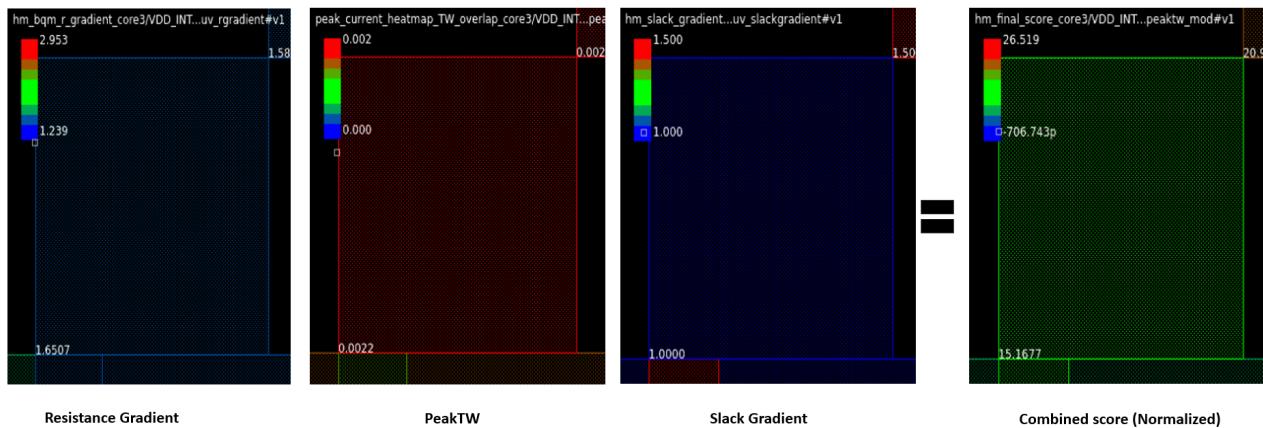
4.7.4.1. Viewing Consolidated Heatmaps

To view a consolidated heatmap, make the required selections in **Views**, **Heatmaps**, and **Layers**.

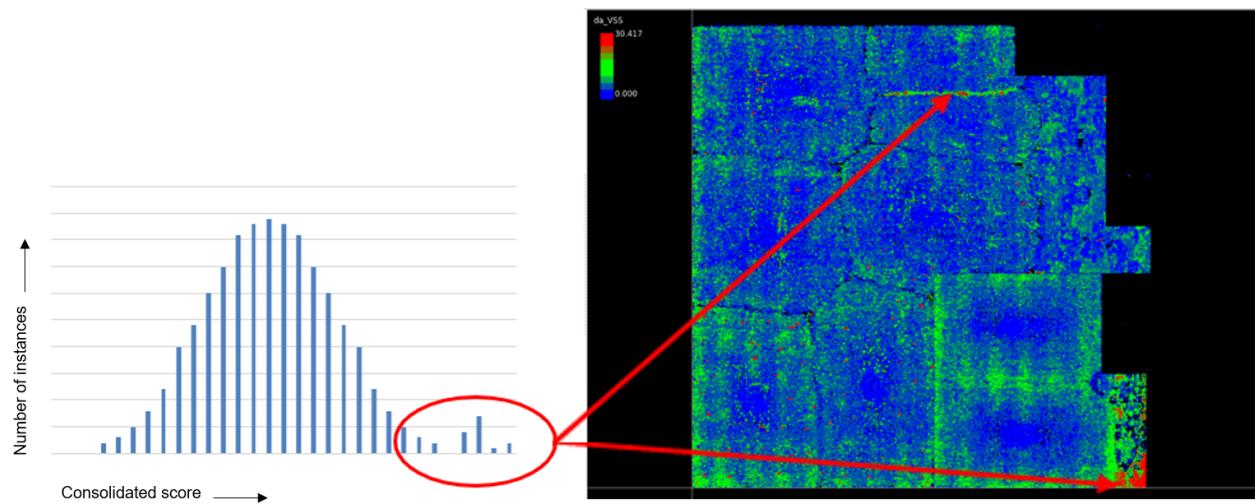
In the following figure, the red regions show grid weakness due to resistance (voltage drop), simultaneous switching (peakTW), and slack combinations.



The following figure shows the consolidated and individual heatmaps of a single instance. Though the PeakTW score shows grid weakness, the consolidated score does not show any grid violations. In another case, the individual scores might be showing a grid without any issues while the consolidated score shows grid weakness.



This analysis also generates worst case or outlier data that you should fix. The following figure shows an example spread of the design instances with respect to the consolidated score (of peakTW, resistance gradient, and slack gradient data).



5: Computing Power, Static IR Drop and EM Analysis

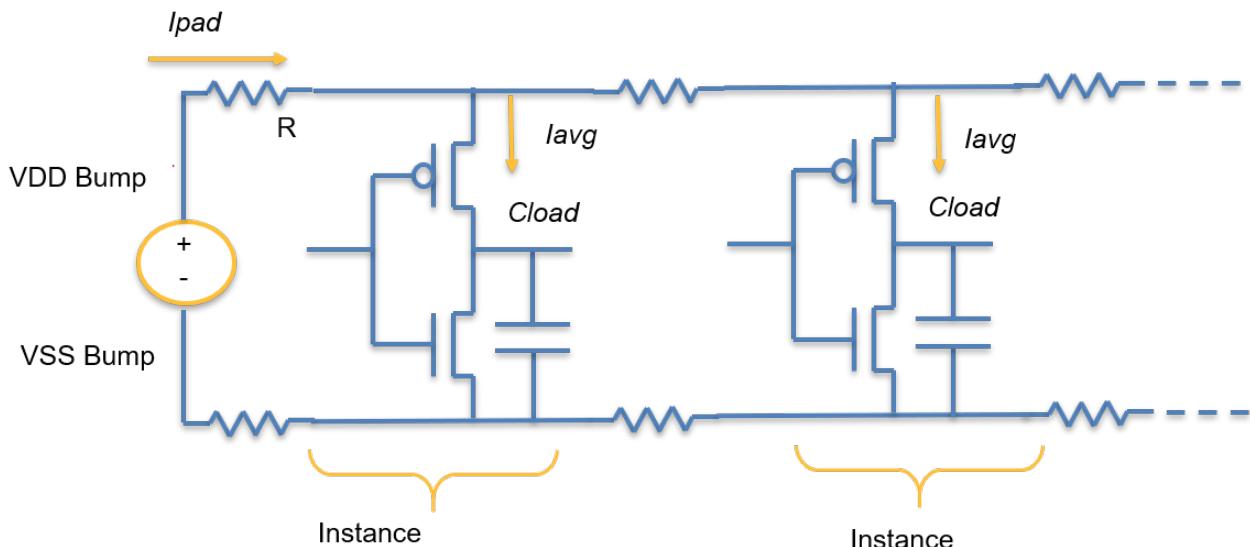
Static voltage drop analysis is a voltage drop estimation method to identify design weakness. In this method, the average voltage drop across PG pins of design instances is determined. The voltage drop values represent the strength or weakness of the PG grid. The method is used to detect floorplan issues such as macro placement, switch placement, and power clustering, grid issues such as shorts, disconnects, and missing vias, and bump placement issues such as missing bumps.

The following topics describe how to perform static voltage drop and electromigration analysis by using the RedHawk-SC tool:

- [Static Analysis Methodology](#) on page 103
- [Static Analysis Flow](#) on page 104
- [Setting Switching Activity](#) on page 105
- [Computing Power](#) on page 112
- [Scaling Power](#) on page 120
- [Performing Static Analysis](#) on page 125
- [Reporting Static Analysis Results](#) on page 126
- [Viewing Static Analysis Results in GUI](#) on page 129
- [Generating Histograms](#) on page 136
- [Static Electromigration Analysis](#) on page 138

5.1. Static Analysis Methodology

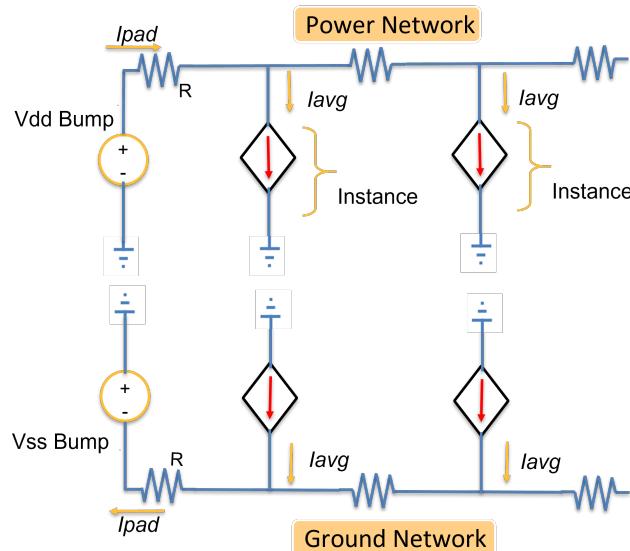
For static voltage drop analysis, the tool converts the on-chip power-ground network into a mesh of resistors during the ExtractView stage. The following circuit is a simple example of such a mesh.



The tool replaces each design instance (current sink) with an equivalent constant current sink. To do this, the tool first computes the average power consumed by each instance and then determines the average DC current:

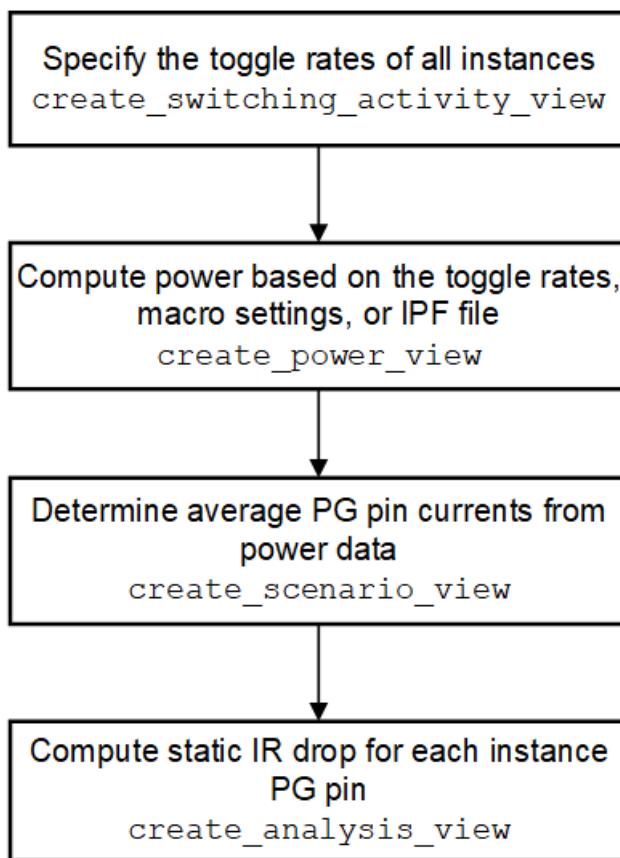
$$I_{avg} = P_{avg} / V_{supply}$$

The power network and the ground network are decoupled and the static voltage drop at each node is computed by using Ohm's law. The total static voltage drop for each instance (V_{static}) is computed by adding the static voltages at the power and the ground nodes. The following is a simple example of a decoupled circuit.



5.2. Static Analysis Flow

The following figure shows the high-level flow of RedHawk-SC static voltage drop analysis and the main commands to perform these steps. You must create the base views before performing these steps. See [Creating Base Views](#) on page 32.



5.3. Setting Switching Activity

Setting switching activity in a design is a prerequisite to [Computing Power](#) on page 112 for static analysis when complete input power data, such as Instance Power Files (IPFs), are not available.

If vector-based switching activity information (SAIF, VCD, or FSDB files) is available, use these to configure the switching activity of your design. In the absence of pre-existing switching activity information, define these using vectorless methods, such as design, block, and instance-level toggle rates. You have the flexibility to use vector-based and vectorless methods for different blocks of the same design.

To set the design activity, use the `create_switching_activity_view` command. The following sections describe the methods to set and report switching activity.

- [Defining Activity](#) on page 105
- [Loading SAIF Files](#) on page 108
- [Loading VCD or FSDB Files](#) on page 109
- [Using Vectorless and Vector-Based Methods Together](#) on page 111
- [Reporting Switching Activity Values](#) on page 112

5.3.1. Defining Activity

You must define activity when input switching activity data (SAIF, VCD, or FSDB files) are not available. The activity at the output of a gate is determined by the propagation of activity from its inputs. Because static power is estimated from the switching activity data, the quality of static voltage analysis results depend on the quality of the switching activity data.

Therefore, the `create_switching_activity_view` command has various settings to control activity propagation. This topic includes the following sections:

- [Measuring Activity Propagation](#) on page 106
- [Specifying Activity Propagation](#) on page 106
- [Specifying Default Activity](#) on page 107
- [Specifying No Propagation](#) on page 108

Measuring Activity Propagation

The tool uses the following measures of activity for a signal at pin X:

- One probability, that is, fraction of the time the signal is high.
- Toggle rate, that is, the number of transitions per clock cycle.

Depending on the `create_switching_activity_view` command settings, the tool can propagate the following through a design:

- Number of transitions per second
- Probabilities of signals being high and low
- Constants
- Clock domain information

If the input STA file has missing information, you can propagate the clock information to identify whether an instance is part of the clock network and determine the frequency for the instance.

Specifying Activity Propagation

To define activity propagation, set the `clock_precedence` and `activity_precedence` keys in the `object_settings` dict under the `settings` argument of the `create_switching_activity_view` command, as shown in the following example:

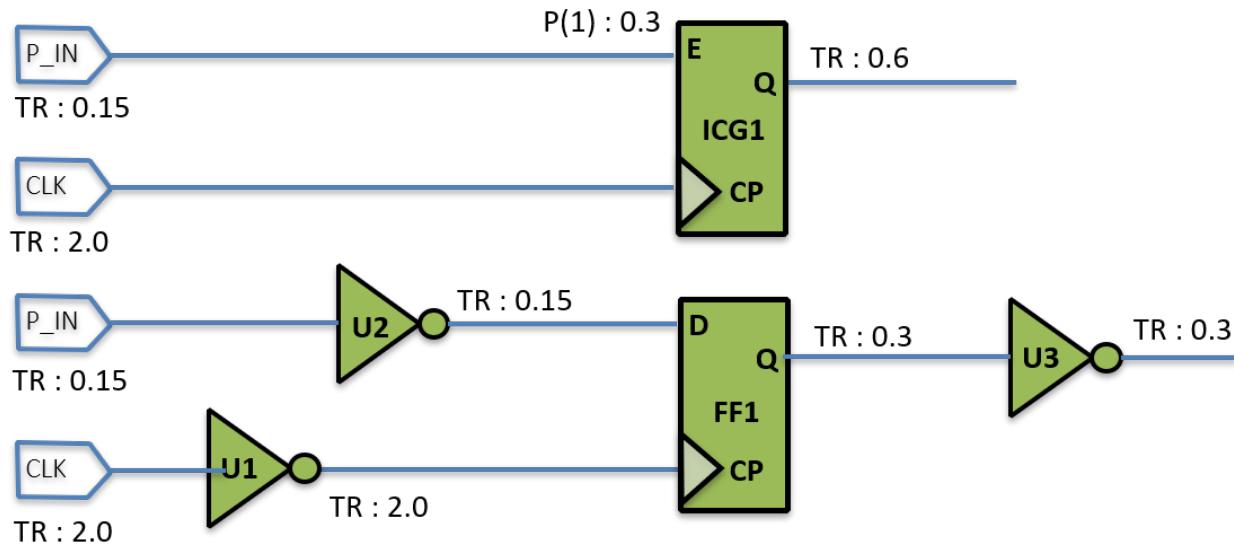
```
swa_settings = {
    object_settings = { 'design_values' : {
        'clock_pin_toggle_rate' : 2.0,
        'sequential_output_pin_toggle_rate' : 0.3,
        'icg_output_pin_toggle_rate' : 1.0,
        'icg_enable_pin_one_probability' : 0.3,
        'combinational_pin_toggle_rate' : 0.10,
        'clock_precedence': ['sta', 'propagated'],
        'activity_precedence': ['propagated'],
    ...
    }
}
```

The `clock_precedence` key is set to the default `['sta', 'propagated']` meaning instances with no STA information use the propagated clock period. This key is only supported under the `design_values` scope key.

The `activity_precedence` key is also set to the default `['propagated']`. In this case, the `constant_propagation` key is not set (that is, defaults to `auto`) meaning logic constants and set case analysis values are propagated because the `activity_precedence` key is set to `'propagated'`.

Note: SAIF, VCD, or FSDB data have higher priority. The tool uses the `activity_precedence` setting only when SAIF, VCD, or FSDB data are not available.

The following schematic example shows how the specified values are applied.



For example, the ICG output toggle rate is considered as 0.6 to account for propagation, that is, the `clock_pin_toggle_rate` of 2.0 is multiplied by the `icg_enable_pin_one_probability` of 0.3. The tool ignores the `icg_output_pin_toggle_rate` value even when you specify the same due to propagation.

Specifying activity data in `design_values` applies the data to the entire design. You can specify activity data at the block, cell, and instance levels also. If you define activity data at multiple levels, the lower-level specification overrides the higher-level specification. In the following example, activity is propagated in all blocks of the design except for one block that has default activity.

```

object_settings = {
    'design_values' : {
        'activity_precedence' : ['propagated'],
        'block_values' : [
            {
                'patterns' : 'u23',
                'activity_precedence' : 'default_activity'
            }
        ]
    }
}
settings = {object_settings=object_settings, ...}
create_switching_activity_view(tv, settings=settings, ...)

```

For more information, see [help\(SeaScapeDB.create_switching_activity_view\)](#).

Specifying Default Activity

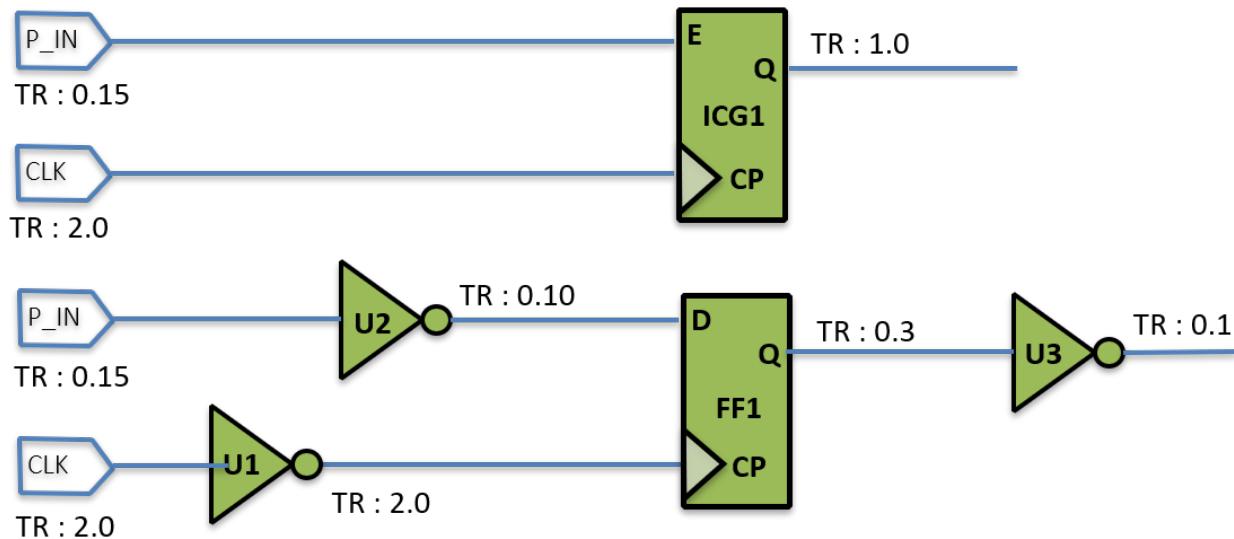
To define the default activity, set the `activity_precedence` key to `default_activity` in the `object_settings` dict under the `settings` argument of the `create_switching_activity_view` command, as shown in the following example:

```

swa_settings = {
    'object_settings' : {
        'design_values' : {
            'clock_pin_toggle_rate' : 2.0,
            'sequential_output_pin_toggle_rate' : 0.3,
            'icg_output_pin_toggle_rate' : 1.0,
            'icg_enable_pin_one_probability' : 0.3,
            'combinational_pin_toggle_rate' : 0.10,
            'clock_precedence' : ['sta', 'propagated'],
            'activity_precedence' : ['default_activity'],
        }
    }
}

```

The following schematic example shows how the specified values are applied.



The specified output pin toggle rates are directly applied. Only the constants and clock information are propagated. For example, the ICG output toggle rate is 1.0 and inverter U3 toggle rate is 0.1.

Specifying No Propagation

To define no propagation, the `constant_propagation` key is set to `off` meaning constants are not propagated.

The `clock_precedence` key is set to `sta` meaning instances without STA information use the default clock period. Because clock information is not propagated, you can create the TimingView for this SwitchingActivityView without graph object for propagation (using `logic_graph` set to `False`).

The `activity_precedence` key is set to `default_activity`. A no propagation specification is useful for large designs as it saves run time.

```

settings={ 'constant_propagation': 'off',
           'object_settings': { 'design_values' : {
               'clock_precedence' : ['sta'],
               'activity_precedence' : ['default_activity']
           }
       }
create_switching_activity_view(tv, settings=settings, options=options,
tag='swa')

```

5.3.2. Loading SAIF Files

A Switching Activity Interchange Format (SAIF) file contains toggle data of instance pins including:

- T0: period of time a signal is at logic 0
- T1: period of time a signal is at logic 1
- TC: toggle count over the duration of simulation
- The duration of simulation, timescale, and direction: forward or backward

To load one or more SAIF files to specify design activity, use the `saif_files` argument with the `create_switching_activity_view` command. The `saif_files` argument takes in a list of dicts. Each dict is for one SAIF file, and contains the following keys:

Key	Description
<code>file_name</code>	(Required) Path to the SAIF file
<code>cell_name</code>	(Optional) Name of the DEF cell for which to apply the file
<code>instances</code>	(Optional) Hierarchical instance object for which to apply the file
<code>preamble</code>	(Optional) Name prefix to remove from identifiers in the file

The following example shows how to read in a SAIF file to specify activity.

```
saif_files = [ {'file_name' : 'example.saif',
                'instances' : [Instance('')],
                'preamble' : 'sys/block1'}]
```

On finding invalid SAIF syntax, such as empty instance names, the tool continues to run and writes out appropriate error messages in log files.

Annotating Nets With SAIF Data

To annotate design nets with SAIF data when annotation on design pins fails, set the `annotate_net_in_saif` key under the `settings` dict of the `create_switching_activity_view` command to `True`. The default is `False`.

5.3.3. Loading VCD or FSDB Files

A Value Change Dump (VCD) file is generated by logic simulation programs and is stored in a text format. It contains waveform data of each net and instance pin of the design. A Fast Signal Database (FSDB) file also stores simulation waveform data in a binary format. An FSDB file size is much smaller than VCD.

Use either of the two methods to load VCD or FSDB files into the `SwitchingActivityView`:

- Directly load VCD or FSDB files into the `SwitchingActivityView`.

In the `SwitchingActivityView`, the tool reads only the number of rise transitions , number of fall transitions, and states with constant signals from VCD or FSDB files. So, directly loading a VCD or an FSDB file in `SwitchingActivityView` is faster than loading the same file in `SwitchingActivityView` from the `ValueChangeView`.

- Load VCD or FSDB files into the `SwitchingActivityView` from the `ValueChangeView`.

The `ValueChangeView` stores the complete information from VCD or FSDB files including time of events. Use this method when the `ValueChangeView` is available and when you plan to perform dynamic voltage drop analysis after static analysis.

Directly Loading VCD or FSDB Files

To directly load one or more VCD or FSDB files to specify design activity, use the `vcd_files` argument with the `create_switching_activity_view` command. The `vcd_files` argument takes in list of dicts. Each dict is for one VCD or one FSDB file, and contains multiple keys. The relevant keys are the following:

Key	Description
file_name	Path to the VCD file
instances	(Optional) Hierarchical instance name to apply the VCD file data
cell	(Optional) The VCD file data applies to all instances of this cell
preamble	(Optional) Name prefix to remove from identifiers in the file
start_time	(Optional) Start time value to consider in VCD
stop_time	(Optional) End time value to consider in VCD
top_only	(Optional) Only read signals at top level of hierarchy

The following example shows how to read in a VCD file for a block of the design to specify activity.

```
vcd_files = [ {'file_name' : 'block2.vcd',
               'preamble' : 'sys/',
               'instances' : 'top/block2',
               'start_time': 1e-9,
               'stop_time' : 6e-9}]
```

If you do not specify the `start_time` and `stop_time` keys as shown in the following example, the tool uses start time and end time from the input VCD or FSDB file.

```
vcd_files_def = [
    {'file_name': './design_data/vcd/us_top.vcd',
     'preamble' : 'top/us_top/'},
    ]]
```

The following example shows how to input multiple slices from a VCD or an FSDB file to the `SwitchingActivityView`. The `time_slices` key under the `vcd_files` argument specifies the time slices.

```
vcd_files_swa = [ {'file_name' : design_dir + 'test_swa3.vcd',
                   'preamble' : 'top/',
                   'time_slices':[{ 'slice_name': 'slice0',
                                   'start_time': 2.5e-10, 'stop_time' : 2.25e-9},
                                 { 'slice_name': 'slice1',
                                   'start_time': 2.28e-9, 'stop_time' : 3.8e-9}],
                   }]
create_switching_activity_view(..., vcd_files=vcd_files_swa)
```

The following example shows how to use the same FSDB or VCD file for different parts of a design. The FSDB file, `vless`, applies to all the hierarchical instances of `cpu1` and `cpu2`:

```
vcd_files_swa= [ {'file_name': my_data/vectors/vless.fsdb',
                  'instances': ['cpu1','cpu2'],
                  'preamble' : 'top/',
                  'start_time' : 0,
                  'stop_time' : 5e-10}]
```

Loading a ValueChangeView

To load activity stored in the `ValueChangeView` into the `SwitchingActivityView`, use the `value_change_data` argument with the `create_switching_activity_view` command. The `value_change_data` argument takes in list of dicts. Each dict is for one `ValueChangeView` and contains the following keys:

Key	Description
'view'	ValueChangeView tag name, whose data you want to apply
'slice_name'	Time slice of the specified ValueChangeView
'time_delay'	Amount of time to shift events/waveforms. Optional.
'time_scale_factor'	All time values for signal changes will be multiplied by this factor. Optional.

The following example shows how to read in the value change information from a `ValueChangeView` to specify activity.

```
value_change_data = [ {'view' : vcv, 'slice_name' : 'all'} ]
```

5.3.4. Using Vectorless and Vector-Based Methods Together

You can use a combination of vectorless and vector-based methods to set switching activity in a design. Some blocks can have vector-based activity and some blocks can have vectorless activity.

The following example shows the use of both the `vcd_files` argument and the `activity_precedence` key in the design.

```
swa_mixed = db.create_switching_activity_view(timing_view=tv,**swa_mixed_args)

swa_mixed_args = dict(
    vcd_files = vcd_files_swa,
    settings = swa_settings,
    tag='swa_mixed',
    options=options)

swa_settings={
    object_settings = { 'design_values' : {
        'clock_pin_toggle_rate' : 2.0,
        'sequential_output_pin_toggle_rate' : 0.15,
        'macro_output_pin_toggle_rate' : 0.15,
        'icg_output_pin_toggle_rate' : 1.0,
        'combinational_pin_toggle_rate' : 0.15,
        'activity_precedence': ['default_activity'],
        'clock_precedence': ['sta'],
        'clock_period' : 8e-09,           }
    }
}
```

5.3.5. Reporting Switching Activity Values

To query the attributes in the `SwitchingActivityView` database, use the `get_attributes` command as shown in the following examples.

In this example, '`source': 'default'` shows that the source of activity is the default toggle rate. For the output pin Z, the `Toggle Rate = toggles_per_second * clock_period = 0.15`.

In the next example, '`source': 'VCD/SAIF'` implies that activity is read from the input VCD or SAIF files.

```
swa_mixed.get_attributes(Instance("core0.regfile_program_memory.MUX2_X1_1377"))
{Pin('A'): {'clock_name': 'dco_clk',
            'clock_period': 7.99999973744548e-09,
            'one_probability': 0.5,
            'source': 'default',
            'toggles_per_second': 18750002.0,
            'zero_probability': 0.5},
 
Pin('Z'): {'clock_name': 'dco_clk',
            'clock_period': 7.99999973744548e-09,
            'one_probability': 0.5,
            'source': 'default',
            'toggles_per_second': 18750002.0,
            'zero_probability': 0.5}}
 
swa_mixed.get_attributes(Instance("core3/regfile_program_memory.MUX2_X1_3464"))
{Pin('Z'): {'clock_name': 'dco_clk',
            'clock_period': 7.99999973744548e-09,
            'one_probability': 0.570888876914978,
            'source': 'VCD/SAIF',
            'toggles_per_second': 67500000.0,
            'zero_probability': 0.429111123085022}}
```

5.4. Computing Power

You use the `create_power_view` command to compute the total power of a design. PowerView contains power per instance values and related information, such as load capacitance, transition time, toggle rate, frequency, and voltage levels. For details, see `help(<SeaScapeDB>.create_power_view)`.

The following sections describe how to determine the total power with different input types and for different cells.

- [Determining Power Using Switching Activity](#) on page 113
- [Determining Power Values from IPF](#) on page 114
- [Creating PowerView from Existing ScenarioView](#) on page 115
- [Determining Power of Macro Cells](#) on page 115

The following sections describe how to query and report power values:

- [Querying PowerView](#) on page 118
- [Reporting Power Calculation](#) on page 119

5.4.1. Determining Power Using Switching Activity

When you use switching activity to determine the total power, the tool computes the average power for static analysis by adding the leakage, internal, and switching power components:

$$P_{\text{total}} = P_{\text{leakage}} + P_{\text{internal}} + P_{\text{switching}}$$

To compute the total power of a design from switching activity, use the `create_power_view` command as shown in the following example:

```
pwr_mixed = db.create_power_view(switching_activity_view=swa_mixed,
                                  external_parasitics=evx, **pwr_mixed_args)

pwr_mixed_args = dict(
    settings = settings,
    tag = 'pwr_mixed',
    options = options)
```

The following topics describe the methods used by the tool to compute power components.

- [Reading Leakage Power](#) on page 113
- [Reading Internal Energy](#) on page 113
- [Computing Switching Power](#) on page 114

Reading Leakage Power

For each leakage input state of a design instance, the tool reads the leakage power value from the `cell_leakage_power` groups of the Liberty file. The leakage power for an input state is then multiplied by the probability of its occurrence, and all such terms are added to determine the average leakage power (P_{leakage}) of the instance.

If the leakage input state of an instance does not match the available Liberty `leakage_power` states, the tool uses the state-independent power defined using the `cell_leakage_power` Liberty attribute.

Reading Internal Energy

Internal power dissipation occurs due to the charging or discharging of internal capacitance of switching instances.

For each rise and fall transition of the input and the output pins of a design instance, the tool reads the internal energy values from the pin `internal_power` groups of the Liberty file. A Liberty `internal_power` group can include `rise_power` and `fall_power` syntax to store the corresponding internal energy values. The internal energy of the given state is then multiplied by the probability of its occurrence, and all such terms are added to determine the average internal energy (E_{internal}) of the instance.

The tool then determines the internal power using the following formula:

$$P_{\text{internal}} = \frac{1}{2} (TR * f * E_{\text{internal}})$$

Note: In the Liberty files, the power characterization includes both rising and falling signals. Therefore, the internal energy measurement is halved because switching transition means either a rising or a falling signal.

TR is the toggle rate computed from the SAIF, VCD, or FSDB files and f is the frequency.

Computing Switching Power

The tool computes the switching power using the following formula:

$$P_{\text{switching}} = 1/2 (TR * f * C * V^2)$$

TR is the toggle rate computed from the SAIF, VCD, or FSDB files, or toggle settings specified with the `object_settings` dict in `SwitchingActivityView`.

f is the frequency.

C is the total load capacitance of the instance and is determined by adding the receiver input pin capacitance to the output capacitance read from the SPEF file.

V is the total supply voltage.

5.4.2. Determining Power Values from IPF

To read the power information from one or more IPFs, use the `power_files` argument with the `create_power_view` command. IPFs are written by power calculation tools. You can also use the `write_instance_power_file` command with a `PowerView` object to write IPFs. Both nine-column (detailed) and three-column IPF (total power per pin) types are supported.

Note: The IPFs should cover all the instances of the design. Instances missing from IPF files are not assigned power unless covered by `SwitchingActivityView` toggle information.

The `power_files` argument takes in a list of dicts. Each dict is for one IPF and contains the following keys:

Key	Description
<code>file_name</code>	Path to the IPF file
<code>instance_name</code>	Data will be applied to this hierarchical instance (optional)
<code>cell_name</code>	Data will be applied to all instances of this hierarchical cell (optional)
<code>scaling_factors</code>	A dict of type {<DomainNet#> scaling_factor} (optional). <DomainNet#> Specify domain net to be scaled by the scaling_factor. Use '*' to specify all the domains not specified.
<code>override</code>	Boolean, True means data in this file will be with higher precedence (optional). False by default. There can only be one power file with 'override' : TRUE
<code>type</code>	Type of data in this power file (optional). Possible values ['instance','cell']. The default is 'instance', which means the first column in power file is instance name, 'cell' means it is cell name. if 'cell' is used, 'scaling_factors' setting will be ignored, also 'instance_name' or 'cell_name' can not be defined.

The following example shows how to read in an IPF that applies to all instances of the design.

```
power_files = [
    {'file_name': '../design_data/top.ipf',
     'instance_name': '',
    }, ]
```

The following example shows how to assign power numbers based on pratio values to GDS cell instances. You should set the `pratio_power_assignment` key to `True`. The default is `False`. You can specify only one file with `pratio_power_assignment`. For a top-level hierarchical instance, the pratio file must not include the pratios of instantiated cells.

```
power_files = [
    {'file_name': 'gds_inst.pwr', 'pratio_power_assignment': True, }]
```

The following shows a snippet of a typical three-column IPF.

```
#instance_name total_power pin_name
ZBUF_INST3 4.12345654321e-07 VDD
ZBUF_INST4 5.12345654321e-07 VDD
...
```

The following shows a snippet of a typical nine-column IPF.

```
# instance_name pin_name voltage toggle_rate frequency total_power switching_power internal_power leakage_power
ZBUF_314_inst_65462 VDD 1.10000002384 0.15000000596 125000000.0 4.67768529688e-07 3.38084106488e-07 9.91261970285e-08 3.05582155136e-08
ZBUF_3658_inst_51786 VDD 1.10000002384 0.15000000596 125000000.0 1.3510846486e-07 8.57065742821e-08 3.81877960365e-08 1.12140927655e-08
ZBUF_5852_inst_51787 VDD 1.10000002384 0.15000000596 125000000.0 2.21814434553e-06 1.9550336674e-06 1.76988777412e-07 8.6121801246e-08
ZBUF_9013_inst_51856 VDD 1.10000002384 0.15000000596 125000000.0 1.68285282598e-06 1.54588008172e-06 1.06414510981e-07 3.05582155136e-08
inv drc cIn62326 VDD 1.10000002384 2.0 125000000.0 5.78630315431e-05 5.57769817533e-05 1.97122358259e-06 1.1482629958e-07
```

5.4.3. Creating PowerView from Existing ScenarioView

To specify a particular time window from an existing ScenarioView for which power is to be calculated and populated into PowerView, use the `create_power_view_from_scenario` function with the `start_time` and `end_time` arguments. For example,

```
pwr = db.create_power_view_from_scenario(
    scenario_views = [scn_high_power_10ns],
    start_time=4e-09,
    end_time=8e-09,
    options=None,
    tag = 'pwr'
)
```

5.4.4. Determining Power of Macro Cells

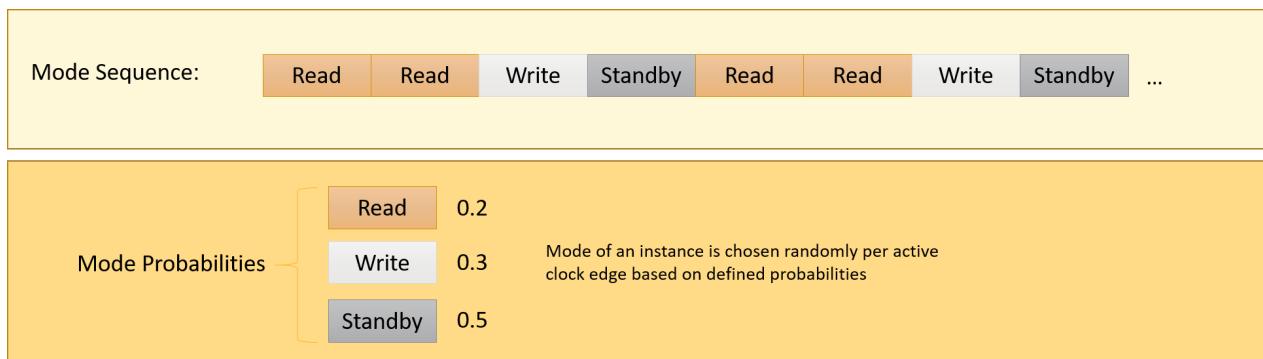
The current consumed by a macro cell depends on its mode of operation. To determine macro power consumption, annotate the macro modes (switching states) by using the `mode_control` key in the `object_settings` dict under the `settings` argument of the `create_power_view` command.

`mode_control` is a dict or a list of dicts and includes the following mutually-exclusive keys:

Key	Description
mode_sequence	<p>A sequence of mode names (a Python list) that is applied to each macro instance.</p> <p>In PowerView, the sequence starts repeating after all the modes in the sequence are applied when clock cycles are left.</p> <p>Mode sequence is typically used for cell or leaf instance level inputs where the exact macro instance and exact mode sequence is known.</p>
mode_probabilities	<p>Rather than the exact sequence, probability of occurrence of each mode is input. It is a python dict with keys as mode names, and values as their probabilities. Probability values for the modes must sum up to 1.0 for each scope. If the total probability is less than 1, <code>_leakage_only_mode_</code> is applied for the remaining probability. If the total probability exceeds 1, the tool automatically scales the values to 1.0. Mode probabilities is typically used for design or block level inputs where the exact macro instance and exact mode sequence is not known but a control on overall mode distribution is desired.</p>

PowerView computes the average power for a given mode sequence or mode probabilities, and assign this power to the macro cell. When you specify `mode_sequence`, each mode in the sequence is assigned equal probability and the average of all mode powers is computed. When you specify `mode_probabilities`, the probabilities are multiplied with the mode power values and then added to get the macro power.

You must specify either mode sequence or mode probabilities for a design, block, cell, or an instance. The following figure shows how to assign mode sequence or mode probabilities.



The following example shows how to assign a sequence to modes in `mode_control`.

```
'mode_control' :
    {'mode_sequence' : ['READ', 'WRITE', 'READ', 'STANDBY'], }
```

The following example shows how to specify the probabilities to the modes.

```
'mode_control' : {'mode_probabilities' : {'READ':0.3, 'WRITE':0.5,
    'STANDBY':0.2}, }
```

Implicit Modes

The previous section dealt with examples where the mode names in input files are known. When the current source is APL, the modes have names that you can use. When input is Liberty (NLPM or CCS), it might be difficult to interpret the modes. During DesignView generation, the tool identifies the following pre-defined modes for each cell based on the energy consumption. You can use these implicit energy modes to annotate mode_sequence or mode_probabilities in macro mode_control.

Energy Mode	Description
_low_energy_mode_	Lowest Energy Mode (in Liberty file)
_median_energy_mode_	Median Energy Mode
_high_energy_mode_	Highest Energy Mode
_leakage_only_mode_	Only Leakage is Consumed
_off_mode_	Off Mode, that is, not even Leakage

The following example shows how to use the implicit and explicit modes together. You can also use implicit modes when APL is present, for example, when several APL modes exist in the input current file and the highest energy mode is considered for mode_control annotation.

```
object_settings = {
  'leaf_instance_values' : [
    {'instances' : Instance('c1.pm'), 'mode_control' : { 'mode_sequence' :
      ['MEM_READ'], } },
    {'instances' : Instance('c2.dm'), 'mode_control' : { 'mode_sequence' :
      ['MEM_WRITE'], } },
    {'instances' : Instance('c3/dm'), 'mode_control' : { 'mode_sequence' :
      ['MEM_WRITE'], } },
    {'instances' : Instance('c1.dm'), 'mode_control' : { 'mode_sequence' :
      ['high_energy_mode'], } },
    {'instances' : Instance('c0.dm'), 'mode_control' : { 'mode_sequence' :
      ['leakage_only_mode'], } },
    {'instances' : Instance('c2.pm'), 'mode_control' : { 'mode_sequence' :
      ['low_energy_mode'], } },
    {'instances' : Instance('c3/pm'), 'mode_control' : { 'mode_sequence' :
      ['median_energy_mode'], } },
    {'instances' : Instance('c0.pm'), 'mode_control' : { 'mode_sequence' :
      ['high_energy_mode'], } },
  ]
}
```

Custom Modes

You can define a mode name and associate it with a when Boolean condition defined in the Liberty file. This is useful when current sources are CCS power or NLPM and the current for a specific Boolean condition is applied for analysis. For example, new_ram_mode_def is a list of custom-defined modes.

```
new_ram_mode_def = {
  'my_WRITE' : { 'when' : '!we&cs&!byp&!se' },
  'my_READ' : { 'when' : 'we&cs&!byp&!se' },
  'my_STANDBY' : { 'when' : '!cs' },
}
my_modes_def = [{ 'name' : 'my_ram_modes', 'modes' : new_ram_mode_def }]
```

You can now reference these modes in mode_control to annotate mode_sequence or mode_probabilities.

You must specify mode definition information in both `mode_control` of `object_settings` and with the `mode_definitions` key under the `settings` argument of the `create_power_view` command.

```
settings_macro = {
    object_settings_macro = {'cell_values': [{patterns': 'ram*',
        'mode_control': {
            'mode_sequence': ['my_READ', 'my_WRITE', 'my_READ', 'my_STANDBY'],
            'mode_definition': 'my_ram_modes'}}]}
    mode_definitions = 'my_modes_def'
}

pwr_args = dict(..., settings = settings_macro)
```

Different cells might have different `when` conditions to specify the same mode (such as READ). In such cases, multiple mode definitions can be defined and applied to different cell patterns.

Specifying Period for Mode Change of Macros Without Clock Pins

To set the period after which modes change for macro instances without clock pins, use the `period` key under the `mode_change` dict as shown in the example:

```
object_settings = {'leaf_instance_values': [{'instances':
    [Instance('I1/adsU1'), Instance('I2/adsU1'), Instance('I3/adsU1')], 'mode_control': {'mode_sequence': ['CYCLE1', 'CYCLE2', 'CYCLE3'],
    'mode_change': {'period': 8.45e-10}}}]}
```

5.4.5. Querying PowerView

You can use query commands to query information from PowerView as described in the following sections:

- [Querying PowerView Attributes](#) on page 118
- [Querying Total Power Per Net](#) on page 119
- [Querying Output Load Capacitance](#) on page 119

Querying PowerView Attributes

To query the PowerView attributes, use the `get_attributes` command as shown in the following examples.

In this example, '`source`': 'ModeControl' shows that the instance has mode control and the total power of this macro is computed based on the mode control annotated in PowerView `object_settings`. All the three components of power and the total power are reported separately.

```
pwr_mixed.get_attributes(Instance("core3/program_memory"))

{Pin('vdd'): {'clock_pin_power': 0.00016499549383297563,
    'frequency': 125000000.0,
    'internal_power': 0.0001634641521377489,
    'leakage_power': 8.49892785481643e-06,
    'source': 'ModeControl',
    'switching_power': 1.5313393078031368e-06,
    'toggle_rate': 0.60999995470047,
    'total_power': 0.00017349442350678146,
    'voltage': 1.100000023841858}}
```

In the following example, `'source': 'SwitchingActivity'` shows that the power is computed based on the toggle rates from the `SwitchingActivityView`.

```
pwr_mixed.get_attributes(Instance("core3/regfile_program_memory.MUX2_X1_3464"))

{Pin('VDD'): {'clock_pin_power': 0.0,
    'frequency': 125000000.0,
    'internal_power': 1.9928337735564128e-07,
    'leakage_power': 3.856697716742019e-08,
    'source': 'SwitchingActivity',
    'switching_power': 9.795139988000301e-08,
    'toggle_rate': 0.5399999618530273,
    'total_power': 3.3580175795577816e-07,
    'voltage': 1.100000023841858}}}
```

Querying Total Power Per Net

The following example uses the `get_total_power` command to report only the power components per net and the total power per net from the PowerView.

```
pwr_mixed.get_total_power()

{Net('core3/VDD_INT'): [{}{'leakage_power': 0.0035269377985969186,
    'internal_power': 0.027774976566433907,
    'total_power': 0.08402520697563887,
    'switching_power': 0.0527232950553298}],
Net('VDD'): [{}{'leakage_power': 0.011251124528294909,
    'internal_power': 0.04594817524332839,
    'total_power': 0.12869410832713513,
    'switching_power': 0.07149480845043055}]}}}
```

Querying Output Load Capacitance

The following example queries the total capacitance at each output pin of an instance from the PowerView.

```
pwr_mixed.get_output_total_capacitance(Instance("_56833"))

{Pin('Q'): 4.048312317425813e-14,
 Pin('QN'): 0.0}
```

5.4.6. Reporting Power Calculation

To report the power of each design instance and the power summary, use the `write_instance_power_report_and_summary` command as shown in the following example:

```
reports_dir = 'reports/powercalc'
gp_util.makedirs(reports_dir)
emir_reports.write_instance_power_report_and_summary
    (pwr, reports_dir+'/instance_power.rpt',
    reports_dir+'/power_summary.rpt')
```

The following is a typical report with PowerView attribute values of each design instance.

#	clock_pin_power is included in internal_power	pin	domain	frequency	toggle_rate	clock_pin_power	internal_power	leakage_power	switching_power	total_power	voltage	cell_name	instance
#				(Hz)		(W)	(W)	(W)	(W)	(W)	(V)		
VDD	VDD	1.25e+08	2.00	0		1.012e-05	4.593e-07	0.0001029	0.0001135	1.10	INV_X32	cts_inv_590761458	
VDD	VDD	1.25e+08	2.00	0		8.234e-06	4.593e-07	0.0001028	0.0001115	1.10	INV_X32	cts_inv_528960840	
VDD	VDD	1.25e+08	2.00	0		1.026e-05	4.593e-07	9.763e-05	0.0001084	1.10	INV_X32	cts_inv_527960848	
VDD	VDD	1.25e+08	2.00	0		1.146e-05	4.593e-07	9.52e-05	0.0001071	1.10	INV_X32	cts_inv_527960830	
VDD	VDD	1.25e+08	2.00	0		1.146e-05	4.593e-07	9.519e-05	0.0001071	1.10	INV_X32	cts_inv_526960820	
VDD	VDD	1.25e+08	2.00	0		9.955e-06	4.593e-07	9.606e-05	0.0001065	1.10	INV_X32	cts_inv_537661127	
VDD	VDD	1.25e+08	2.00	0		3.641e-06	2.297e-07	0.0001023	0.0001062	1.10	INV_X16	cts_inv_526960817	
VDD	VDD	1.25e+08	2.00	0		4.285e-06	2.297e-07	0.0001017	0.0001062	1.10	INV_X16	cts_inv_526160812	
VDD	VDD	1.25e+08	2.00	0		8.316e-06	4.593e-07	9.746e-05	0.0001062	1.10	INV_X32	cts_inv_530360856	
VDD	VDD	1.25e+08	2.00	0		1.012e-05	4.593e-07	9.414e-05	0.0001047	1.10	INV_X32	cts_inv_530260853	
VDD	VDD	1.25e+08	2.00	0		8.49e-06	4.593e-07	9.529e-05	0.0001042	1.10	INV_X32	cts_inv_528760838	
VDD	VDD	1.25e+08	2.00	0		4.687e-06	2.297e-07	9.916e-05	0.0001041	1.10	INV_X16	cts_inv_528460835	
VDD	VDD	1.25e+08	2.00	0		8.923e-06	4.593e-07	9.472e-05	0.0001041	1.10	INV_X32	cts_inv_588961440	
VDD	VDD	1.25e+08	2.00	0		1.129e-05	4.593e-07	9.174e-05	0.0001035	1.10	INV_X32	cts_inv_558561136	
VDD	VDD	1.25e+08	2.00	0		1.02e-05	4.593e-07	9.259e-05	0.0001032	1.10	INV_X32	cts_inv_526560816	
VDD	VDD	1.25e+08	2.00	0		8.797e-06	4.593e-07	9.374e-05	0.0001003	1.10	INV_X32	cts_inv_558261133	
VDD	VDD	1.25e+08	2.00	0		8.875e-06	4.593e-07	9.359e-05	0.0001028	1.10	INV_X32	cts_inv_528060831	
VDD	VDD	1.25e+08	2.00	0		9.046e-06	4.593e-07	9.281e-05	0.0001023	1.10	INV_X32	cts_inv_590461455	
VDD	VDD	1.25e+08	2.00	0		8.415e-06	4.593e-07	9.263e-05	0.0001015	1.10	INV_X32	cts_inv_589961450	
VDD	VDD	1.25e+08	2.00	0		1.055e-05	4.593e-07	9.04e-05	0.0001014	1.10	INV_X32	cts_inv_530760858	
VDD	VDD	1.25e+08	2.00	0		1.194e-05	4.593e-07	8.482e-05	0.0001008	1.10	INV_X32	cts_inv_589461445	
VDD	VDD	1.25e+08	2.00	0		4.774e-06	2.297e-07	9.552e-05	0.0001005	1.10	INV_X16	cts_inv_589161442	
VDD	VDD	1.25e+08	2.00	0		1.057e-05	4.593e-07	8.941e-05	0.0001004	1.10	INV_X32	cts_inv_530460055	
VDD	VDD	1.25e+08	2.00	0		3.734e-06	2.297e-07	9.647e-05	0.0001004	1.10	INV_X16	cts_inv_531960870	
VDD	VDD	1.25e+08	2.00	0		1.179e-05	4.593e-07	8.766e-05	9.99e-05	1.10	INV_X32	cts_inv_529460845	

This example shows a typical power summary. The power values are separately reported for each power domain, each frequency domain, groups that you define.

**** RedHawk-SC Power Summary Report ****													
created: Sun Sep 20 21:18:41 2020													
A total of 1185384 instances were summarized for this report while 0 were omitted due to missing power data (100.00% coverage).													
A total of 0 pins were omitted because they were not attached to a power domain.													

grouping	clock_pin_power(W)	internal_power(W)	leakage_power(W)	switching_power(W)	total_power(W)	percent_power(%)	pin_count	instance_count					
*** Power Domains:													
VDD	0.025298	0.045948	0.011256	0.071495	0.1287	60.50	912472	909926					
core3/VDD_INT	0.0078372	0.027775	0.0035269	0.052723	0.084025	39.50	275458	275458					
Total	0.033135	0.073723	0.014782	0.12422	0.21272	100.00	1187930	1185384					
*** Frequency Domains:													
1.25e+08	0.033135	0.073723	0.013739	0.12422	0.21168	99.51	377445	377445					
0	0	7.9383e-08	0.0010435	0	0.0010436	0.49	810485	807939					
Total	0.033135	0.073723	0.014782	0.12422	0.21272	100.00	1187930	1185384					
*** User Defined Groups:													
combinational logic	0	0.021952	0.0090181	0.11249	0.14346	67.44	310766	308220					
sequential logic	0.032636	0.051279	0.0057205	0.011719	0.068719	32.30	71800	71800					
memory	0.00049972	0.00049296	4.3939e-05	6.9543e-06	0.00054366	0.26	8	8					
decap/filler	0	0	0	0	0	0.00	805356	805356					
Total	0.033135	0.073723	0.014782	0.12422	0.21272	100.00	1187930	1185384					
**** End Report ****													

5.5. Scaling Power

The power from user-defined toggle rates in PowerView might not always match the design target power. The tool can scale the power from an input PowerView to the target power or the target toggle rate. You can generate multiple scaled power views from the same input power view. Leakage power can also be scaled.

To scale power, use the `create_scaled_power_view` command as shown in the following example. The example uses the `disable_scaling_sources` key to exclude macro power values from scaling.

```
object_settings = {'design_values':{
    'target_power':{Net('VDD'):0.15,
    Net('core3/VDD_INT'):0.0332},
    'disable_scaling_sources':['modecontrol']}}

scaled_power_settings= {'object_settings':object_settings}

pwr_mixed_scaled_args = dict(tag='pwr_mixed_scaled',
                             settings = scaled_power_settings,
                             options=options)

pwr_mixed_scaled = db.create_scaled_power_view
                    (power_view=pwr_mixed,**pwr_mixed_scaled_args)
```

The `object_settings` dict under the `settings` argument of the `create_scaled_power_view` command has the following scope-level keys:

- `design_values`
- `block_values`
- `cell_values`
- `leaf_instance_values`
- `leaf_instance_pin_values`

The following examples show how to use the different keys that are available under the scope-level keys.

Example 1: Specifying pin-based target_power:

```
object_settings
    = {'leaf_instance_pin_values': [ {'instances' : [Instance('U1/U2/RAM1')],
    'pins': [Pin('VDD')], 'target_power':1.2e-6 }, ]}
```

Example 2: All instantiations of CPU have same power target:

```
object_settings =
    {'cell_values' : [ {'patterns': 'CPU', 'target_power':{Net('VDD'):1.5e-3,
    Net('VDDB'):0.3e-4}, 'min_clock_scale_factor':{Net('VDD'):0.3}} ]}
```

Example 3: Using wildcard to set power target of multiple blocks:

```
object_settings =
    {'block_values' : [ {'patterns': 'CPU_*',
    'target_power':{Net('VDD'):1.5e-3,
    Net('VDDB'):0.3e-4}, 'min_clock_scale_factor':{Net('VDD'):0.3}} ]}
```

Example 4: Using toggle rate setting and leakage_scaling_factor:

```
object_settings =
    {'design_values': { 'toggle_rate':0.2, 'clock_pin_toggle_rate':1.2},
     'block_values': [ { 'patterns' : 'cpu', 'toggle_rate':0.1},
                      { 'patterns' : 'gpu', 'enable_scaling':False}],
     'cell_values' : [ { 'patterns' : 'AND2', 'leakage_scale_factor' :
    0.6} ] }
```

Example 5: Setting with leaf_instance_pin_values:

```
object_settings =
    {'design_values': { 'toggle_rate':0.2, 'clock_pin_toggle_rate':1.2},
     'leaf_instance_pin_values': [ { 'instances':[Instance('ram1'),
    Instance('ram2')], 'pins':[Pin('VDD')], 'leakage_scale_factor':1.2,
    'toggle_rate':0.8} ]}
```

Example 6: Setting target power for all pins of a leaf instance using leaf_instance_values:

```
object_settings =
    {'leaf_instance_values' : [ { 'instances':[Instance('ram1')],
    'target_power':{Pin('VDD'):1e-7, Pin('VDDB'):2e-8}} ]}
```

Note: `toggle_rate` or `clock_pin_toggle_rate` cannot be used with target power related keys, that is, `target_power`, `min_clock_scale_factor`, `min_data_power_fraction`, and `scale_leakage_power`.

Limiting Maximum Clock Toggle Rate

When you input a high target power to the `create_scaled_power_view` command, the tool might scale the clock toggle rate to an unrealistic value. To limit the maximum clock toggle rate of clock instances while meeting the target power, use the `max_clock_toggle_rate` key. For example,

```
scaled_power_settings = {
    'object_settings' = {
        'design_values': {
            'target_power': 5.0,
            'max_clock_toggle_rate': 2.0}
    'block_values'[] = {
        'patterns': 'U2/U3'
        'target_power'[] = {Net('VDD'): 2.0},
        'max_clock_toggle_rate': 1.6}}}

scaled_pwr = db.create_scaled_power_view(..., settings = scaled_power_settings)
```

Scaling Power Components For Different Cell Types

You can define specific scale factors for different power components, such as leakage power, switching power, and internal power for different `cell_types`. You can define these both at the design and the block level.

The tool supports the following cell types:

combinational	flip_flop_bank
combinational_clock	latch
sequential_clock	latch_bank
memory	decap
macro	power_gate
flip_flop	

The following examples show how to scale different power components in the `settings` dict of the `create_scaled_power_view` command.

- Cell-type based scaling for blocks

```
settings= {'object_settings': {'block_values': [
    {'patterns':'part0_i', 'scaling_factors': [
        {'leakage':0.3, 'internal':0.4, 'switching':0.5,
        'cell_types':['latch','flip_flop']}]}]}
```

- Latch instances have 0.9 scaling factor for power components (internal and switching) and 0.3 scaling factor for leakage power.

```
settings={'object_settings':{'design_values':
    {'scaling_factors':[
        {'switching':0.9,'internal':0.9,'cell_types':['combinational','latch']},
        {'leakage':0.3,'cell_types':['latch','flip_flop']}],}}}
```

- Under the same scope, if you define different scaling factors for the same `cell_types` and power component, the tool considers the first `cell_types` setting.

```
settings={'object_settings': { 'design_values' : { 'scaling_factors':[
    {'leakage':0.9, 'switching':0.9, 'cell_types' : ['combinational', 'latch']},
    {'leakage':0.3, 'cell_types': ['latch', 'flip_flop']]},}}}
```

- Scaling based on the power domain

```
settings={'object_settings':{'design_values': { 'scaling_factors':[
    {'leakage':{Net('VDD'):0.5}, 'switching':0.6, 'internal':{Net('VDD'):0.8},
    'cell_types': ['flip_flop','latch','power_gate']}],}}}
```

Specifying Target Power Without Voltage Domain

The `object_settings` dict under the `settings` argument of the `create_scaled_power_view` command supports domainless power scaling, that is, setting the target power without specifying the voltage domain.

When you specify the total target power of one scope without specifying voltage domains, the tool automatically distributes the total target power to each voltage domain before scaling. Dynamic Power gets distributed for the domains based on their original dynamic power ratio. Static power remains same.

You can set domainless power scaling for all scopes, that is, `design_values`, `cell_values`, `block_values`, and `leaf_instance_values`.

Example1 : Setting target power for the design.

```
object_settings={'design_values':{'target_power':0.3}}
```

Example2 : Setting target power for all blocks of the same cell.

```
object_settings={'cell_values':[{'patterns':'cpu','target_power':0.3}]}
```

Example3 : Setting target power for a block.

```
object_settings={'block_values':[{'patterns':'core3','target_power':0.3}]}
```

Example4 : Setting target power for a leaf instance.

```
object_settings:{'leaf_instance_values':
[{'instances':[Instance("top/block/inv1")],'target_power':0.0004,}]}]
```

Scaling Power Based on Logic Hierarchy

The tool supports logic hierarchy-based power assignment and scaling. You can scale the power based on toggle rate, target power, or scale factor settings specified with the `create_scaled_power_view` command.

To generate and store the logic hierarchy data, use the `create_logic_hierarchy_data` command. The `create_logic_hierarchy_data` command returns a UserView that can be used by the `create_scaled_power_view` command to perform logic hierarchy-based power assignment and scaling. For more details, use `help(SeaScapeDB.create_logic_hierarchy_data)` command at the tool command prompt.

To define a block as a logic block with the `create_scaled_power_view` command, set the `logic_block` key to `True` in the dict of `block_values`:

```
object_settings = {'block_values':[{'patterns':<block_name>,
                                    'logic_block':True}]}
```

The default is `False`. When `logic_block` is set to `True`, the `patterns` should not be a regular expression.

To specify the UserView that holds the logic hierarchy data with the `create_scaled_power_view` command, use the `logic_hierarchy_data` argument.

Note: You must specify the `logic_hierarchy_data` argument when `logic_block` is set to `True` under the `object_settings` dict.

The following example shows how to scale the power of logic blocks based on toggle rate, target power or scale factor settings:

```
# toggle rate assignment
settings={'object_settings' : {'block_values' : [
                                            {'patterns':'xfs_ep1_1/memdb_efta30/mem0',
                                             'toggle_rate':1.5,
                                             'logic_block':True}
                                         ],},}

# target power assignment
settings={'object_settings' : {'block_values' : [
                                            {'patterns' : 'xfs_ep1_1/memdb_efta30/mem0',
                                             'logic_block': True,
                                             'target_power' : {Net('VDD') :2},}
                                         ],},}

# scale factor assignment
settings= {'object_settings' : {'block_values' : [
                                            {'patterns' : 'u_a7ss_pwr_hm',
                                             'logic_block': True,
                                             'scaling_factors' : [
                                                 {'leakage' : 0.2 },
                                                 {'internal' : 0.3 },
                                                 {'switching' : 0.5 }
                                             ],
                                         ],},}

# storing logic hierarchy data
logic_hier_data = db.create_logic_hierarchy_data(dv, tag='logic_hier_data')

# scaling power
sp_block = db.create_scaled_power_view(power_view=pwr, settings=settings,
options=options, tag='sp_block', logic_hierarchy_data=logic_hier_data)
```

For more details, see `help(SeaScapeDB.create_scaled_power_view)`.

5.5.1. Reporting Scaled Power

To report the scaled power of each design instance and the power summary, use the `write_instance_power_report_and_summary` command as shown in the following example:

```
emir_reports.write_instance_power_report_and_summary(pwr_mixed_scaled,
    reports_dir+'/instance_power_mixed_scaled.rpt',
    reports_dir+'/power_summary_mixed_scaled.rpt')
```

The following report shows the power summary after scaling. The power values are separately reported for each power domain, each frequency domain, and groups that you define.

The VDD domain total power is now 0.15 and the core3/VDD_INT power is 0.0332 as set with `target_power` key of the scaled PowerView. The memory total power remains the same as their power scaling was disabled in `object_settings`.

```
**** RedHawk-SC Power Summary Report ****
Created: Sun Sep 20 21:19:37 2020

A total of 1185384 instances were summarized for this report while 0 were omitted due to missing power data (100.00% coverage).

A total of 0 pins were omitted because they were not attached to a power domain.

***  

grouping      clock_pin_power(W)  internal_power(W)  leakage_power(W)  switching_power(W)  total_power(W)  percent_power(%)  pin_count  instance_count  

*** Power Domains:  

VDD           0.029815          0.054228          0.011256          0.084516          0.15          81.88          912472          909926  

core3/VDD_INT 0.0028889         0.010238          0.0035269         0.019435          0.0332          18.12          275458          275458  

Total          0.032704          0.064466          0.014782          0.10395          0.1832          100.00          1187930          1185384  

*** Frequency Domains:  

1.25e+08     0.032704          0.064466          0.013739          0.10395          0.18216          99.43          377445          377445  

0             0                 2.9262e-08        0.0010435         0                 0.0010435          0.57          810485          807939  

Total          0.032704          0.064466          0.014782          0.10395          0.1832          100.00          1187930          1185384  

*** User Defined Groups:  

combinational logic   0           0.017603          0.0090181          0.095338          0.12196          66.57          310766          308220  

sequential logic     0.032204          0.04637           0.0057205          0.0086055          0.060696          33.13          71800           71800  

memory             0.00049972         0.00049296        4.3939e-05        6.9543e-06          0.00054366          0.30            8              8  

decap/filler       0           0                 0                 0                 0                 0.00          805356          805356  

Total              0.032704          0.064466          0.014782          0.10395          0.1832          100.00          1187930          1185384  

**** End Report ****
```

5.6. Performing Static Analysis

This section describes:

- [Generating DC Current per PG Pin from Instance Power](#) on page 125
- [Querying Current Per PG Pin](#) on page 126
- [Computing Static Voltage Drop for PG Pins](#) on page 126

Generating DC Current per PG Pin from Instance Power

The static ScenarioView converts the total power per instance to DC current per PG pin. To create the static ScenarioView, use the `create_scenario_view` command as shown in the following example:

```
scn_mixed_static = db.create_scenario_view
(power_view=pwr_mixed_scaled, **scn_mixed_static_args)
```

```
scn_mixed_static_args = dict(
    settings = {
        'pvt' : {'voltage_levels' : voltage_levels},
        'scenario_type' : 'Static'
```

```

        }
tag='scn_mixed_static',
options=options)

```

Querying Current Per PG Pin

The static ScenarioView reads in the total power per instance and the voltage levels of each domain from the PowerView to compute the DC current per PG pin. For the vdd pin in [Querying PowerView Attributes](#) on page 118, the total power of 0.0001734 W is divided by the voltage of 1.1 V to get the current of 1.5772×10^{-4} A in the static ScenarioView. To query the current values, use the `get_demand_current` command as shown in the following examples:

```

scn_mixed_static.get_demand_current(Instance("core3/program_memory"), Pin('vdd'))
Waveform([
(0.0, 0.000157722199219279),
])
scn_mixed_static.get_demand_current(Instance("core3/program_memory"), Pin('gnd'))
Waveform([
(0.0, -0.000157722199219279),
]

```

Computing Static Voltage Drop for PG Pins

The static AnalysisView takes the SimulationView and ScenarioView as inputs and stores the results of simulation. All voltage and current queries are available after simulation from the AnalysisView. You can query these by using the `emir_reports` commands listed in [Reporting Static Analysis Results](#) on page 126. To create the static AnalysisView, use the `create_analysis_view` command as shown in the following example:

```

av_mixed_static = db.create_analysis_view(simulation_view=sv,
scenario_views=scn_mixed_static, **av_mixed_static_args)

av_mixed_static_args = dict(
    tag='av_mixed_static',
    settings = {'keep_stats_level' : 'Full'},
    options=options)

```

5.7. Reporting Static Analysis Results

The following commands are available to report static IR drop results:

```

reports_dir = 'reports/static_analysis'
gp_util.makedirs(reports_dir)

emir_reports.write_all_instance_voltages(av_mixed_static,
reports_dir+'/inst_voltage.rpt')
emir_reports.write_bump_currents(av_mixed_static,
reports_dir+'/bump_current.rpt')
emir_reports.write_bump_voltages(av_mixed_static,
reports_dir+'/bump_voltage.rpt')
emir_reports.write_demand_currents(av_mixed_static,
reports_dir+'/demand_current.rpt')
emir_reports.write_layer_voltage_report(av_mixed_static,
reports_dir+'/layer_voltage.rpt')
emir_reports.write_node_voltage_report(av_mixed_static,

```

```

reports_dir+'/node_voltage.rpt')
emir_reports.write_supply_currents(av_mixed_static,
reports_dir+'/supply_currents.rpt')
emir_reports.write_switch_report(av_mixed_static,
reports_dir+'/switch_report.rpt')

```

The following are typical report examples.

Note: To perform static IR drop analysis, the tool decouples the power network and the ground network. Further, the tool uses multiple input data methods. It is possible for power and ground currents to be unbalanced and have slightly different values.

inst_voltage.rpt

The `write_all_instance_voltages` command outputs the instance voltage across each instance of the design.

#	loc_x	loc_y	min_instance_static_voltage	pg_arc	instance
	(u)	(u)	(v)	pwr/gnd	
1196.265	221.305		1.098	core3/VDD_INT/VSS	core3/openMSP430_inst1.clock_module_0.clock_gate_dbg_clk.CLKGATETST_X1_1
1214.505	221.305		1.098	core3/VDD_INT/VSS	core3/_14250
1210.515	221.305		1.098	core3/VDD_INT/VSS	core3/_24170
1206.715	221.305		1.098	core3/VDD_INT/VSS	core3/_24128
1206.525	221.305		1.098	core3/VDD_INT/VSS	core3/HFSBUF_352_3764
1201.775	221.305		1.098	core3/VDD_INT/VSS	core3/_24139
1196.265	221.305		1.098	core3/VDD_INT/VSS	core3/_23837
1194.555	221.305		1.098	core3/VDD_INT/VSS	core3/_23840
1194.555	221.305		1.098	core3/VDD_INT/VSS	core3/_23841
1193.035	221.305		1.098	core3/VDD_INT/VSS	core3/_23834
1192.845	221.305		1.098	core3/VDD_INT/VSS	core3/HFSBUF_5391_328

bump_current.rpt

The `write_bump_currents` command outputs bump current waveforms.

TitleText: Bump Currents		
#	Time (ps)	I (A)
"VSS_1	0.00	-0.0000010
"VSS_10	0.00	-0.0004140
"VSS_100	0.00	-0.0008939
"VSS_101	0.00	-0.0002740
"VSS_102	0.00	-0.0006438

bump_voltage.rpt

The `write_bump_voltages` command outputs bump voltage waveforms.

```
TitleText: Bump Voltages
# Time (ps) V (V)
"VSS_1
    0.00      0.0000000
"VSS_10
    0.00      0.0000000
"VSS_100
    0.00      0.0000001
"VSS_101
    0.00      0.0000000
"VSS_102
    0.00      0.0000001
```

demand_current.rpt

The `write_demand_currents` command outputs the per voltage domain demand current waveforms.

```
TitleText: Demand Currents
# Time (ps) I (A)

"core3/VDD_INT
    0.00      0.0247338

"VSS
    0.00     -0.1384887

"VDD
    0.00      0.1394365
```

layer_voltage.rpt

The `write_layer_voltage_report` command reports the minimum and maximum node voltage per layer for each net of the design.

#	min_x (u)	min_y (u)	min_voltage_drop (v)	max_x (u)	max_y (u)	max_voltage_drop (v)	layer_drop (v)	net	layer
1001.56	156.905	6.891e-05	1225.98	603.4	0.001351	0.001282	VSS	metall1	
960.885	189.165	0.0001897	1197.8675	222.645	0.0009766	0.000787	core3/VDD_INT	metall1	
1001.685	156.905	6.818e-05	1206.525	159.075	0.0006576	0.0005894	VSS	metall2	
1002.485	156.905	6.718e-05	1182.485	224.105	0.0003217	0.0002545	VSS	metall3	
999.285	156.8	5.313e-05	1180.885	221.8	0.000307	0.0002538	VSS	metall10	
1002.235	156.905	6.594e-05	1182.485	224.105	0.0003197	0.0002538	VSS	metall4	
1002.235	156.905	6.544e-05	1182.685	224.105	0.0003186	0.0002532	VSS	metall5	

node_voltage.rpt

The `write_node_voltage_report` command outputs the voltage at each node.

```
# contents are sorted in decending order by layer, net, node_voltage
# loc_x    loc_y      layer     net      node_voltage
# (u)      (u)          (v)
1088.4    64.8    metal12    VSS      5.391e-05
1083.6    64.8    metal12    VSS      5.327e-05
1088.4    63.2    metal12    VSS      5.193e-05
1086.8    64.8    metal12    VSS      5.169e-05
1085.2    64.8    metal12    VSS      5.145e-05
1083.6    63.2    metal12    VSS      5.128e-05
1088.4    66.4    metal12    VSS      4.98e-05
1083.6    66.4    metal12    VSS      4.977e-05
1086.8    66.4    metal12    VSS      4.971e-05
```

supply_currents.rpt

The `write_supply_currents` command outputs the per package domain supply current waveforms.

```
TitleText: Supply Currents
# Time (ps) I (A)

"VSS
 0.00  -0.1384887

"VDD
 0.00   0.1394365
```

For more information about the reporting commands, use the `help` command at the tool command prompt. For example,

```
help(emir_reports.write_all_instance_voltages)
```

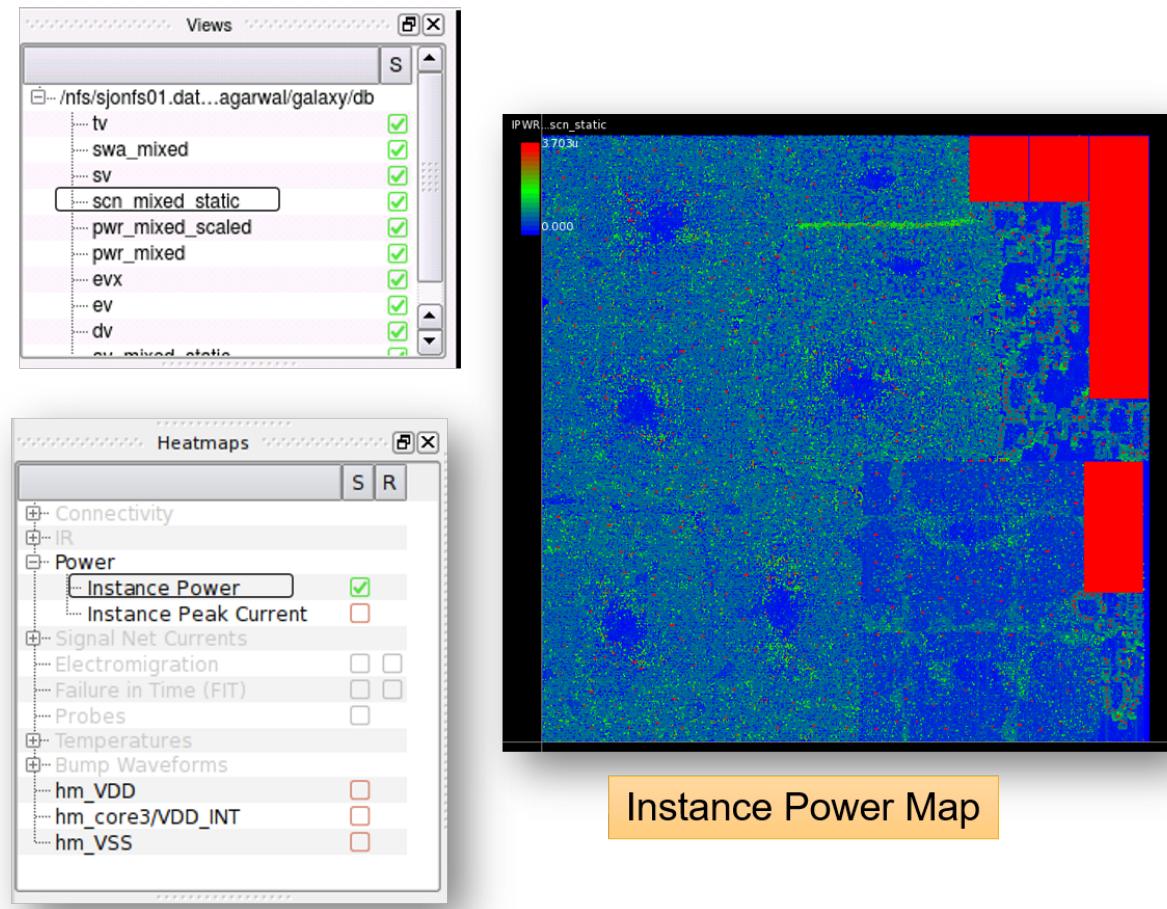
5.8. Viewing Static Analysis Results in GUI

Instance Power Map

When the ScenarioView is completed, you can view and analyze the heatmaps of instance power and the average instance currents.

To view the instance power map in GUI, follow these steps:

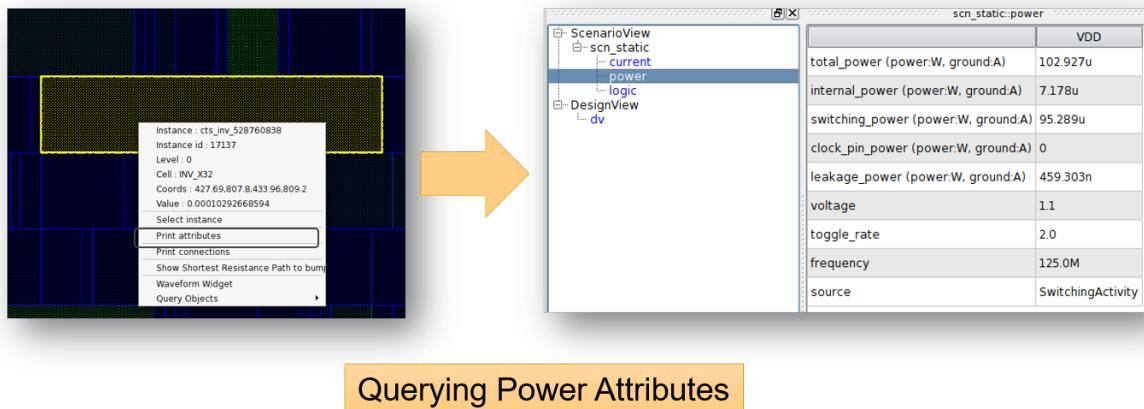
1. Select the ScenarioView database name under **Views**.
2. Select **Instance Power** under **Power** in **Heatmaps**.
3. To highlight the instances with high power dissipation, enable the Flame icon ().



To view the power attributes of any instance of the design:

1. Select the instance.
2. Right-click and select **Print attributes**.

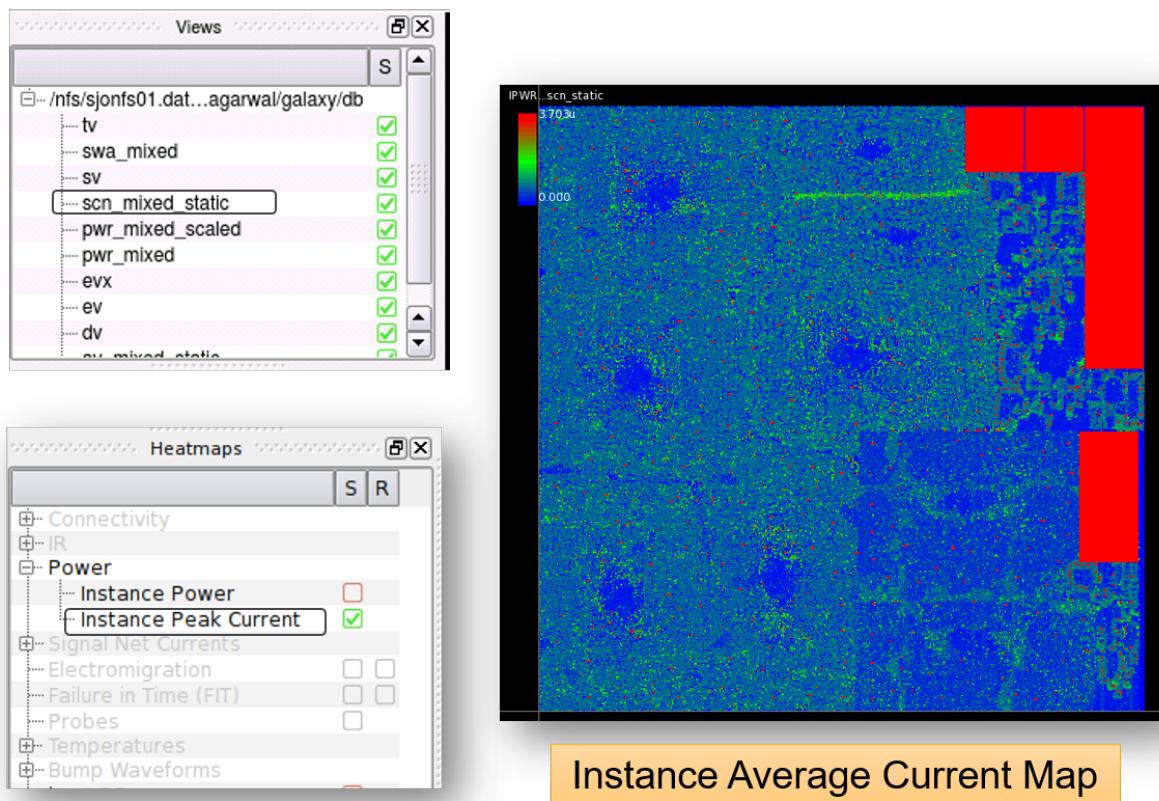
The GUI displays a table of the power attributes when you select **power** under **ScenarioView**.



Instance Current Map

To view the instance current map in GUI, follow these steps:

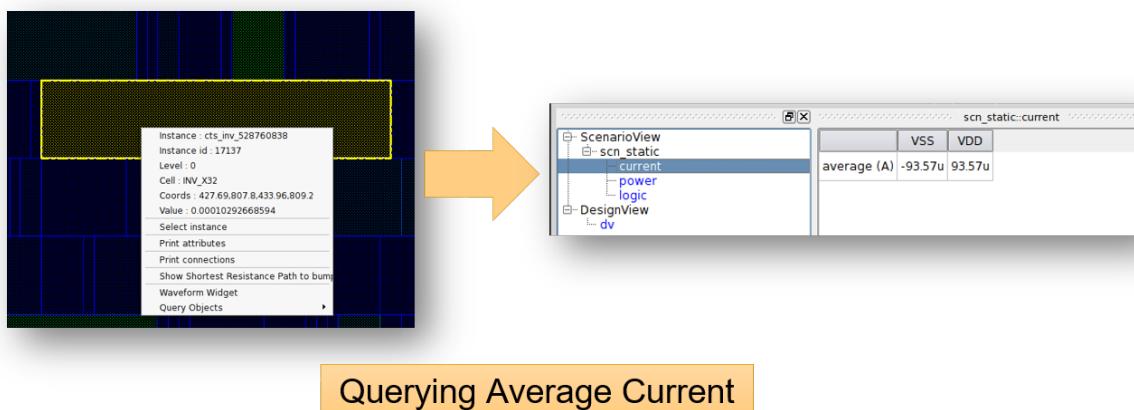
1. Select the ScenarioView database name under **Views**.
2. Select **Instance Peak Current** under **Power** in **Heatmaps**.
3. To highlight the instances with high average current, enable the Flame icon ().



To view the average current drawn by any instance of the design:

1. Select the instance.
2. Right-click and select **Print attributes**.

The GUI displays a table with the average current values for the VSS and VDD nets when you select **current** under **ScenarioView**.

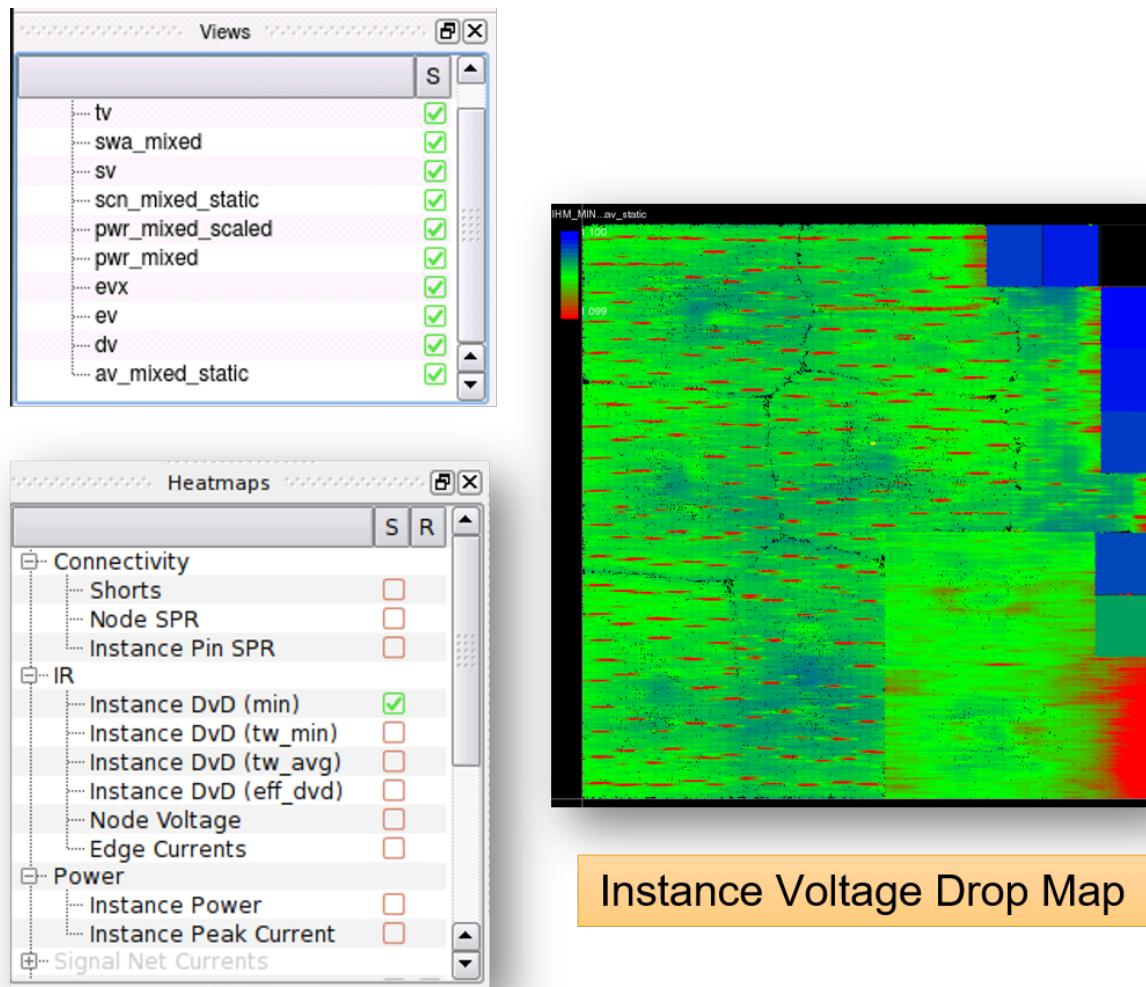


Instance Static Voltage Drop Map

When the AnalysisView is completed, you can view and analyze the instance voltage drop and node voltage drop heatmaps.

To view the instance voltage drop map in GUI, follow these steps:

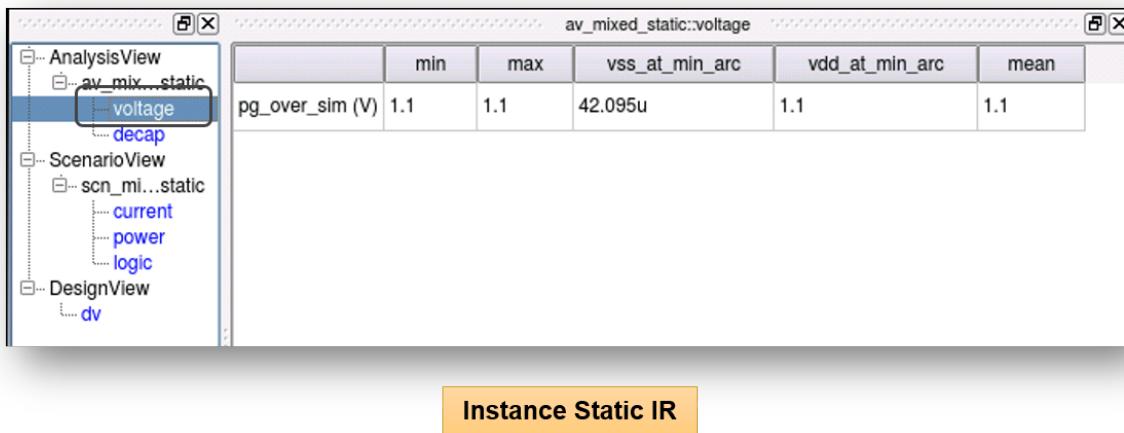
1. Select the AnalysisView database name under **Views**.
2. Select any one of **Instance DvD(min)**, **Instance DvD(tw_min)**, and **Instance DvD(tw_avg)** under **IR in Heatmaps**. For static analysis, each of these three parameters have the same results.



To query the static voltage drop values of an instance of the design:

1. Select the instance.
2. Right-click and select **Print attributes**.

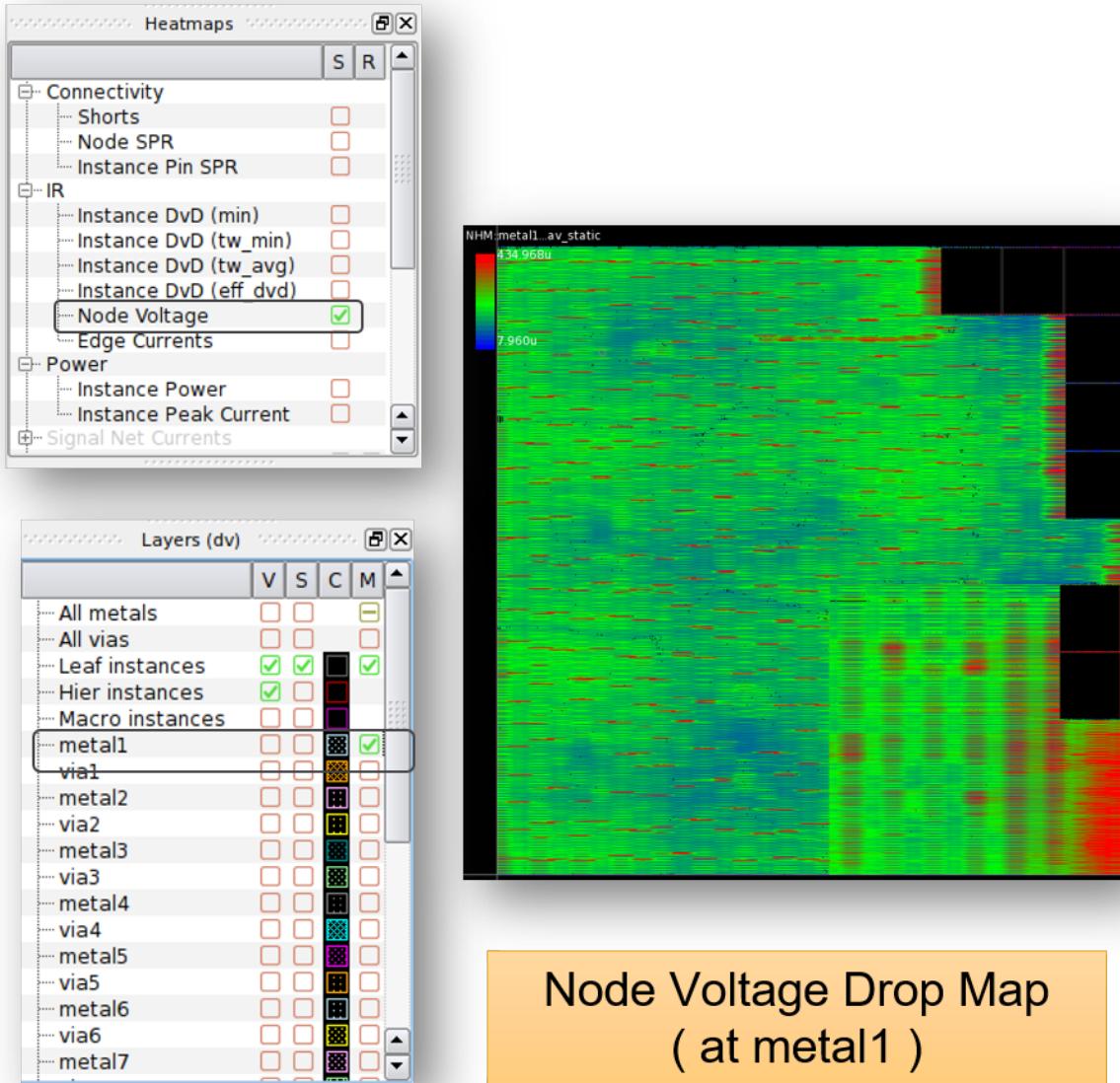
The GUI displays a table of static voltage values when you select **voltage** under **AnalysisView**.



Node Voltage Drop Map

To view the node voltage drop map in GUI, follow these steps:

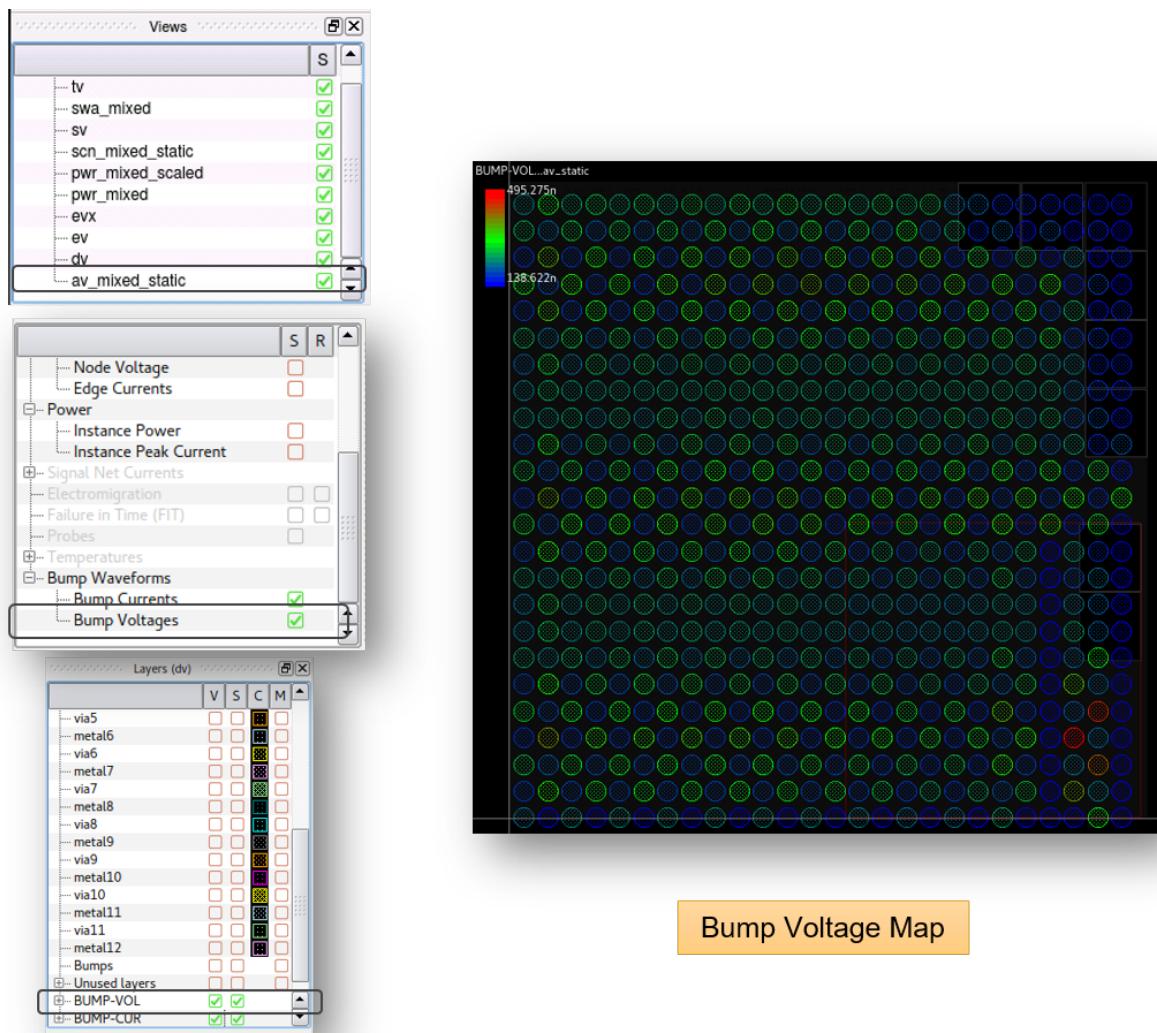
1. Select the AnalysisView database name under **Views**.
2. Select **Node Voltage** under **IR** in **Heatmaps**.
3. Select the required layer for node voltage drop under **Layers**.



Bump Voltage Map

To view the bump voltage map in GUI, follow these steps:

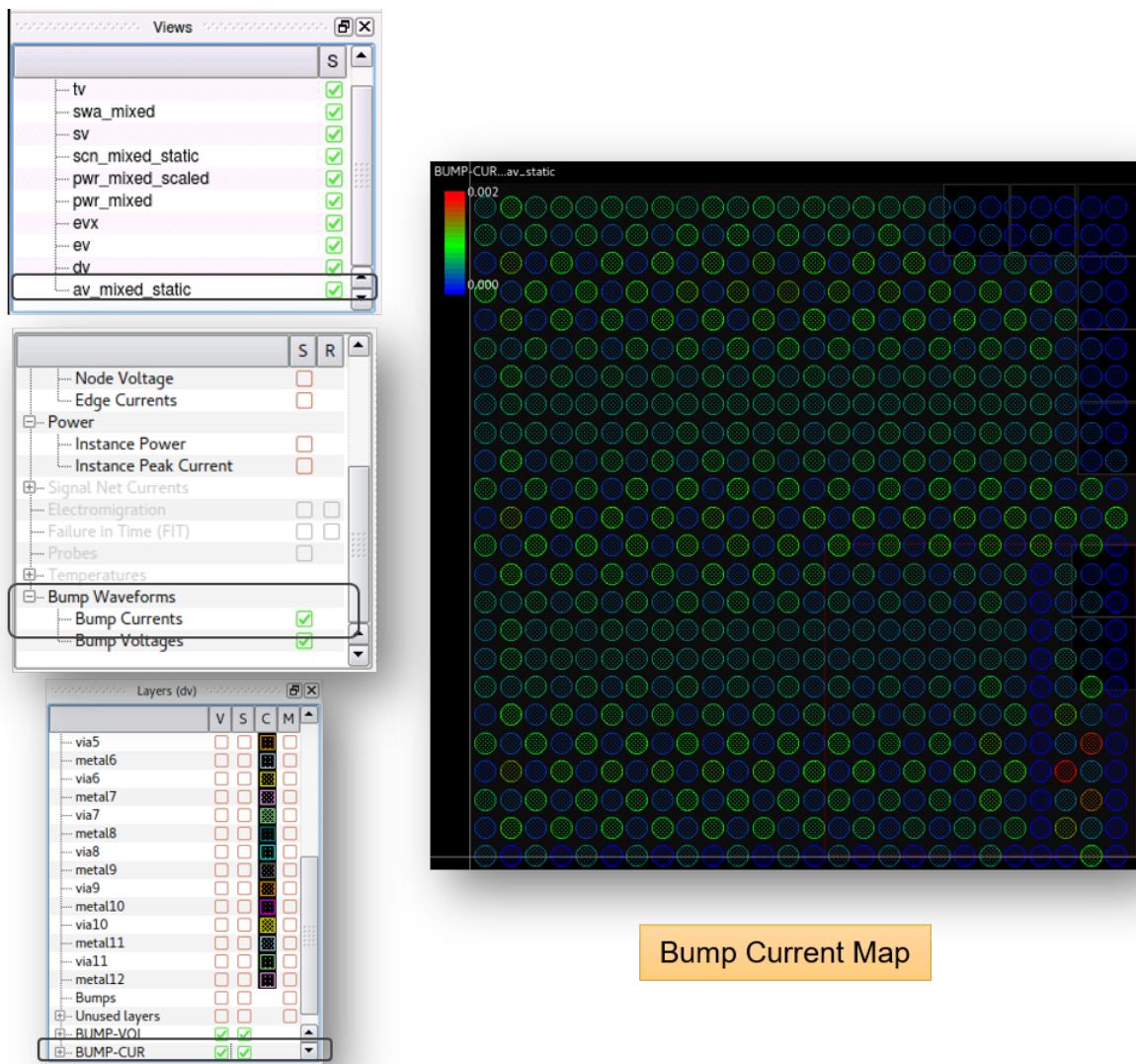
1. Select the AnalysisView name under **Views**.
2. Select **Bump Voltages** under **Bump Waveforms** in **Heatmaps**. This enables the **BUMP-VOL** layer.
3. Select both **BUMP-VOL** and **Bumps** under **Layers**.



Bump Current Map

To view the bump voltage map in GUI, follow these steps:

1. Select the AnalysisView name under **Views**.
2. Select **Bump Currents** under **Bump Waveforms** in **Heatmaps**. This enables the **BUMP-CUR** layer.
3. Select both **BUMP-CUR** and **Bumps** under **Layers**.



5.9. Generating Histograms

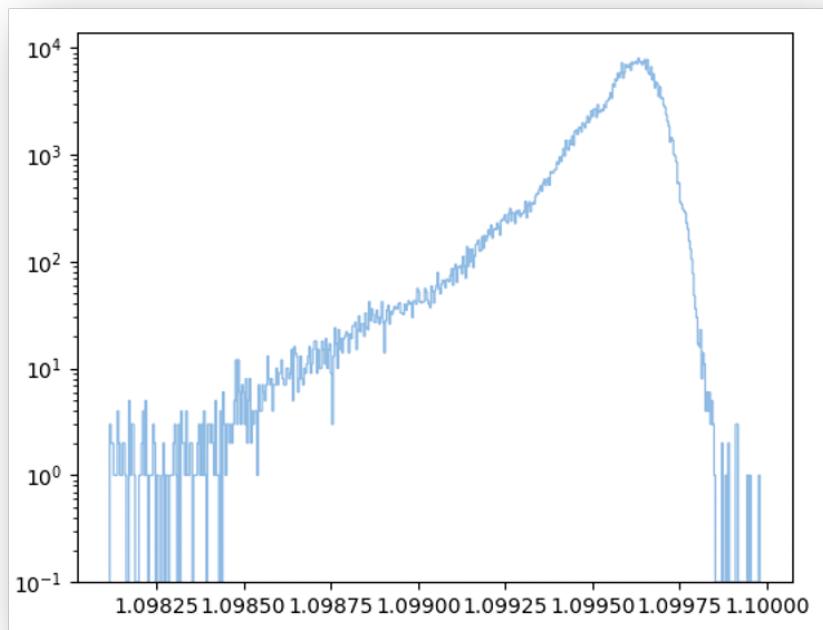
Post AnalysisView, you can create and plot histograms for static power analysis. The following sections describe how to generate the different types of histograms.

Instance Voltage Histogram

To generate the instance voltage histogram, use the command as shown in the following example:

```
plot(av_mixed_static.get_instance_voltage_histogram())
```

The following is a plot of the static voltage drop across an instance vs the number of instances and shows that how the instances are distributed about the voltage drop.



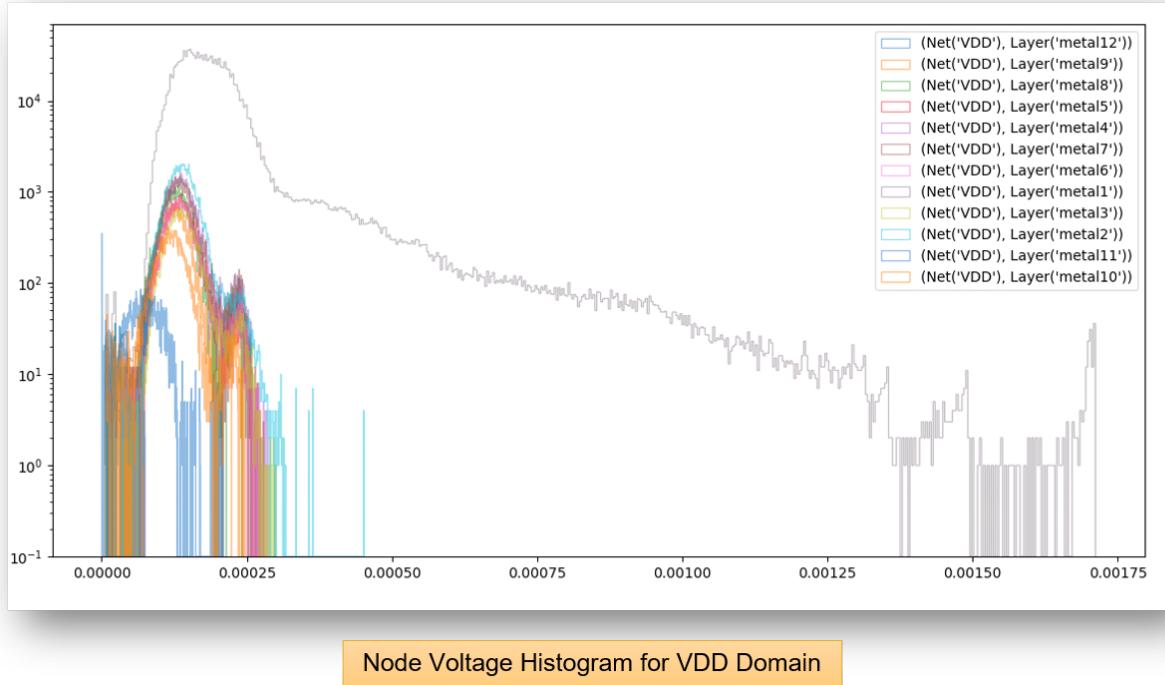
Instance Voltage Histogram

Node Voltage Histogram

To generate the node voltage histogram, use the commands as shown in the following example:

```
plot(av_mixed_static.get_node_voltage_histograms() [Net('VDD')])
```

The `get_node_voltage_histogram` command returns histograms for all the voltage domains in the design. The following is a plot of only the VDD node voltages versus the number of instances at all the layers.



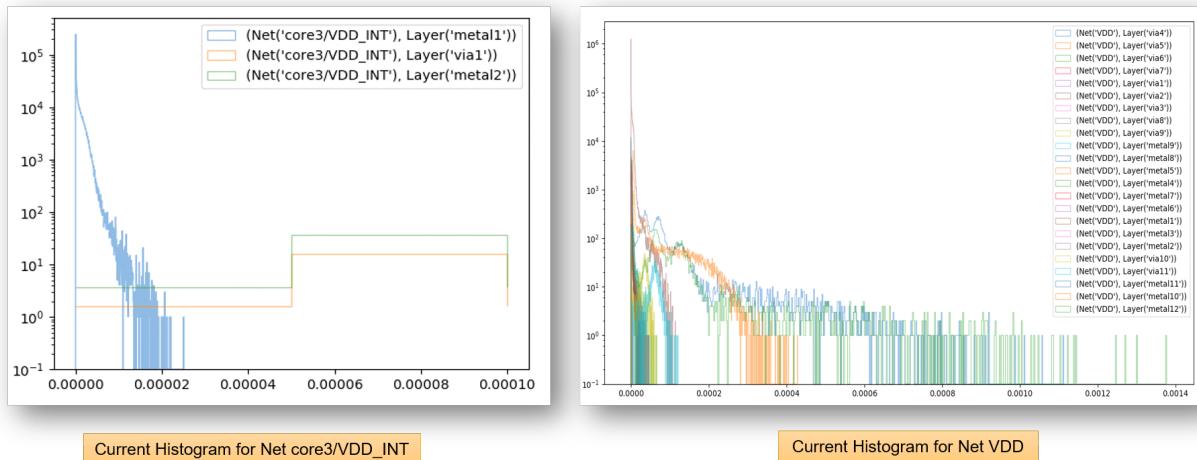
Node Voltage Histogram for VDD Domain

Current Histograms

To generate a current histogram, use the commands as shown in the following example:

```
plot(av_mixed_static.get_current_histograms() [Net('core3/VDD_INT')])
plot(av_mixed_static.get_current_histograms() [Net('VDD')])
```

The following plots show how the number of instances are distributed with respect to the net/node currents on different metal layers.



5.10. Static Electromigration Analysis

Electromigration (EM) is the movement of material that results from the transfer of momentum between electrons and metal atoms under an applied electric field. This momentum transfer causes the metal atoms

to be displaced from their original positions. This effect increases with increasing current density in a wire, and at higher temperatures the momentum transfer becomes more severe. Thus in advanced technology node designs, with higher device currents, narrower wires, and increasing on-die temperatures, the reliability of interconnects and their possible degradation from electromigration is a serious concern.

To check for electromigration, foundries specify the maximum current that can flow through a wire under varying conditions. These limits depend on several design parameters, such as wire topology, width, and length. Electromigration degradation and limits depend on the temperature at which interconnects operate, as well as on the material properties of the wires and vias, on the direction of current flow in the wire, and on the distance of the wire segment from the drivers.

The tool can perform electromigration checks for average, root mean square (RMS), and peak currents on both power/ground and signal nets of your design. This section describes the electromigration check to compare the average or DC current flowing through a wire or via against foundry-specified limits and report the violations as an example.

For power and ground net electromigration check, the current values are obtained from the AnalysisView. For signal net electromigration check, the current values are obtained from the SignalNetCurrentView. The electromigration limits or rules are input from the technology file information stored in the TechnologyView.

Electromigration analysis is described in the following topics:

- [Performing DC Electromigration Analysis](#) on page 139
- [Reporting Power Electromigration Results](#) on page 139
- [Viewing Electromigration Results in GUI](#) on page 140

5.10.1. Performing DC Electromigration Analysis

To perform power electromigration analysis, use the `create_electromigration_view` command as shown in the following example. This command checks for electromigration violations on all the metal layers and vias of the design.

```
pem_dc = db.create_electromigration_view(av_static, **pem_dc_args)
pem_dc_args = dict(
    tag='pem_dc',
    settings = {'temperature_em': 85.0, 'mode': 'dc'},
    options=options)
```

`tag` is the name of the ElectromigrationView database, `'mode': 'dc'` means that the electromigration check is for DC currents, and `temperature_em` specifies the temperature in Celsius.

For more details, see `help(SeaScapeDB.create_electromigration_view)`.

5.10.2. Reporting Power Electromigration Results

To report electromigration violations, use the `write_em_metal_report` and `write_em_via_report` commands after the ElectromigrationView (`pem_dc`) is created, as shown in the following example:

```
reports_dir = 'reports/powerem_analysis'
gp_util.makedirs(reports_dir)
emir_reports.write_em_metal_report(pem_dc,
reports_dir+'/DC_EM_metal_report.rpt')
emir_reports.write_em_via_report(pem_dc, reports_dir+'/DC_EM_via_report.rpt')
```

DC Electromigration Metal Report

The report header specifies the type of analysis, that is DC, and whether slivers were ignored. The columns report the wire co-ordinates, length, and width. The constraint column is the maximum permitted current, and the violation column shows the percentage of current with respect to the constraint. A violation of more than 100 % causes the metal electromigration check to FAIL.

# em_type = DC, ignore_sliver = True, 'length' column is blech length	# layer	from	to	length	width	current	constraint	violation	status	net
	#	(u)	(u)	(u)	(u)	(A)	(A)	(%)		
metall		(495.14,1136.8)	(495.23,1136.8)	393.6	0.17	0.0001134	0.0003156	35.94	PASS	VDD
metall		(495.23,1136.8)	(495.785,1136.8)	393.6	0.17	0.0001133	0.0003156	35.9	PASS	VDD
metall		(511.565,950.6)	(511.615,950.6)	393.6	0.17	0.0001111	0.0003156	35.2	PASS	VSS
metall		(271.32,88.2)	(271.615,88.2)	406.5	0.17	0.0001096	0.0003156	34.73	PASS	VSS
metall		(270.455,88.2)	(271.32,88.2)	406.5	0.17	0.0001096	0.0003156	34.73	PASS	VSS
metall		(256.5,238.0)	(256.775,238.0)	407.4	0.17	0.0001067	0.0003156	33.82	PASS	VDD
metall		(256.215,238.0)	(256.5,238.0)	407.4	0.17	0.0001067	0.0003156	33.82	PASS	VDD
metall		(256.025,238.0)	(256.215,238.0)	407.4	0.17	0.0001067	0.0003156	33.82	PASS	VDD
metall		(255.14,238.0)	(256.025,238.0)	407.4	0.17	0.0001067	0.0003156	33.82	PASS	VDD
metall		(256.775,238.0)	(257.54,238.0)	407.4	0.17	0.0001065	0.0003156	33.76	PASS	VDD
metall		(257.54,238.0)	(258.145,238.0)	407.4	0.17	0.0001064	0.0003156	33.71	PASS	VDD

DC Electromigration Via Report

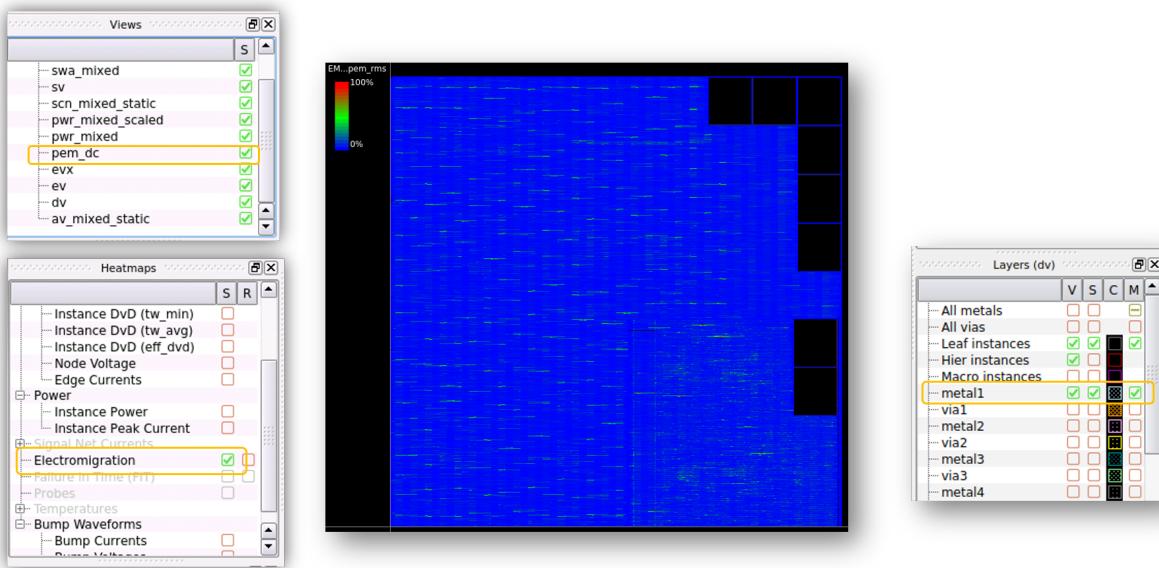
The columns report the via co-ordinates, length, and width. Similar to the metal report, the constraint column is the maximum permitted current, and the violation column shows the percentage of current with respect to the constraint. A violation of more than 100 % causes the via electromigration check to FAIL.

# em_type = DC, ignore_sliver = True	# layer	loc_x	loc_y	via_length	via_width	current	constraint	violation	status	net
	#	(u)	(u)	(u)	(u)	(A)	(u)	(%)		
viall		1088.4	61.6	0.8	0.8	0.0001266	0.0004978	25.43	PASS	VSS
viall		1088.4	153.6	0.8	0.8	0.0001237	0.0004978	24.86	PASS	VSS
viall		1129.6	204.4	0.8	0.8	0.0001212	0.0004978	24.34	PASS	VDD
viall		71.6	1078.4	0.8	0.8	0.0001189	0.0004978	23.88	PASS	VSS
viall		1129.6	107.6	0.8	0.8	0.0001183	0.0004978	23.77	PASS	VDD
viall		950.4	204.4	0.8	0.8	0.0001169	0.0004978	23.49	PASS	VDD
viall		71.6	61.6	0.8	0.8	0.0001145	0.0004978	23	PASS	VSS
via8		894.6	935.2	0.4	0.4	2.847e-05	0.0001244	22.88	PASS	VDD
via8		892.08	935.2	0.4	0.4	2.846e-05	0.0001244	22.87	PASS	VDD
viall		71.6	153.6	0.8	0.8	0.0001137	0.0004978	22.85	PASS	VSS

5.10.3. Viewing Electromigration Results in GUI

To view the power electromigration heatmap in GUI, follow these steps:

1. Select the ElectromigrationView name under **Views**. This step enables the electromigration heatmap.
2. Select **Electromigration** under **Heatmaps**.
3. Select the metal or the via layer under **Layers** in the multiselector (**M**) column.

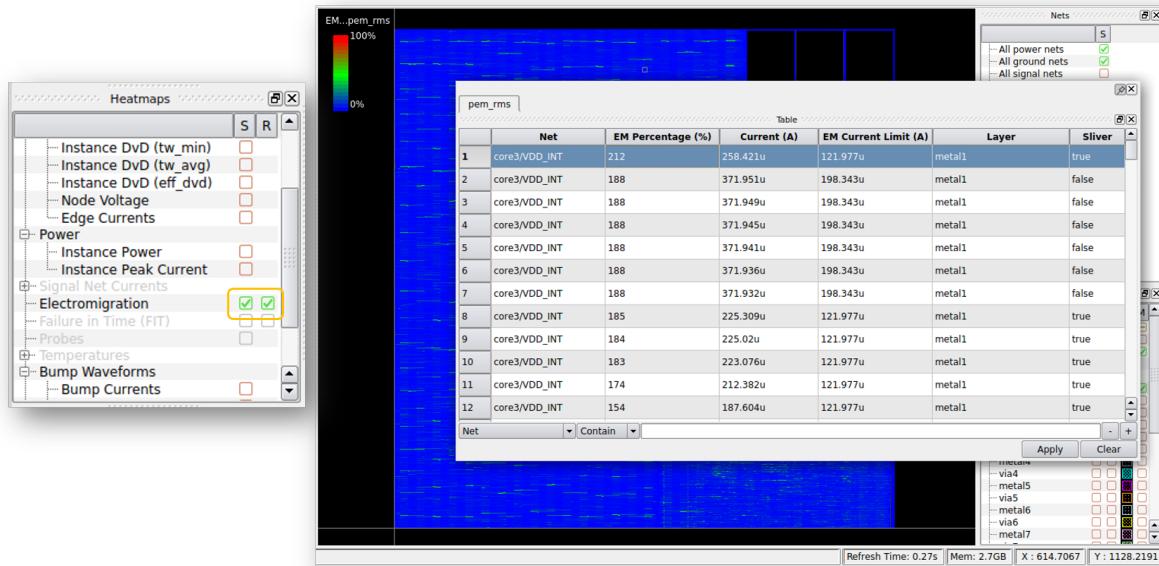


To view the electromigration violation browser from the heatmap, follow these steps:

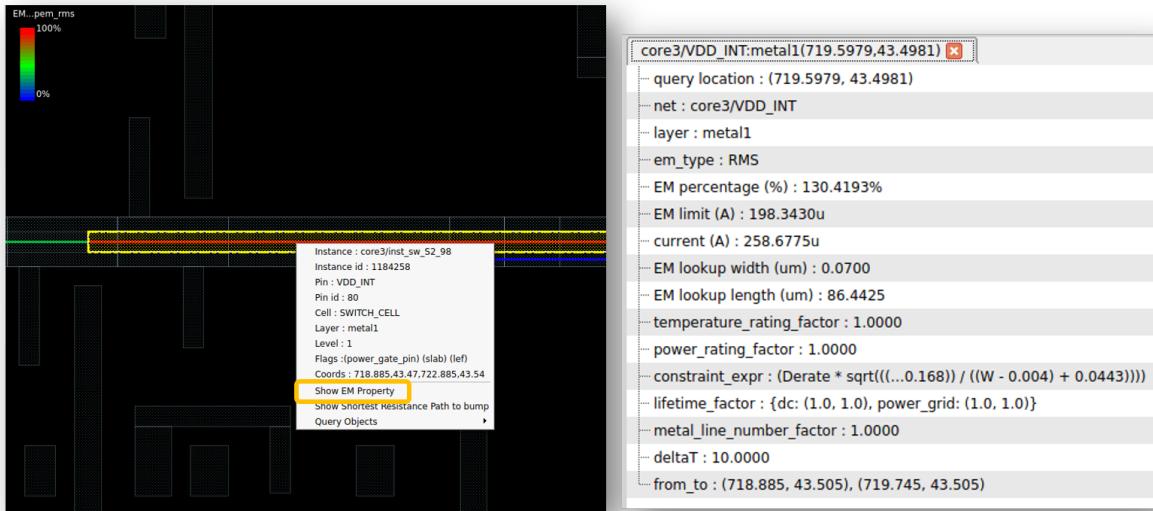
- Click the check box under the **R** column against **Electromigration** in **Heatmaps**.

The GUI opens the browser and reports the violations in the descending order of current.

- Double-click any row to view the corresponding wire segment.



- Right-click the wire segment and select **Show EM Property** to see the electromigration properties.



6: Vectorless Dynamic Analysis

Static IR drop accounts for the average current drawn from the power grid under average switching conditions. Dynamic IR or voltage drop is the instantaneous drop in rail voltages due to the instantaneous current drawn from the power grid because of one or more switching events. A high dynamic voltage drop (DVD) represents a weak power grid that is unable to meet peak current demand due to switching cells. DVD analysis also accounts for the high peak current demand of simultaneously switching design instances that create local hotspots. This current demand might be highly localized and last for a brief duration, such as, a single clock cycle, causing additional timing violations.

Because DVD depends on the switching activity of the logic, DVD analysis requires vectors. Vector data, such as VCD files are typically available late in the design cycle. Simulation vectors can be insufficient, do not necessarily identify all voltage drop problems in the design, and often need to be augmented for full design coverage.

The RedHawk-SC tool enables you to specify switching information and internally generates vectors based on these inputs to analyze DVD early in the design cycle. Because there are no input vectors, this is known as vectorless DVD analysis. In this case, the switching specification might not correspond to an actual event sequence.

In vectorless analysis, you can select not to propagate the logic state transitions or events through the circuit (`create_no_prop_scenario_view` command) or propagate the events (`create_scenario_view` command). You can further constrain a propagated scenario for power.

If vectors are available, you can use these to perform DVD analysis. In this case, an external switching specification with exact switching information is input to the tool. For details, see [Dynamic Analysis With Vectors](#) on page 237.

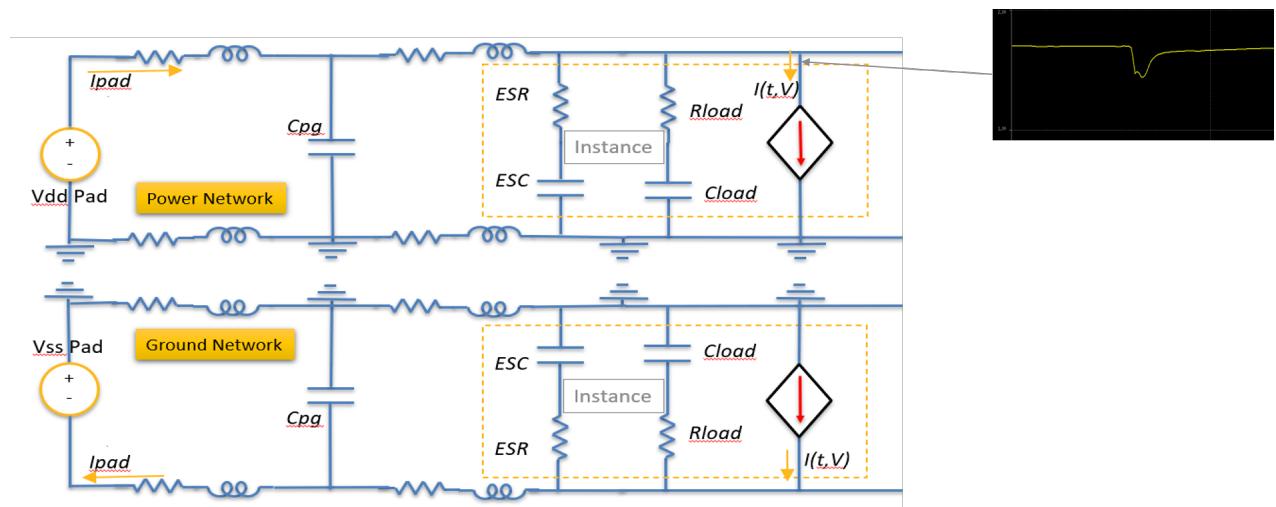
The following topics describe how to perform dynamic voltage drop and electromigration analysis by using the RedHawk-SC tool:

- [DVD Analysis Methodology](#) on page 143
- [DVD Analysis Flow](#) on page 144
- [Creating No-Propagation Vectorless \(NPV\) Scenarios](#) on page 145
- [Creating Logic Propagation Scenarios](#) on page 163
- [Power Constrained Vectorless Scenario](#) on page 174
- [Vectorless Scan Flow](#) on page 190
- [Controlling Macro Switching in RedHawk-SC](#) on page 196
- [Event Replay Flow](#) on page 207
- [Performing Dynamic Voltage Drop Analysis](#) on page 210
- [Reporting Dynamic Analysis Results](#) on page 215
- [Viewing Dynamic Analysis Results in GUI](#) on page 219
- [Querying Analysis Data](#) on page 223
- [Querying Instance Data](#) on page 224
- [Generating DVD Histograms](#) on page 228
- [Recommended Flow](#) on page 228
- [Dynamic PG Electromigration Analysis](#) on page 229

6.1. DVD Analysis Methodology

For dynamic voltage drop (DVD) analysis, the tool converts the on-chip power-ground network into a mesh of resistors, inductors, and capacitors during the ExtractView stage. The tool reads the effective series resistance (ESR) and effective series capacitance (ESC) values from APL or Liberty (intrinsic parasitic CCS power models) files.

The power network and the ground network are decoupled. The following circuits are simple examples of such meshes.

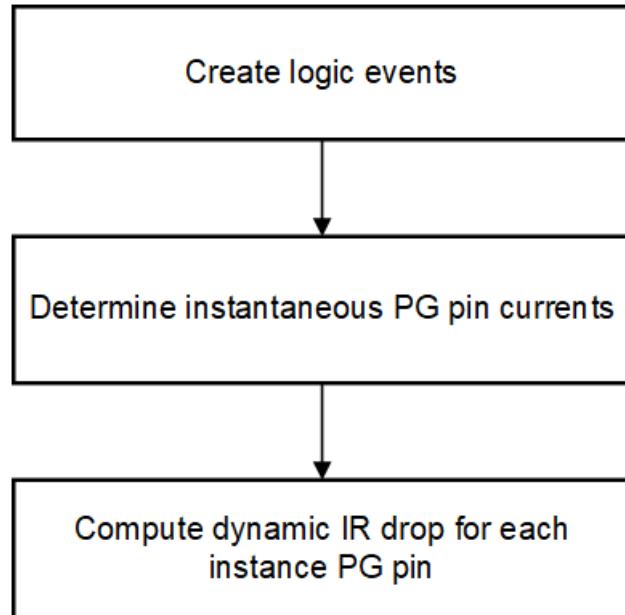


The tool replaces each design instance (current sink) with an equivalent dynamic current source. To do this, the tool first generates logic transitions called events at the input and output pins of an instance and then determines the PG current drawn by each instance from library data.

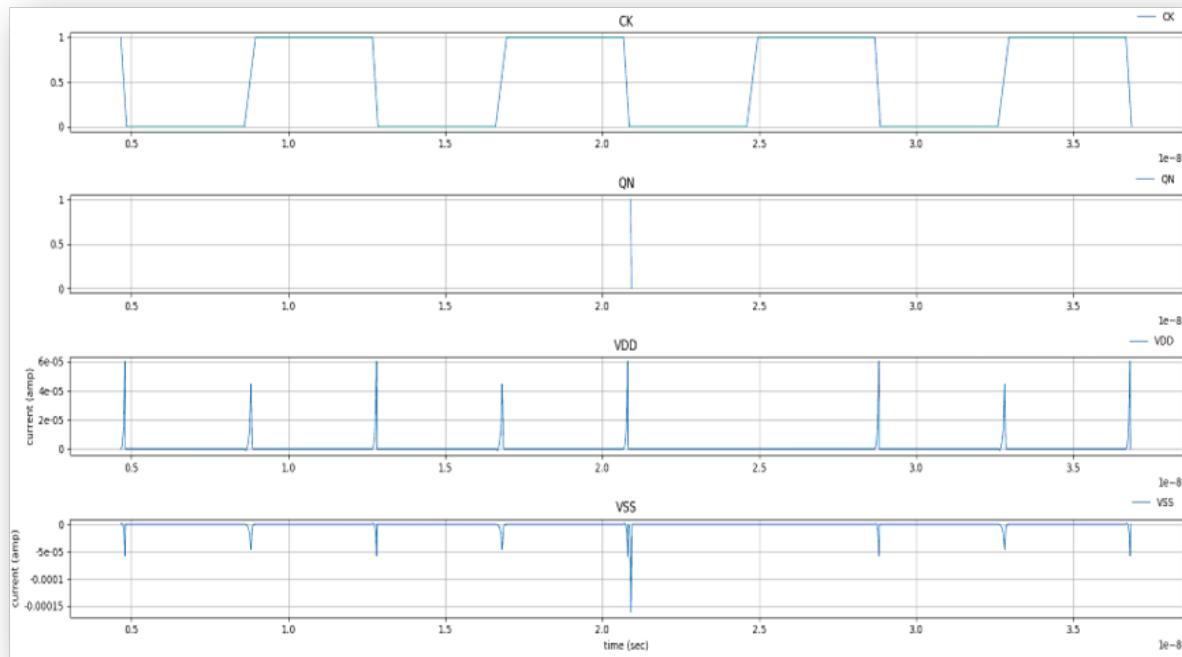
The instantaneous voltage at each PG node is determined by solving the representative circuits with the current sources and the impedance parameters. The total dynamic voltage drop for each instance is computed by adding the dynamic voltage drops at VDD and VSS.

6.2. DVD Analysis Flow

The following figure shows the high-level flow of RedHawk-SC dynamic voltage drop analysis. You must create the base views before performing these steps. See [Creating Base Views](#) on page 32.



In the first step, the tool creates the logic events based on your inputs as shown in the first two waveforms of the following figure. In the second step, the tool determines the instantaneous PG currents due to these events as shown in the last two waveforms.



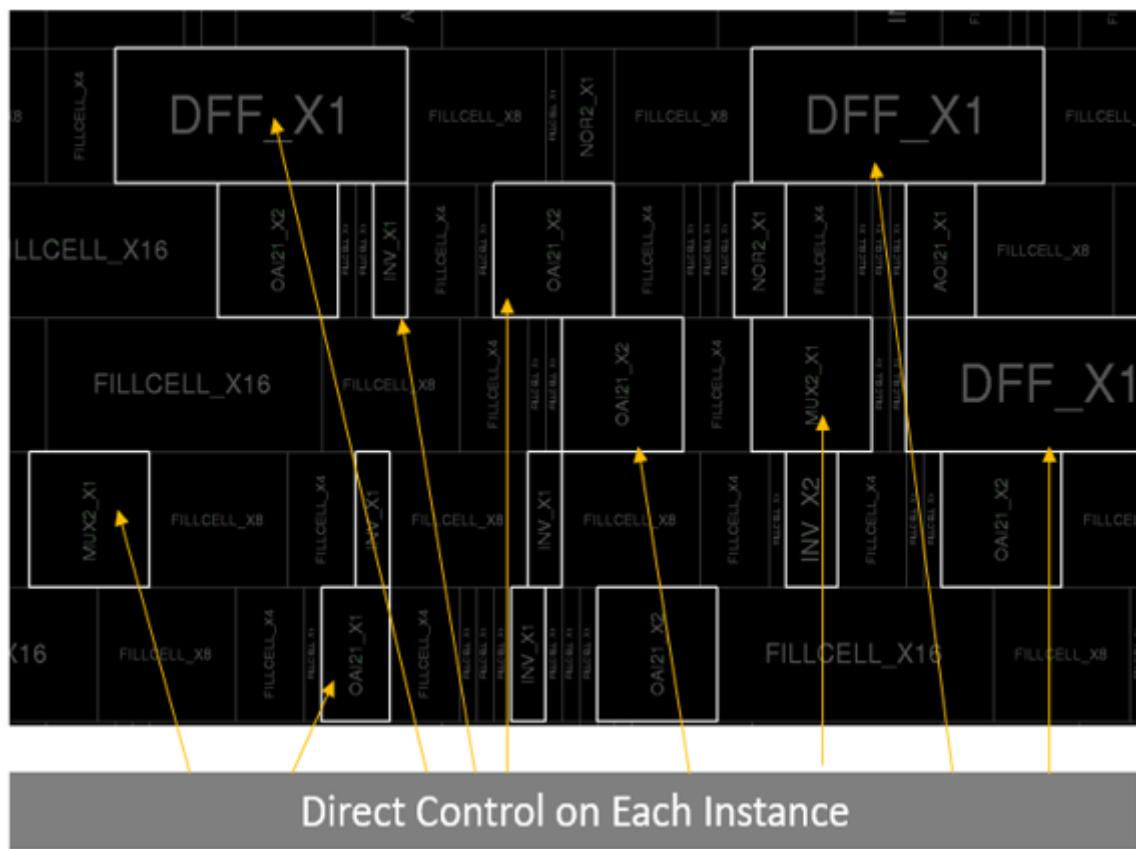
For the first two steps, use the `create_no_prop_scenario_view` or the `create_scenario_view` command depending on whether to propagate or not propagate the events through the design logic.

In the third step, the tool computes the dynamic voltage drop (DVD) by using the instantaneous current values.

For the third step, use the `create_analysis_view` command.

6.3. Creating No-Propagation Vectorless (NPV) Scenarios

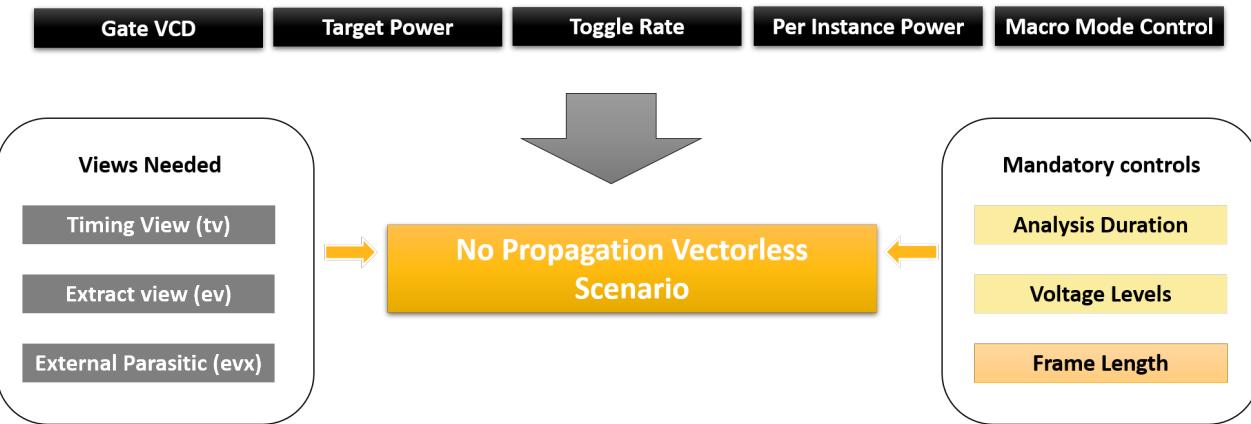
The no-propagation vectorless (NPV) scenario enables you to create events for each instance without propagating the events through the design logic.



Using a NPV scenario has the following benefits over a logic-propagation scenario:

- High design coverage in a short simulation
- Reduced memory and run time due to non-propagation of events
- Improved power matching with the user-specified target power

The following figure shows the various inputs to create a NPV scenario.



For the tool to create events, you can provide the input as gate VCD data, target power, toggle rates, or instance power files (IPFs). You can also use a combination of these inputs. If your design has macro instances, you must also input macro mode information.

For information about how to use different input combinations, see [Using Multiple Input Types Together](#) on page 161.

Only Gate VCD data are supported. RTL VCD data are not supported as they require event propagation. For information about how to input gate VCD to the NPV scenario, see [Dynamic Analysis With Vectors](#) on page 237.

To create a NPV scenario, use the `create_no_prop_scenario_view` command as shown in the following example:

```
scn_npv = db.create_no_prop_scenario_view
            (timing_view=tv, extract_view=ev, external_parasitics=evx,
             **npv_args)

npv_args = dict(
    scenario_duration=28e-09,
    settings=settings_npv,
    tag='npv',
    options=options)

settings_npv = {
    'pvt' : {'voltage_levels' : voltage_levels},
    # # specify default clock when STA annotation is missing
    'event' : {'default_clock' : {'policy':'custom', 'period':8e-9}},
    'frame_length' : 16e-09,
    'object_settings' : object_settings_npv
}
```

Use the `scenario_duration` argument to specify the total time for which the scenario is required. `scenario_duration` is a required argument. Use the `frame_length` key under the `settings` argument to specify the duration of a distinct switching pattern. See [Specifying Frame Length](#) on page 149.

Creating a no-propagation vectorless scenario includes the following sections:

- [Prerequisite Views](#) on page 147
- [Inputting object_settings Syntax as JSON Files](#) on page 148
- [Specifying Frame Length](#) on page 149
- [Specifying Target Power](#) on page 150
- [Specifying Toggle Rates](#) on page 152
- [Specifying Probability of Initial Events](#) on page 151
- [Loading PowerView](#) on page 155
- [Specifying Macro Modes](#) on page 158
- [Defining Default Clock Policy](#) on page 158
- [Defining Arrival Time Policy](#) on page 158
- [Specifying Switching Sequence for Standard Cell Instances](#) on page 159
- [Using Multiple Input Types Together](#) on page 161

Prerequisite Views

You must provide the following input views to the `create_no_prop_scenario_view` command:

- `TimingView` by using the `timing_view` argument.

To reduce run time, set `create_graph` to `False` while creating the `TimingView` provided the `TimingView` is input only where logic propagation is not required. If your entire flow does not contain logic propagation

`ScenarioView` (`create_scenario_view`) or logic propagation `SwitchingActivityView` (`create_switching_activity_view`), you should set `create_graph` to `False`.

```
tv_args = dict(
    timing_window_files=tw_files,
    settings = {'logic_graph': {'create_graph': False, ...}},
    tag="tv",
    options=options)
```

- External Parasitic View by using the `external_parasitics` argument.

The view stores input SPEF data.

- ExtractView by using the `extract_view` argument.

For nets that are not covered in SPEF files, the tool considers the internally calculated net capacitances from ExtractView. To enable extraction on both PG and signal nets, set the `detail_extraction_net_type` key to `all` under the `settings` dict of the `create_extract_view` command. By default, ExtractView does not calculate per segment signal parasitics that is only needed for signal electromigration analysis. However, ExtractView estimates the total capacitance for each net and those are used when SPEF data is not available.

6.3.1. Inputting object_settings Syntax as JSON Files

You can input the `object_settings` dict syntax as a JavaScript Object Notation (JSON) files. This includes but is not limited to `mode_control` for macro instances, `mode_control` for standard cells, toggle rate, current scale factor, and demand current restart specifications. This makes it easy to prepare the input for a large number of instances.

Note: JSON input files have valid JSON syntax.

The following example shows a typical JSON input file, **per_macro_input.json**, with macro mode control specifications:

```
{
  "file_version": "1.0",
  "content_type": "object_settings",
  "units": {
    "time": 1.0,
    "capacitance": 1.0
  },
  "object_settings": [
    {
      "leaf_instance_values": [
        {
          "instances": [ "macro_0" ],
          "mode_control": [
            {
              "mode_sequence": [
                "_median_energy_mode",
                "_leakage_only_mode"
              ]
            }
          ],
          "mode_sequence": [
            "_high_energy_mode",
            ...
          ]
        }
      ]
    }
  ]
}
```

```

        "_low_energy_mode_"
    ],
},
{
  "instances": [ "macro_2" ],
  "mode_control": [
    {
      "mode_sequence": [
        "_ACTIVE_write",
        "_leakage_only_mode_"
      ]
    }
  ]
}
}

```

The following example shows how to input the **per_macro_input.json** file to NPV ScenarioView:

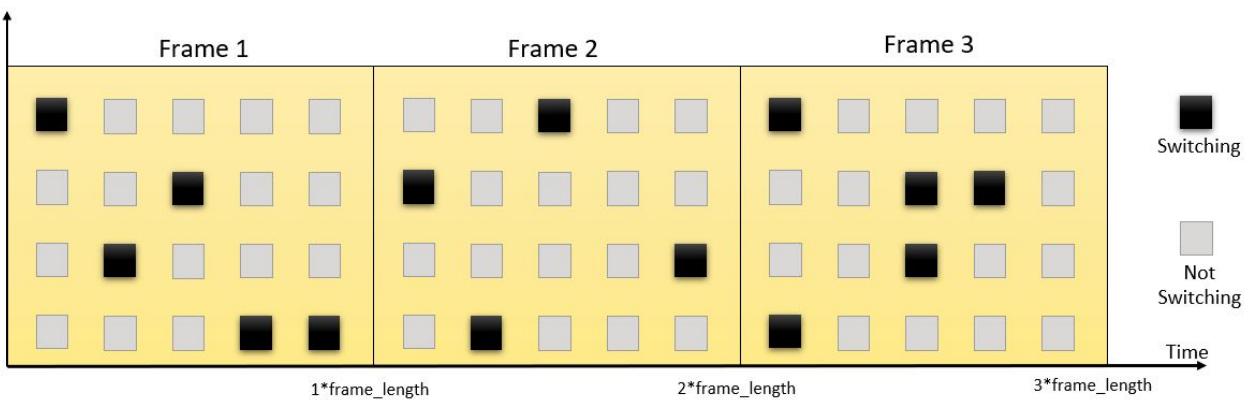
```

        npv_settings = {
            'object_settings' : {
                'input_files' : [ {'file_name' : 'per_macro_input.json'} ], {}}
        npv = db.create_no_prop_scenario_view(..., settings = npv_settings)

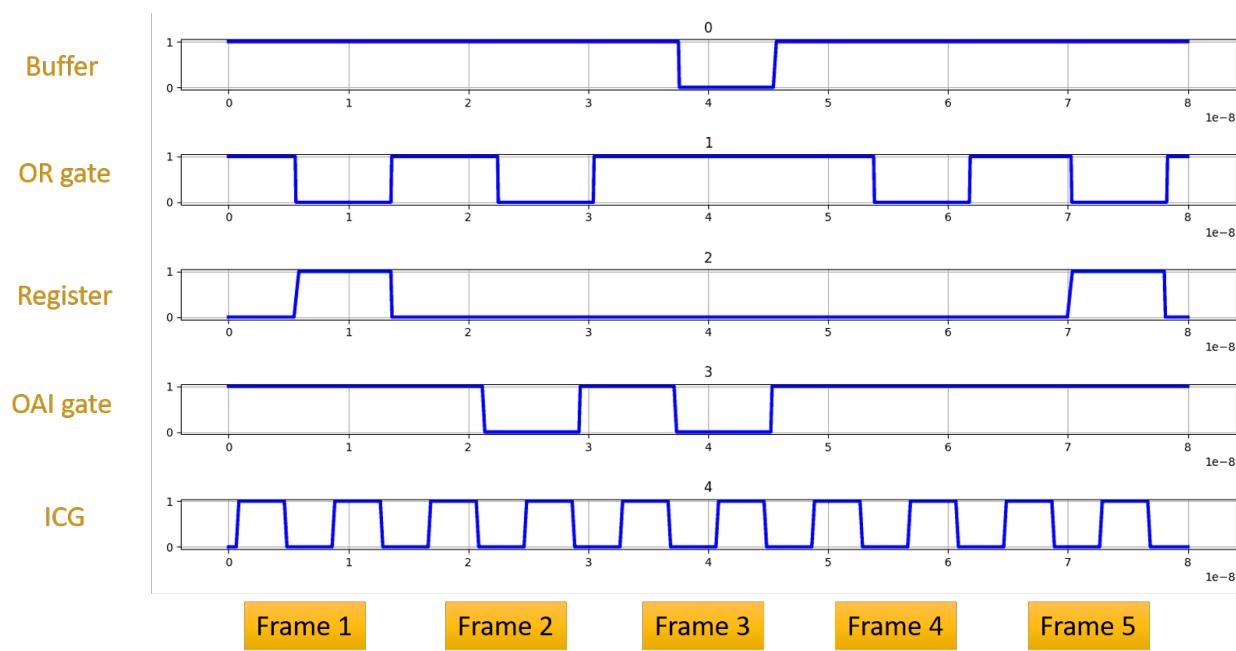
```

6.3.2. Specifying Frame Length

The tool creates a unique switching pattern for each frame. When you specify `frame_length`, the tool randomly selects some instances to switch in each frame duration. Because the tool uses random selection to switch instances, some switching instances in adjacent frames can be the same. The following figure shows an example where different instances switch in different frames. In this case, input switching is approximately 25%.



Because switching patterns change across frames, you should use multiple frames for better design coverage. However, increasing the number of frames also increases the run time. The following example shows the tool-induced switching behavior of different gates of a design when the frame length is twice the clock period. In two clock cycles, a switching instance can have one rise and one fall event.



For designs with multiple clocks, it is recommended to specify `frame_length` as an integral multiple of the dominant clock period (that drives most instances) of the design.

If you do not specify `frame_length`, the default `frame_length` equal to `scenario_duration` applies..

Note: You should explicitly specify `frame_length` because the default causes a single switching pattern (the same instances to switch) throughout the scenario duration.

The scenario duration need not be an integral multiple of the frame length. In such a case, the tool generates events for all frames and clips those events of the last frame that occur after scenario duration.

6.3.3. Specifying Target Power

For no-propagation vectorless dynamic analysis, you can specify the target power at the top level, blocks level, or for a voltage domain. The tool selects a set of instances to switch for each frame to meet the power targets.

To specify target power, use the `target_power` key in the `object_settings` dict under the `settings` argument of the `create_no_prop_scenario_view` command. You can specify `target_power` either at the top or the block level, that is under `design_values` or `block_values` key.

In `block_values`, you can specify `patterns` to apply the target power to matching block names. You can use wild cards. For example, '`patterns': 'block1*'` applies the specified target power to blocks with names, `block11`, `block12`, and so on. You can specify multiple patterns for blocks as `block_values` is a list of dicts `[{}, ..., {}]`.

You can specify the target power per voltage domain using `Net`, as shown in the following examples.

```
object_settings = {
    'design_values' : {
```

```

    'target_power':{
      Net('VDD') : 2.3} } }

object_settings = {
  'block_values': [
    {'patterns':'u_ABC/core1*',
     'target_power':{
       Net('VDD1'):0.16}}] }

```

You can also specify the same target power for all voltage domains using the wild card (*) syntax, as shown in the following examples.

```

object_settings = {
  'design_values' :{
    'target_power':{
      '*' : 3.3}} }

```

```

object_settings ={
  'block_values': [
    {'patterns': 'block1',
     'target_power':{
       '*':0.6}}] }

```

Note: The `target_power` syntax does not apply to macro cell instances.

Specifying Refinement Effort for Target Power Input Flow

To improve target power matching, use the `refinement_effort` key. When you specify this key, the tool checks the power achieved versus the specified target power, and changes the switching status of some instances when a better match is required. For example,

```

npv_settings = {'target_power' : {'refinement_effort' : 'low'}}
npv = db.create_no_prop_scenario_view(..., settings = npv_settings)

```

The default `refinement_effort` is high.

6.3.4. Specifying Probability of Initial Events

You can explicitly specify the probability of initial events by using the `first_event_rise_probability` key in the `event` dict under the `settings` argument of the `create_no_prop_scenario_view` command as shown:

```

npv_settings = {
  'event' : {'first_event_rise_probability' : 0.6}
}
scn_np = db.create_no_prop_scenario_view(..., settings = npv_settings, ... )

```

The default key value is 0.5, meaning equal probability of rise and fall events.

6.3.5. Specifying Toggle Rates

You can specify the toggle rate or activity factor. The tool selects a set of instances to switch based on the toggle rate. For example, a toggle rate of 0.3 causes about 30 percent of the instances to switch in each frame. Or, a toggle rate of 0.3 constraints the tool to switch an instance in 30 percent of the frames.

To specify the toggle rates, use the various available toggle rate keys in the `object_settings` dict under the `settings` argument of the `create_no_prop_scenario_view` command.

You can specify the toggle rates either at the top or the block level, that is, under `design_values` or `block_values` key. The block-level specification overrides the design-level toggle rates.

To specify different toggle rates for combinational and sequential circuits, use the `combinational_pin_toggle_rate` and `sequential_output_pin_toggle_rate` keys as shown in the following example.

```
object_settings = {
    'design_values': {
        'clock_pin_toggle_rate': 1.5,
        'combinational_pin_toggle_rate': 0.1,
        'sequential_output_pin_toggle_rate': 0.15},
    'block_values' : [
        {'patterns' : 'block3',
         'clock_pin_toggle_rate': 2.0,
         'combinational_pin_toggle_rate': 0.2,
         'sequential_output_pin_toggle_rate': 0.3}]}}
```

To specify the switching of a clock instance, use the `clock_pin_toggle_rate` key. The tool identifies an instance as a clock instance based on the inputs from the STA file. Instances where the output pin is marked as a clock in the STA file are considered clock instances.

To specify the toggle rate of an input clock pin to a sequential circuit, use the `sequential_clock_input_pin_toggle_rate` key.

The following is a complete example that shows how to specify toggle rates using the `create_no_prop_scenario_view` command. `mode_control` is used to control the switching of macro instances. For more information about controlling macro modes, see [Controlling Macro Switching in RedHawk-SC](#) on page 196.

```
# toggle rate specification
settings_npv={
    object_settings_npv = {
        'design_values' : {
            'clock_pin_toggle_rate': 1.6,
            'combinational_pin_toggle_rate': 0.15,
            'sequential_output_pin_toggle_rate': 0.2,
            'mode_control': {
                'mode_probabilities':{
                    '_high_energy_mode_': 0.6,
                    '_leakage_only_mode_': 0.4}}}},
    'block_values' : [
        {'patterns' : 'core3',
         'clock_pin_toggle_rate': 2.0,
         'combinational_pin_toggle_rate': 0.3,
         'sequential_output_pin_toggle_rate': 0.4},],},
    'pvt' : {'voltage_levels' : {'VDD':1.1, 'VSS':0.0}},
    'frame_length' : 8e-09,
    'event' : {'default_clock' : {'policy':'custom', 'period':8e-9}}
}
```

```

# command arguments specification
npv_args = dict(
    scenario_duration=40e-09,
    settings=settings_npv,
    tag='npv',
    options=options)

# top-level command inputs
npv = db.create_no_prop_scenario_view(
    timing_view=tv, external_parasitics=evx, extract_view = ev, **npv_args)

```

Specifying toggle rates includes the following sections:

- [Reading Instance Toggle Rate Files](#) on page 153
- [Controlling Clock Pins Switching](#) on page 154
- [Configuring MBFF Switching](#) on page 154
- [Maximizing Switching Coverage](#) on page 155

6.3.5.1. Reading Instance Toggle Rate Files

You can have the toggle rate of each design instance annotated in a file. The `create_no_prop_scenario_view` command cannot directly read this file and you must read in the file using PowerView. For the `create_power_view` command to read this file, you must first format this file as shown in the following example:

```

#inst_name - - #TR - - - - -
Inst1 - - 0.3 - - - - -
Inst2 - - 0.2 - - - - -

```

The file format is similar to a nine-column IPF file with only the first and the fourth columns populated. Specify the toggle rates in the fourth column. All the other seven columns must be empty and represented with a hyphen (-). Such a file is called an Instance Toggle Rate (ITR) file, and contains the toggle rates for each instance of the design. Rather than a single global toggle rate, using an ITR file enables you to specify toggle rates at a more granular level.

Use the `power_files` argument with the `create_power_view` command to read in this file and then input the PowerView to the `create_no_prop_scenario_view` command. The following example reads in the toggle rates from the `npv_TR_only` ITR file. The ITR file overrides design and block level toggle rate inputs.

```

power_files = ['npv_TR_only']
pwr_itr_args = dict(
    power_files = power_files,
    settings=settings_pwr,
    options = options,
    tag = pwr_itr')
npv_args = dict(
    scenario_duration=32e-09,
    settings=settings_npv,
    tag='npv',
    options=options)
pwr_itr = db.create_power_view(dv,**pwr_itr_args)
scn_npv = db.create_no_prop_scenario_view(
    timing_view=tv, external_parasitics=evx, extract_view = ev, power_view =
pwr_itr,**npv_args)

```

Note: In general, it is difficult to meet the toggle rate of each design instance unless the `create_no_prop_scenario_view` command runs for a long duration. Therefore, the tool uses the toggle rates for specific instances, such as, a set of fast switching instances. The tool might not meet the exact toggle rates from the file. The tool honors the toggle rate trends for a local area or region from the ITR file. For a region having instances with high toggle rates, the corresponding switching in NPV is also more.

6.3.5.2. Controlling Clock Pins Switching

To control the switching of clock instances, use the `always_active_clocks` key under the `settings` argument.

To specify that all the clock instances of a design switch in all frames irrespective of the `clock_pin_toggle_rate` value, set the `clock_instances` key to `True`.

To specify that clocks to all sequential instances of the design switch across all the frames, set the `sequential_instances` key to `True`. The default is `False` for both `clock_instances` and `sequential_instances`.

The following example shows how to use the `always_active_clocks` key.

```
npv_args = dict(...  
settings={  
    always_active_clocks={  
        'clock_instances' : True,  
        'sequential_instances' : True}  
}  
...)
```

To control clock switching at the leaf instance level, set the `always_active_clocks` key to `True`. The default is `False`. In the following example, `inst2` is a flip-flop where 60% of the clock cycles cause clock-to-Q switching in a frame while the remaining 40% have only clock switching. If you do not specify `always_active_clocks`, `inst2` remains in the leakage power state 40% of the frame duration. Use this feature when you require a pessimistic clock network scenario.

```
settings= {  
    object_settings = {  
        'leaf_instance_values': [  
            {'instances': Instance('inst2') ,  
             'toggle_rate' :0.6,  
             'always_active_clocks=True}] }  
...}
```

6.3.5.3. Configuring MBFF Switching

By default, output bits of a multibit flip-flop (MBFF) switch randomly based on the specified toggle rate. You can further control MBFF switching by using the `mbff_bit_switching_ratio` key (under `leaf_instance_values`) that defines the fraction of output bits to switch.

In the following example, `toggle_rate` and `mbff_bit_switching_ratio` controls are specified for individual instances. For instances of the `mbff1_list`, a toggle rate of 0.1 means that only 10% of instances

switch at a time and an `mbff_bits_switching_ratio` of 1 implies that all the output bits of these instances switch together.

For `mbff2` and `mbff3`, a toggle rate of 1 means these instances switch in all the frames and an `mbff_bits_switching_ratio` of 0.5 means that the selection of bits to switch is random.

For `mbff4`, the instance switches only in half the frames and only 75% of the output bits switch, that is six bits switch in an 8-bit MBFF. The `always_active_clocks` key set to `True` means that the clocks to this instance switches in all frames.

```
npv_settings = {
    'object_settings': [
        'leaf_instance_values': [
            {'instances': mbff1_list,
             'toggle_rate': 0.1,
             'mbff_bits_switching_ratio': 1}]}
    'leaf_instance_values': [
        {'instances': [Instance('mbff2'), Instance('mbff3')],
         'toggle_rate': 1,
         'mbff_bits_switching_ratio': 0.5}]}
    'leaf_instance_values': [
        {'instances': Instance('mbff4'),
         'toggle_rate': 0.5,
         'mbff_bits_switching_ratio': 0.75,
         'always_active_clocks': True}]}
...
}

npv_args = dict(..., settings = npv_settings)
```

6.3.5.4. Maximizing Switching Coverage

In no-propagation vectorless scenario, the tool generates switching patterns per frame. Increasing the number of frames improves switching coverage across the design. The `create_no_prop_scenario_view` command also enables you to meet maximum switching coverage in minimum number of frames by generating mutually-exclusive scenarios per frame. To do so, use the `ensure_coverage` argument with the command.

For example, if you specify a toggle rate of 0.25 and set `ensure_coverage` to `True`, about 25 % of the total instances switch in one frame. Because the instances switching in each frame are mutually exclusive, it might be possible to switch 100% instances in four frames provided there are no slow frequencies in the design outside the frame length.

Note: Selecting mutually-exclusive sets of instances to switch is an unlikely scenario and might create mismatched total demand currents across frames as well as localized switching effects. You should use this feature with caution and only in toggle-rate based flows.

6.3.6. Loading PowerView

The `create_no_prop_scenario_view` command can create a scenario that meets the power or toggle rate information contained in a PowerView. The command meets the approximate per region power from the input PowerView, that is, creates events per frame to match the power from PowerView. To read the power information, use the `create_power_view` command. See [Computing Power](#) on page 112 and [Scaling Power](#) on page 120.

One way to create the input PowerView is to use instance power files (IPFs). See [Determining Power Values from IPF](#) on page 114.

The tool supports both three-column and nine-column IPF formats. A three-column IPF is sufficient if you do not use the PowerView further in logic propagation scenarios.

To load the PowerView, use the `power_view` argument with the `create_no_prop_scenario_view` command as shown in the following example.

```
pwr_ipf_args = dict(
    power_files = ['high_power_mode.ipf'],
    tag='pwr_ipf',
    settings=settings,
    options=options)
pwr_ipf = db.create_power_view(design_view=dv, **pwr_ipf_args)
npv_args = dict(
    scenario_duration=40e-09,
    settings={
        'pvt' : {'voltage_levels' : voltage_levels},
        'frame_length' : 8e-09,
        'event' : {'default_clock' : {'policy':'custom', 'period':8e-9}},
        'macro' : {'ipf_target_power_pin_policy': 'all'}
    }
    tag='scn_npv',
    options=options)
scn_npv = db.create_no_prop_scenario_view(
    power_view=pwr_ipf, timing_view=tv, external_parasitics=evx, **npv_args)
```

For the tool to meet the toggle rate from PowerView rather than the target power, set the Boolean `prefer_power_view_toggle_rate_over_target_power` key to `True`. The default is `False`. This is useful when toggle rates are annotated from activity propagation in RedHawk-SC. For example,

```
npv_settings = {
    'event' : {'prefer_power_view_toggle_rate_over_target_power' : True}}
npv = db.create_no_prop_scenario_view(..., settings = npv_settings)
```

Inputting a PowerView includes the following sections:

- [Matching Power by Cell Type](#) on page 156
- [Matching Macro Power](#) on page 157

6.3.6.1. Matching Power by Cell Type

By default, NPV meets the approximate total power from IPF that is input to PowerView. When you set the `refinement_target` key to `Celltype` in the `target_power` dict, the command meets power values for each valid cell type. The default `refinement_target` is `Total` implying that the tool tries to meet the total input IPF power.

The command identifies the cell type from the library attributes of the master cell. Valid cell types are `clock`, `combinational`, `sequential`, `regbank` or `mbff`, and `macro`.

This increases both run time and memory consumption. In case the power for a particular cell type is not met, the tool honors the power constraints of other cell types and does not compromise their target power to match the total power. The following example shows how to use `refinement_target` with input IPF setting.

```
npv_args = dict(
    scenario_duration=40e-09,
```

```

settings={
    'pvt' : {'voltage_levels' : voltage_levels},
    'frame_length' : 8e-09,
    'event' : {'default_clock' : {'policy':'custom', 'period':8e-9}},
    'macro' : {'ipf_target_power_pin_policy': 'all'}
    'target_power' : {'refinement_target' : "Celltype" }
}
tag='scn_npv',
options=options)
scn_npv = db.create_no_prop_scenario_view(
power_view=pwr_ipf, timing_view=tv, external_parasitics=evx, **npv_args)

```

6.3.6.2. Matching Macro Power

By default, macros do not switch in NPV flows and remain in their leakage states. To enable the macro instances to switch automatically based on the power assigned to them in the input PowerView, use controls in the `macro` key under the `object_settings` dict. The `ipf_target_power_pin_policy` key specifies the power pin to be used for power matching.

This is useful in generating dynamic scenarios where the power distribution across instances is close to that in the IPF.

The `ipf_target_power_pin_policy` key can take one of the following values:

- all

The tool determines the modes to switch based on the power across all the power pins.

Note: For input PowerView based flows, using `all` is recommended.

- primary

The primary power pin defined in the Liberty file is the switching pin.

- dominant

The power pin with the highest power from IPF is the switching pin.

- user_defined

The switching power pin is input using the `macro_ipf_target_power_pin` key in the `object_settings` argument.

```

npv_args = dict(
    scenario_duration=40e-09,
    settings={
        'pvt' : {'voltage_levels' : voltage_levels},
        'frame_length' : 8e-09,
        'event' : {'default_clock' : {'policy':'custom', 'period':8e-9}},
        'macro' : {'ipf_target_power_pin_policy': 'all'}
    }
    tag='scn_npv',
    options=options)
scn_npv = db.create_no_prop_scenario_view(
power_view=pwr_ipf, timing_view=tv, external_parasitics=evx, **npv_args)

```

6.3.7. Specifying Macro Modes

The current consumed by a macro cell depends on its mode of operation. To annotate the macro modes, use the `mode_control` key in the `object_settings` dict under the `settings` argument of the `create_no_prop_scenario_view` command. The tool considers all the output pins of a macro instance when applying modes for current waveform creation.

`mode_control` syntax is also available with the `create_power_view` and `create_scenario_view` commands and the syntax usage is same in all these views. See [Controlling Macro Switching in RedHawk-SC](#) on page 196.

6.3.8. Defining Default Clock Policy

By default, instances with missing clock information in the STA file do not switch. To specify the default clock to switch such instances, use the `default_clock` argument. `default_clock` is a dict with `keys`, `policy` and `period`. The `policy` key can have one of the following values:

- `fastest`
Fastest among STA file frequencies
- `slowest`
Slowest among STA file frequencies
- `dominant`
Most used frequency from STA file
- `custom`
User-input value

If you specify the default clock policy as `custom`, you should also specify the `period` key in the `default_clock` argument. Otherwise, tool uses the default `period` of `0.1e-09` seconds.

The following examples show how to define the default clock policy.

```
default_clock = {'policy' : 'fastest'}
```



```
default_clock = {
    'policy' : 'custom'
    'period' : 1e-09}
```

6.3.9. Defining Arrival Time Policy

A STA file includes the minimum and the maximum arrival time of a switching event. To specify where to place the switching event between these two values, use the `arrival_time_policy` key in the `sta` dict under the `settings` argument.

You specify the `arrival_time_policy` key as a dict. `arrival_time_policy` includes the following two keys. Both the keys can have a value of `min`, `avg`, `max`, or `random`.

- `clock_path_pins`

Controls arrival time policy of pins in the clock path, clock pins of sequential cells, and clock pins of macro cells. Default is `avg`, that is, the tool switches the clock pins at the average of the minimum and the maximum arrival times.

- `data_path_pins`

Controls arrival time policy of non-clock path pins. Default is `random`, that is, the tool switches the data path pins at any random point between the minimum and the maximum arrival times.

The following example shows `arrival_time_policy` usage.

```
'sta' : {
    'arrival_time_policy' : {
        'clock_path_pins' : 'min',
        'data_path_pins' : 'max'
    }
}
```

6.3.10. Specifying Switching Sequence for Standard Cell Instances

You can control the switching sequence of a master cell or a leaf instance as described in the following topics:

- [Specifying Switching Sequence per Instance](#) on page 159
- [Specifying Modes for Standard Cell Instances](#) on page 160
- [Combining Switching Control and Mode Control](#) on page 161

Specifying Switching Sequence per Instance

To define the switching sequence of each master cell or leaf instance for a given clock cycle, use the `switching_control` key under `cell_values` or `leaf_instance_values` in `object_settings`. The `switching_control` key is useful to switch a particular instance or multiple instances of the same cell in a predetermined sequence, and is only available for use with the `create_no_prop_scenario_view` command.

In the following example, the `switching_sequence` of `['high', 'fall', 'low', 'rise']` means that both `buf2` and `buf3` outputs do not switch in the first and third clock cycles, fall in the second cycle, and rise in the fourth cycle. This sequence is repeated in the entire scenario duration.

```
npv_settings = {
    object_settings = {
        'leaf_instance_values' : [
            {'instances' : [Instance('buf2'), Instance('buf3')],,
             'switching_control' : {
                 'switching_sequence' : ['high', 'fall', 'low', 'rise'],},},]
    }
}

npv_args = dict(..., settings = npv_settings)
```

For a sequential instance with multiple output pins, you can define the switching sequence of each output pin. In the following example, `switching_control` includes a list of switching sequences, one defined for each output pin of the sequential `mbff1` instance.

When the Boolean `switching_sequence_repeats` key is set to `False`, the last value in `switching_sequence` repeats until the scenario duration is complete. For pin Q1, the switching sequence is (fall, rise, rise, rise, ...).

```
object_settings = {
    'leaf_instance_values' : [
        {'instances' : [Instance('mbff1')],
         'switching_control' : [
             {'pin' : Pin('Q0'),
              'switching_sequence' : ['rise', 'high', 'fall', 'fall'], },
             {'pin' : Pin('Q1'),
              'switching_sequence' : ['fall', 'rise'],
              'switching_sequence_repeats' : False, }]}]}
```

The NPV scenario allows you to specify logically incoherent sequences, such as `['rise', 'high', 'fall', 'fall']`.

For a combinational instance with multiple output pins, the `switching_control` syntax controls only the last pin that you specify. The other output pins switch according to the logic function. In the following example, the last entry, that is, the CO pin is considered for switching as per the `switching_control` specification. If the S pin has the same related input pin as CO, then it switches as per logic events created on the input pins to drive the logic at CO pin.

```
object_settings = {
    'leaf_instance_values' : [
        {'instances' : [Instance('adder_full1')],
         'switching_control' : [
             {'pin' : Pin('S'),
              'switching_sequence' : ['rise', 'high', 'fall', 'fall'], },
             {'pin' : Pin('CO'),
              'switching_sequence' : ['fall', 'rise'], }]}]}
```

Specifying Modes for Standard Cell Instances

You can also use the `mode_control` key to define the switching sequence of a standard cell instance. This is useful to switch a particular instance or multiple instances of the same cell in their predetermined modes of operation. `mode_control` is a dict or a list of dicts and includes the following keys: `mode_sequence` and `mode_probabilities`.

The following predefined modes are available to specify `mode_sequence` for each standard cell or instance based on the energy consumption from Liberty data:

- `_active_mode_`: When the instance output pin switches
- `_inactive_mode_`: When only the clock pin switches
- `_leakage_only_mode_`: When no switching occurs and the instance draws only leakage current

In addition to `_active_mode_`, `_inactive_mode_`, and `_leakage_only_mode_`, the following predefined modes are available to specify `mode_probabilities` for each standard cell or instance:

- `_high_energy_mode_`: Highest energy mode
- `_low_energy_mode_`: Lowest energy mode

In the following example, the `mode_sequence` of `['_active_mode_', '_leakage_only_mode_','_active_mode_']` means that both `inst1` and `inst2` outputs switch in the first and third clock cycles and do not switch (leakage) in the second cycle.

`mode_probabilities` of `_high_energy_mode_` means a high energy mode in 75% of clock cycles and `_low_energy_mode_` means a low energy mode in 25% of cycles.

```

npv_settings = {
    object_settings = {
        'leaf_instance_values' : [
            {'instances' : [Instance('inst1'), Instance('inst2')],
             'mode_control' : {
                 'mode_sequence' : [
                     '_active_mode_',
                     '_leakage_only_mode_',
                     '_active_mode_'],
                 }
            },
            {'instances' : [Instance('inst1'), Instance('inst2')],
             'mode_control' : {
                 'mode_probabilities' : {
                     '_high_energy_mode_' : 0.75,
                     '_low_energy_mode_' : 0.25},
                 }
            }]}
}
npv_args = dict(..., settings = npv_settings)

```

Combining Switching Control and Mode Control

You can also use `switching_control` and `mode_control` together. In the following example, for `inst1`, a low value of `output_pin_initial_state` means that the instance starts switching from a LOW and the `_active_mode_value` of `mode_sequence` means that the switching sequence is (rise, fall, rise, fall, ...). For `inst2`, a high value of `output_pin_initial_state` means that the instance starts switching from a HIGH and the `mode_sequence` values mean that the switching sequence is (fall, low, rise, high, fall, low, rise, high, ...).

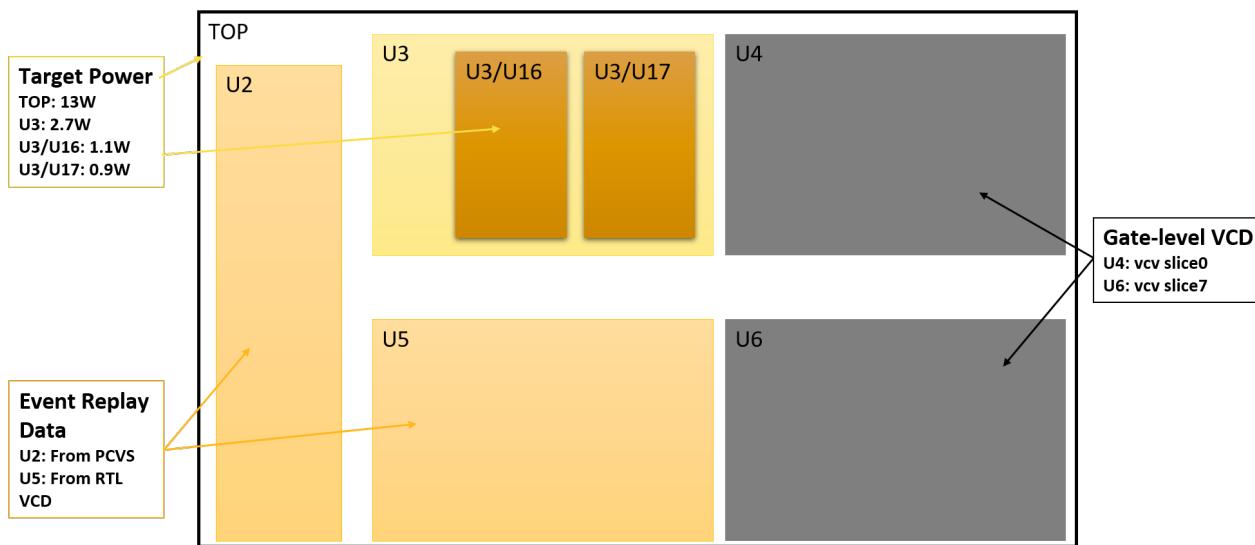
```

object_settings = {
    'leaf_instance_values' : [
        {'instances' : [Instance('inst1')],
         'switching_control' : {
             'output_pin_initial_state' : 'low',
             },
         'mode_control' : {
             'mode_sequence' : ['_active_mode_'],
             }
         },
        {'instances' : [Instance('inst2')],
         'switching_control' : {
             'output_pin_initial_state' : 'high',
             },
         'mode_control' : {
             'mode_sequence' : ['_active_mode_', '_inactive_mode_'],
             }
         }]}

```

6.3.11. Using Multiple Input Types Together

You can create a single NPV scenario by using different available input types for different blocks of a design as shown in the following figure.



The tool follows the following order of priority in descending order when handling different input types. The first item being of highest priority.

1. Leaf instance standard cell control or mode control for macros (deterministic switching)
2. Leaf Instance toggle rate (random switching)
3. Event replay file (deterministic switching)
4. VCD file (deterministic switching)
5. Constant signal input (deterministic switching). See [Disabling Honoring STA Constants](#) on page 162.
6. `always_active_clocks` setting (deterministic switching)
7. PowerView input, that is, IPF flow (random switching)
8. Instance Toggle Rate (ITR) flow (random switching)
9. `target_power` (random switching)
10. Toggle rate at non-leaf scope (random switching)

Disabling Honoring STA Constants

By default, the NPV ScenarioView considers constant inputs and does not switch instances with output pins having such constraints.

The tool can mark a logic pin having constant input in multiple ways. It can be set as constant from the STA file, can be a tie-high or tie-low cell output or tied to a PG net.

For the tool to not honor STA constant values, set the `use_sta_const` key to `False` as shown:

```
npv_settings = {'sta' : {'use_sta_const' : False}}
npv = db.create_no_prop_scenario_view(..., settings = npv_settings, ...)
```

Related information

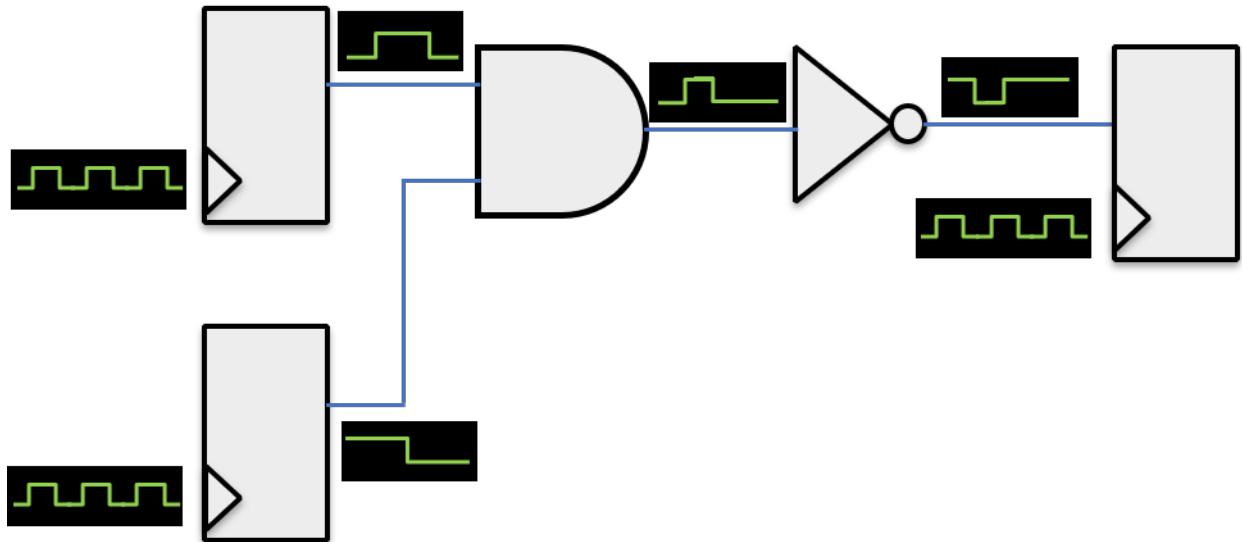
[Specifying Target Power](#) on page 150

[Event Replay Flow](#) on page 207

[Dynamic Analysis With Vectors](#) on page 237

6.4. Creating Logic Propagation Scenarios

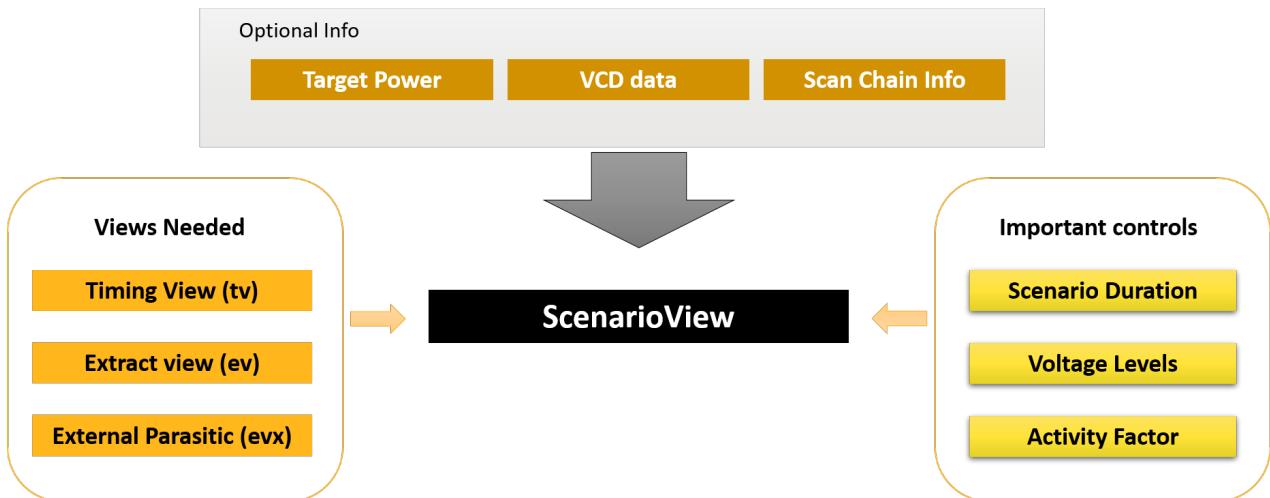
ScenarioView enables you to create logic-coherent events for each instance and propagates them.



ScenarioView has the following features and benefits:

- Uses net and pin connectivity information to propagate events based on the switching logic.
- Requires only DEF files and clock root information from SDC files for setup. This enables you to start the analysis early in the design cycle when STA information is not available.
- Includes in-built delay and transition time calculator to determine the exact switching time of each event.
- Can match target power by using the power-constrained logic-coherent scenario that is achieved by controlling integrated clock-gating (ICG) cell switching.
- Can perform vectorless scan-shift mode analysis.
- Is computationally intensive as large DEF blocks require more effort to process events across all instances.

The following figure shows the various inputs to create a vectorless scenario with logic propagation.



To create a vectorless scenario with logic propagation, use the `create_scenario_view` command as shown in the following example:

```
scn = db.create_scenario_view
      (timing_view=tv, external_parasitics=evx, extract_view = ev, **scn_args)

scn_args = dict(
    scenario_duration=48e-9,
    settings = {
        'pvt' : {voltage_levels : voltage_levels},
        'event' : {'default_clock_period': 8e-09},
        'object_settings' : object_settings_scn,
    }
    tag='scn',
    options=options)
```

Use the `scenario_duration` argument to specify the total time for which the scenario is required. `scenario_duration` is a required argument.

Creating logic propagation scenarios includes the following sections:

- [Input Views](#) on page 164
- [Propagating Clock Events](#) on page 164
- [Propagating Data Events](#) on page 165
- [Determining Event Arrival Time](#) on page 169
- [Determining Transition Time](#) on page 170
- [Inputting Current Waveform Text File for Instance](#) on page 172
- [Controlling Events per Signal With LSO](#) on page 173

Input Views

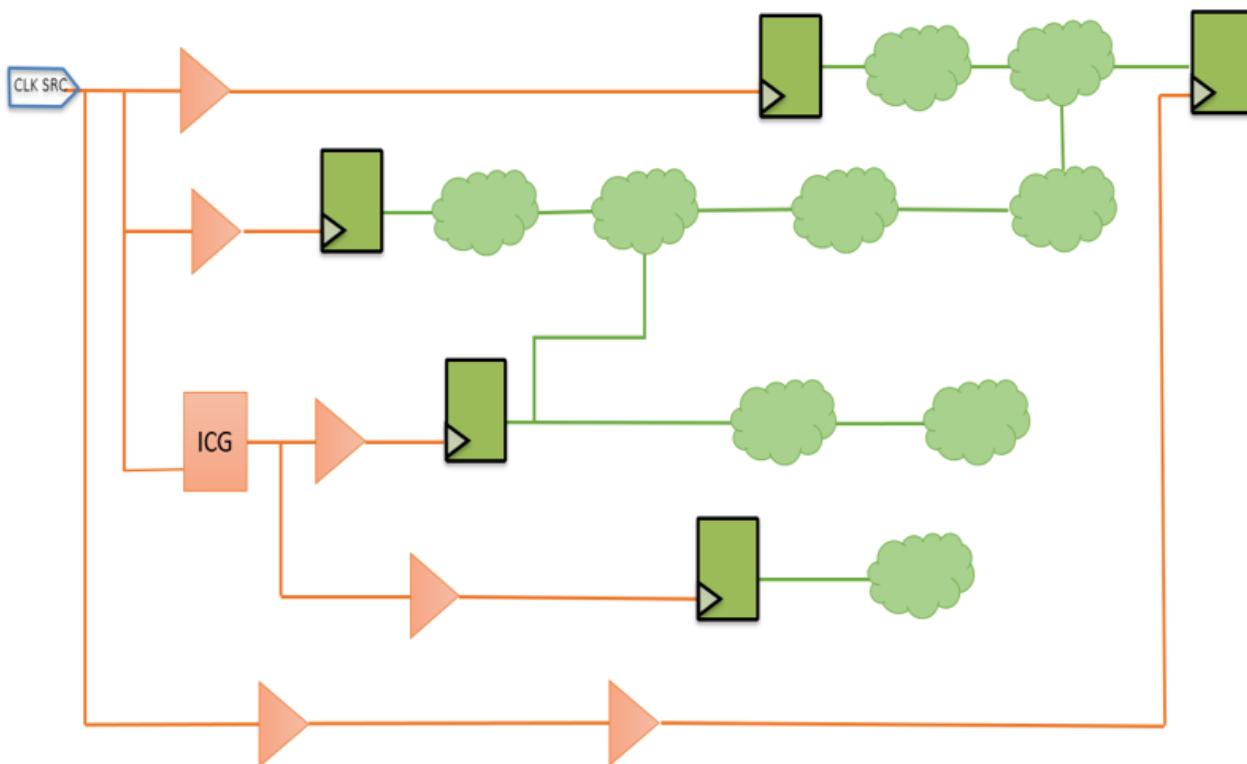
Provide the following input views to the `create_scenario_view` command:

- TimingView by using the `timing_view` argument. `timing_view` is a required argument.
- External Parasitic View by using the `external_parasitics` argument.
The view stores input SPEF data.
- ExtractView by using the `extract_view` argument.

For nets that are not covered in SPEF files, the tool considers the internally calculated net capacitances from ExtractView. To enable extraction on both PG and signal nets, set the `detail_extraction_net_type` key to `all` under the `settings` argument of the `create_extract_view` command.

6.4.1. Propagating Clock Events

By default, the `create_scenario_view` command propagates clocks as a first step to generate logic events. Clock sources defined in SDC or STA files are propagated and applied to the clock pins of design instances through the clock tree network. By default, all instances of the clock tree are on all the time. The following figure shows clock propagation through a clock tree network.



The tool uses the following precedence rules to propagate clocks.

- If the STA file is available, checks the clock source associations of each instance, and if required, corrects them by using the associations from STA.
 - Uses the frequency and duty ratio from the clock source definitions.
 - For multiple clocks at the same instance pin, uses the fastest clock.

6.4.2. Propagating Data Events

After propagating clocks, the `create_scenario_view` command propagates events through the data network.

Events can start from primary input pins and output pins of sequential elements, such as, register and macro output pins. These are called start points. By default, the tool generates random events at sequential output pins and propagates them through the combinatorial network. The sequential circuitry is not used to propagate events. In the vectorless propagation mode, the activity input that you specify determines the switching activity at start points.

Propagating events includes the following sections:

- [Initiating Activity at Primary Input Ports](#) on page 166
 - [Specifying Activity at Start Points](#) on page 166
 - [Using PowerView to Specify Activity at Start Points](#) on page 167
 - [Propagating Events through Combinational Network](#) on page 168

6.4.2.1. Initiating Activity at Primary Input Ports

By default, events are not generated at the primary input pins. To enable event generation at the primary input pins, set the Boolean `toggle_non_sequential_start_points` key to `True` with the `event` key under the `settings` argument as shown in the following example:

```
settings = {
    'event' : {'toggle_non_sequential_start_points' : True}}
scn = db.create_scenario_view(..., settings=settings, ...)
```

Defining Toggle Rates for Specific Primary Input Ports

You can define a specific `toggle_rate`, that is probability of switching, for each primary input port by using a PowerView. You must create the PowerView from a SwitchingActivityView.

In the following example, all ports of the design have events with a 20% probability (toggle rate of 0.2). However, the four ports with specific toggle rates switch with the probability specified in the SwitchingActivityView.

```
object_settings_swa = {'port_values' : [
    {'pins' : [Pin('IN[2)'), Pin('IN[3]')], 'toggle_rate' : 0.3},
    {'pins' : [Pin('AN[3)'), Pin('AN[4]')], 'toggle_rate' : 0.1}]
]
swa = db.create_switching_activity_view(..., settings = {'object_settings' : object_settings_swa}, ...)
pwr = db.create_power_view(..., switching_activity_view = swa, ...)
settings = {
    'event' : {'toggle_non_sequential_start_points' : True},
    'object_settings' : {'design_values' : {'toggle_rate' : 0.2}}}
scn = db.create_scenario_view(..., power_view = pwr, settings = settings, ...)
```

6.4.2.2. Specifying Activity at Start Points

To specify the activity input at start points, use the `toggle_rate` key for various scopes (such as, design and block) in the `object_settings` dict under the `settings` argument of the `create_scenario_view` command. `toggle_rate` specifies the probability of events at sequential output pins.

In the following example, the toggle rate of 0.2 applies to the entire design. However, the sequential elements of `blockC` switch twice that of the rest of design.

For logic with no associated clock in STA file, use the `default_clock_period` key in the `event` dict under the `settings` argument to specify the period of the default clock. The default is 1.0e-09 seconds.

```
scn_settings = {
    'pvt' : {
        'voltage_levels' : voltage_levels},
    'object_settings' : {
        'design_values' : {
            ... , 'toggle_rate' : 0.2, },
        'block_values': [ {
            'patterns' : 'blockC',
            'toggle_rate' : 0.4} ]},
    'event' : {}}
```

```
'default_clock_period' : 8e-09, })
scn_args = dict(..., settings = scn_settings)
```

There is no direct way to control activity factor per sequential instance. Providing PowerView input is an indirect way.

The following section describes how to specify activity for multi-bit flip-flops:

[Specifying Activity for Multi-Bit Flip-Flops](#) on page 167

Specifying Activity for Multi-Bit Flip-Flops

To specify the toggle rate of multibit registers or register banks, use the `mbff_toggle_rate` key in the event dict under the `settings` argument as shown in the following example:

```
settings = {...
    'event' : {'mbff_toggle_rate' : 0.45, ...}}
scn = db.create_scenario_view(..., settings=settings)
```

The tool uses the toggle rate of multibit flip-flops from this setting, and the toggle rate of other registers from the `activity` specification.

If you do not specify the `mbff_toggle_rate` key, the `activity` specification applies to multibit flip-flops as well. Both the `activity` and the `mbff_toggle_rate` value is applied to individual output pins in the same way.

6.4.2.3. Using PowerView to Specify Activity at Start Points

You can create a PowerView such that the `create_scenario_view` command can read the switching activity of each sequential start point from the PowerView. To do so, use the `power_view` argument with the `create_scenario_view` command as shown in the following example:

```
scn = db.create_scenario_view(timing_view=tv, external_parasitics=evx,
                             extract_view = ev, power_view = pwr, **scn_args)
```

For more information about creating a PowerView, see [Computing Power](#) on page 112.

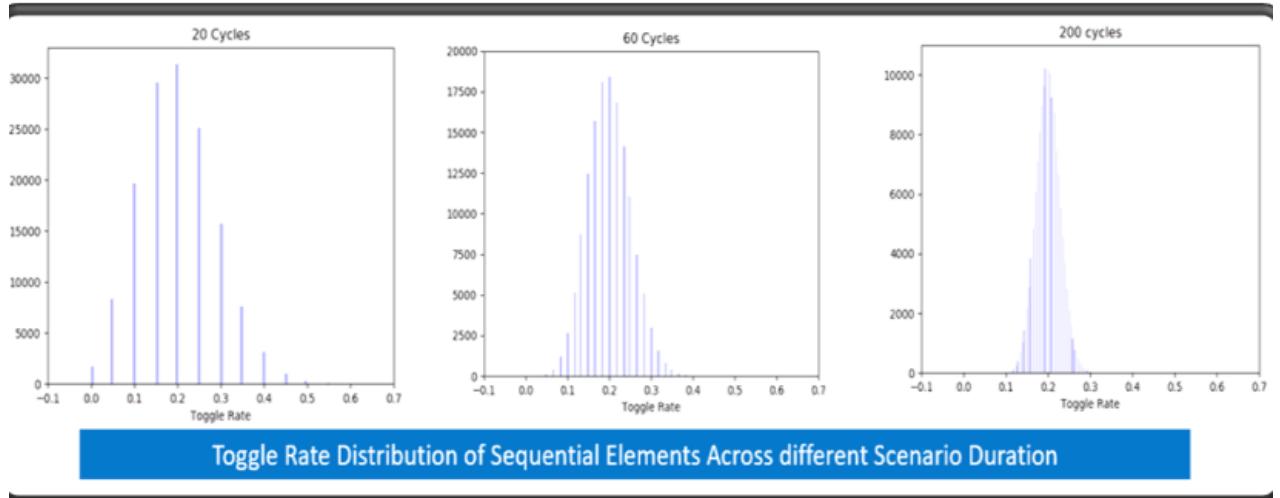
The `create_scenario_view` command reads only the activity factor of sequential start points (including macro instances) and no other data from the PowerView. So, a logic propagation vectorless scenario takes only the register toggle rate whereas a no propagation vectorless scenario meets the total power as well as the region-based power targets from the PowerView.

So, a register with toggle rate 0.25 in PowerView has 25% probability of switching in ScenarioView. Similarly, a toggle rate of 0.2 implies 20% probability of switching of each sequential start point (that is, each output pin of MBFF or macro) in each clock cycle. As the number of clock cycles increase, the number of design instances that actually toggle at the rate of 0.2 or closer increases. For more information, see [Toggle Rate Distribution at Start Points](#) on page 168.

Integrated clock-gating instances (ICGs) with toggle rate of 0 in PowerView are quiet in ScenarioView. The downstream instances also do not switch. If the activity of an ICG instance is greater than zero in PowerView, it is always on in ScenarioView. All clock instances are always on unless controlled by inputs to ScenarioView. Combinational instances do not have any dependence on power or activity from input PowerView.

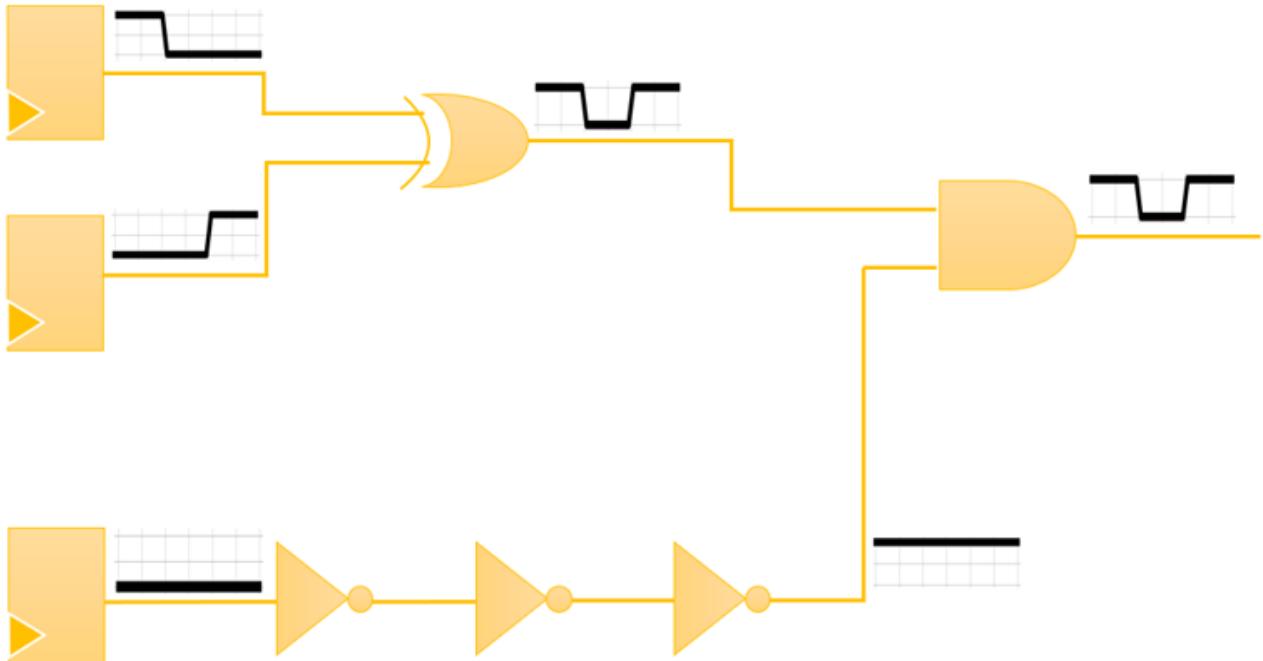
6.4.2.4. Toggle Rate Distribution at Start Points

The following figure shows how the toggle rate distribution at start points improve with increase in scenario duration. The behavior is truly random and follows the normal distribution curve. There can be statistical anomalies that reduce as the number of clock cycles increase.



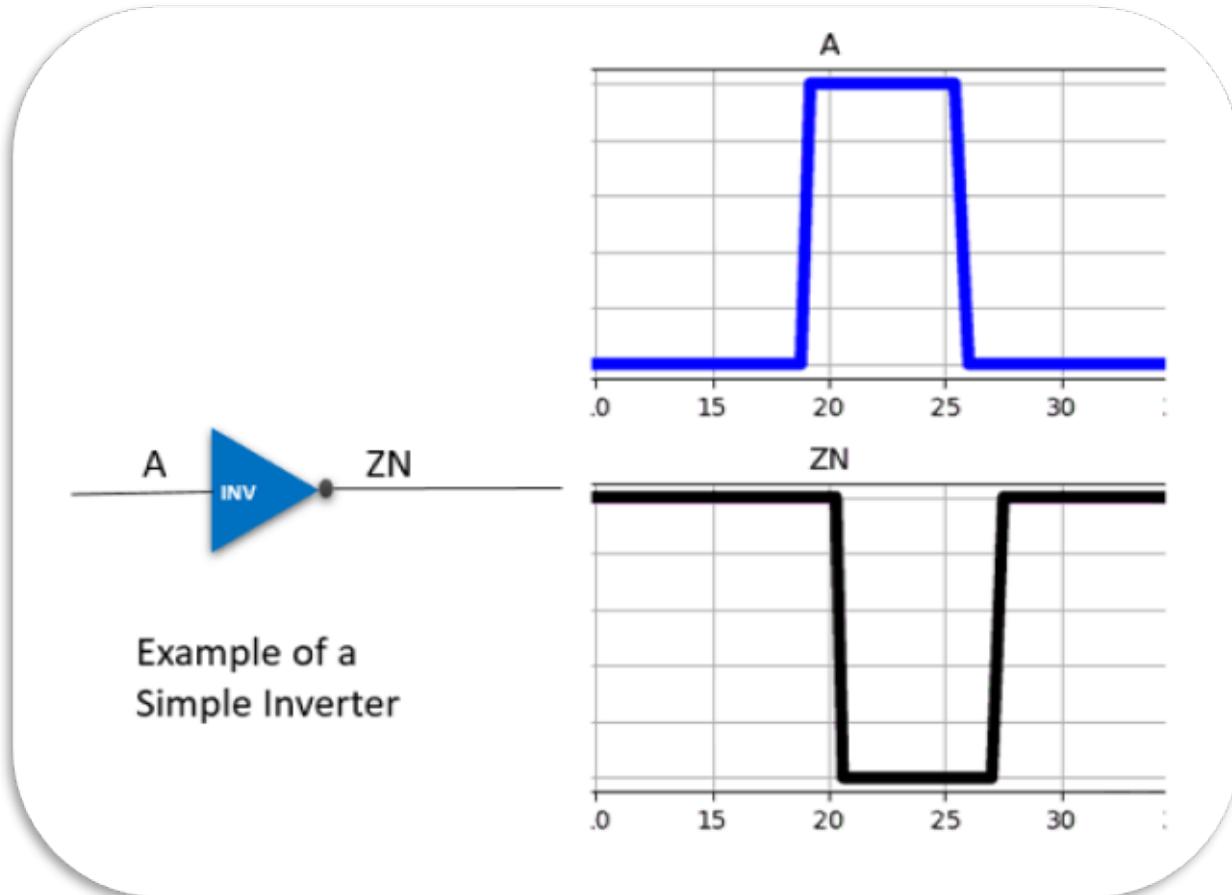
6.4.2.5. Propagating Events through Combinational Network

Events started at the sequential start points are propagated through the combinational elements based on their logic function defined in Liberty files as shown in the following figure.



6.4.3. Determining Event Arrival Time

By default, the tool determines the arrival time of an event at the instance output based on the input transition time and the propagation cell delay. The following figure shows a simple example of propagation delay between the input and output events.



The tool reads the cell delay from Liberty delay tables, such as non-linear delay models (NLDMs). To add net delay to the output event arrival time, set the `net_delay_type` key to `basic` with the `calculation` key under the `settings` argument as shown in the following example. The default `net_delay_type` is `off`. To determine the net delay, the tool reads the net's parasitic net from the SPEF file.

```
settings = {
    'calculation' : {'net_delay_type' : 'basic'}}
```

You can override the default event arrival time with corresponding values from the STA file, as described in the following section:

[Specifying Arrival Time Precedence](#) on page 170

6.4.3.1. Specifying Arrival Time Precedence

To override the default event arrival time with corresponding values from the STA file, use the `event_time_precedence` dict as shown in the following example.

```
sta_precedence = {
    'clock_instance' : ['sta'],
    'data_instance': ['sta'],
    'sequential_launch': ['sta'],
    'non_sequential_start_point': ['sta']}

scn_settings = {
    'object_settings' : {
        'design_values' : {
            'event_time_precedence' : sta_precedence}}}
```

Each specified key can have one or both of the two values:

- `propagated`

Default. The tool uses the calculated event arrival time.

- `sta`

The event arrival times from the STA file override the default `propagated`. If STA annotation is absent, the tool uses the `default propagated`.

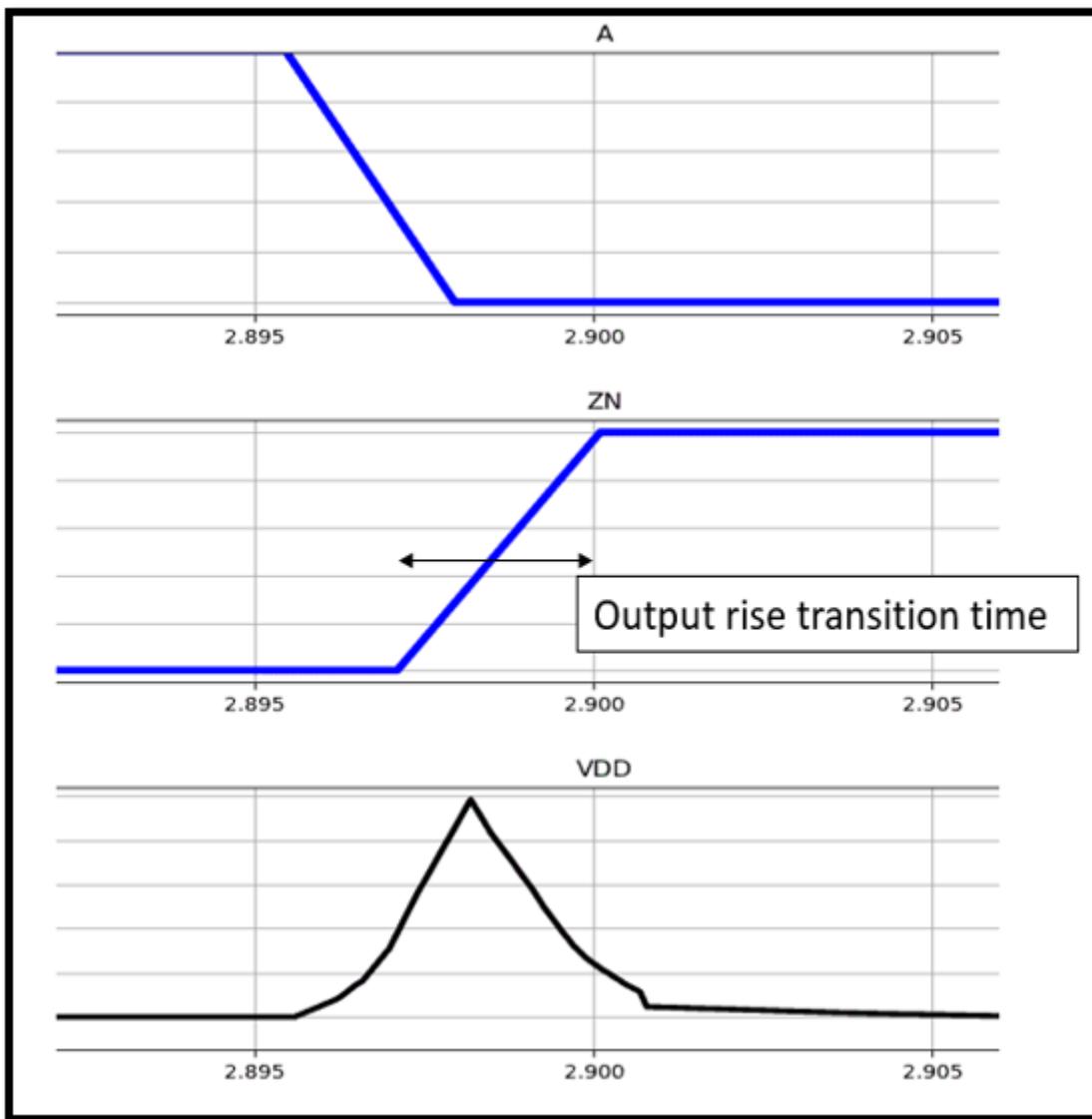
You can also combine `propagated` and `sta` usage. In the following example, clock switching event times are taken from the STA file while the data switching event times are from the `propagated` delay.

```
event_time_precedence = {
    'clock_instance': ['sta'],
    'data_instance': ['propagated'],
    'sequential_launch': ['propagated'],
    'non_sequential_start_point': ['sta']}
```

6.4.4. Determining Transition Time

By default, the tool determines the transition time of each event from Liberty transition time tables.

Note: The tool does not derate transition time across nets, that is, both the driver pin and the receiver pin see the same transition time.



You can override the default transition time with corresponding values from the STA file. To do so, use the `transition_time_precedence` dict as shown in the following example.

```
sta_precedence = {
    'clock_instance': ['sta'],
    'data_instance': ['sta'],
    'sequential_launch': ['sta'],
    'non_sequential_start_point': ['sta']}

scn_settings = {
    'object_settings' : {
        'design_values' : {
            'transition_time_precedence' : sta_precedence}}}
```

Each specified key in the dict can have one or both of the two values:

- propagated

Default. The tool uses the calculated transition time.

- sta

The transition times from the STA file override the default propagated. If STA annotation is absent, the tool uses the default propagated.

Use the `default_input_pin_transition_time` argument to specify the transition time for input instance pins with no data in the STA file (for example, a primary input pin). If you do not specify this argument, the default is 10 ps.

```
scn_args = dict(...,
    settings = { 'event' : {default_input_pin_transition_time = 5e-12}, ...}
)
```

6.4.5. Inputting Current Waveform Text File for Instance

Note: This feature is also applies to NPV ScenarioView.

You can specify piece-wise linear (PWL) current waveforms at PG pins of a small set of instances in a text file and input the file to ScenarioView. This is useful when a specific waveform is needed for some instances, such as when a cell has no APL, CCS power, or NLPM data.

The following example shows the format of such an input text file, **currents_for_1_2**:

```
currents_for_1_2:
$version = 1.0
$time_unit = 1e-12

@ip inst1/VDD {
( 0 , 3e-06 )

@ip inst1/VSS {
( 0 , 3e-06 )

@ip inst2/VDD {
(0, 2e-06)
(200, 2e-06)
(700, 14e-06)
(1400, 9e-06)
(2100, 2e-06) }

@ip inst2/VSS {
(0, -1.9e-06)
(200, -2.5e-06)
(700, -14e-06)
(1400, -9.2e-06)
(2100, -1.9e-06) }
```

The following example shows how to input the **currents_for_1_2** file to ScenarioView:

```
direct_current_input = [
    {'file_name' : 'currents_for_1_2'}]
scn = db.create_scenario_view(..., current_source_files = direct_current_input)
```

Note: You should limit the number of such pin waveform inputs to a few thousand entries.

6.4.6. Controlling Events per Signal With LSO

Logic signal override (LSO) is a method to override control and provide specific control per instance. LSO information overrides events derived from event propagation and VCD files.

An LSO file is an Ansys proprietary file format that contains explicit logic signal waveforms for each signal pin. You write the waveforms in piece-wise linear format. It includes transition time and arrival time information.

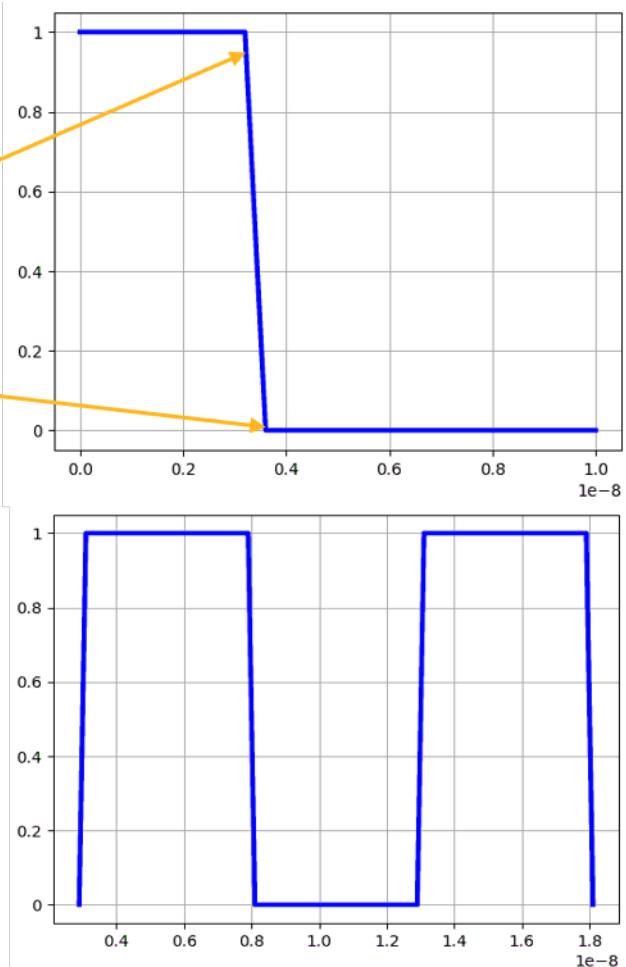
If required, you can query the transition time and arrival time for each pin from TimingView. The tool processes the contents of an LSO file in a manner that it reads an LSO file with a large number of instances in a small time.

```
$version=1.0
$time_unit=1.0

@ip FF1/Q {
(0.0,1)
(3.2e-09,1)
(3.6e-09,0)
}

@ip FF1/CK {
(0.0,0)
(2.9e-09,0)
(3.1e-09,1)
(7.9e-09,1)
(8.1e-09,0)
(12.9e-09,0)
(13.1e-09,1)
(17.9e-09,1)
(18.1e-09,0)
}

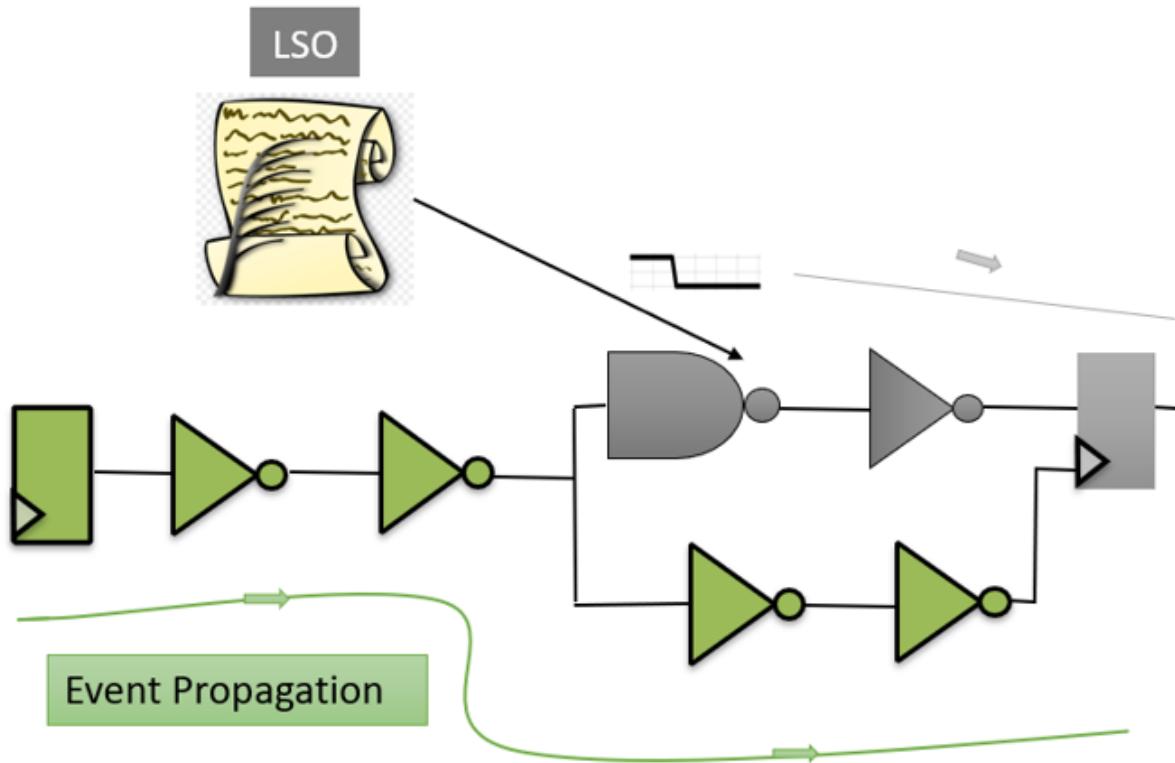
@ip ICG3/EN { (0.0,0) }
```



Events in LSO are propagated downstream from where they are applied, as shown in the following figure. When an LSO file is specified for a driver pin, the signal at the pin is applied to all the receiver pins that

connect to the same net. When an LSO file is specified for an input receiver pin, the override logic signal is not applied to the driver pin or other receiver pins that connect to the same net. However for power-gating instances, the LSO input to any pin is applied to all driver and receiver pins connected to the net.

In dynamic analysis with vector inputs, you can specify LSO for any logic inputs to overwrite the VCD annotation of that input pin. This enables you to control the instance state or mode with LSO input.



To read LSO files, use the `lso_files` argument as shown in the following example. The `file_name` key specifies the path to the LSO file. The `instance_name` key is optional and specifies the hierarchical instance name the LSO file applies to.

```
scn_args = dict(..., lso_files = ['file_name' : './new_override.lso'], ...)
```

6.5. Power Constrained Vectorless Scenario

Overview

PCVS is a unique feature of RedHawk-SC which can create a dynamic scenario that meets user defined power targets while creating logically consistent events. APIs and scripts are available to analyze and understand outputs. Total power contribution by each ICG and its downstream cone is estimated beforehand and using this data a set of ICGs are turned off to meet the target power.

PCVS creates multiple time frame scenarios with different ICG group selections in each frame to ensure maximum ICG coverage over the scenario duration. Multiple intervals can also be provided with different target power settings (transient flow) for each interval. It also honors user-defined Logic Signal Override (LSO) to control individual instance switching scenario. The scenario is seeded at sequential logic instances with proper clock arrival time and is propagated through downstream combinational logics.

This section describes all aspects of PCVS, the different flavours available, inputs/outputs, care abouts, and so on.

6.5.1. Features

The PCVS renders different features and facilities to the RedHawk-SC tool. The following features are discussed in this section:

- Target power control

The scenario is generated based on the target power input. The given target power is met separately for each frame. For details of frame length, see [PCVS Inputs](#) on page 178. There is flexibility to specify target power per domain and block level. PCVS is expected to meet the target power setting in +/-20% deviation.

- Logical consistency

PCVS creates events on sequential cells with respect to clock arrival time. Events generated at sequential cells are then propagated through downstream combinational logic. Therefore, logical coherence is met across the combinational cloud and clock path. For example, if a long buffer chain is activated at one end, all the buffers in the chain will be switching.

- ICG coverage

Different set of ICGs are selected in different frames, therefore maximizing the number of ICGs that are active. That way, a single scenario will have most of the ICGs switching, that is high switching coverage for ICGs.

- User defined toggle rate for sequential instances

For sequential instances in the design, you can define toggle rates. For example, 0.3 causes roughly 30% of sequential instances switching in a clock cycle.

- User-defined switching for specific instances

LSO is used to control individual instance switching scenario. You can input the LSO file where exact switching status and switching time for a specific instances can be input.

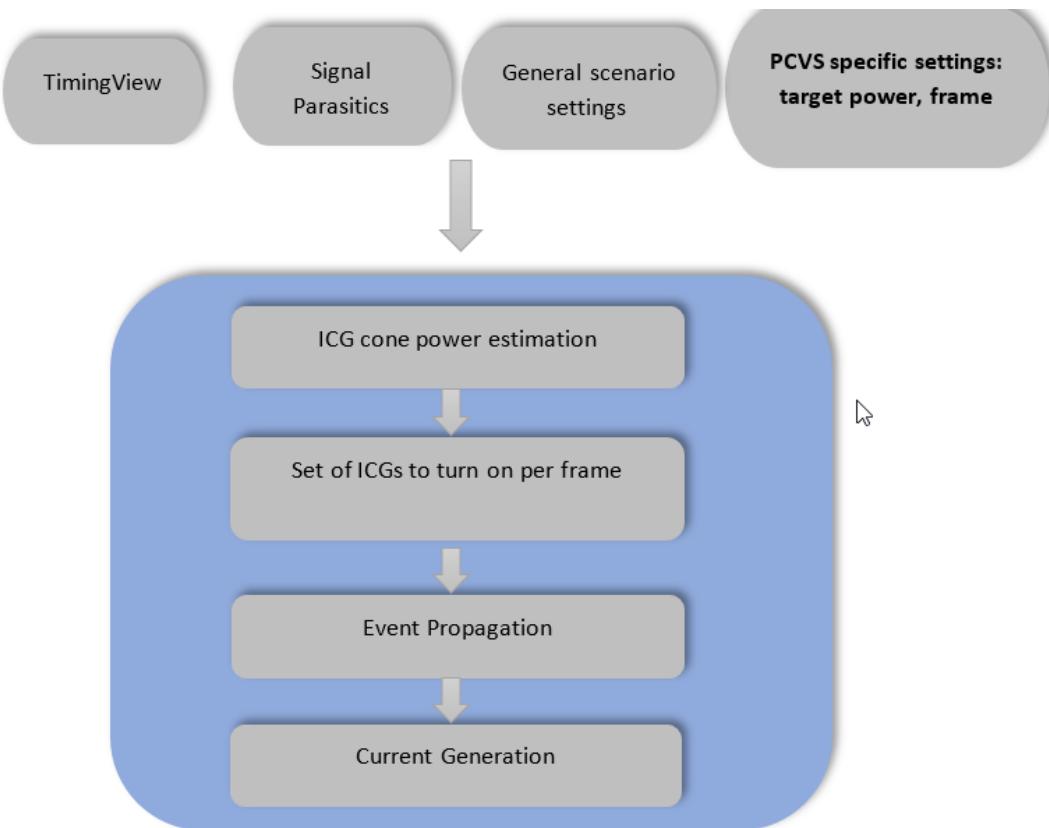
- Power Transient Scenario

- Calculates different power targets for a given duration.
- Analyzes impact of changing average current.

6.5.2. PCVS Flow

The following figure shows the high-level PVCS flow.

PCVS has complex flow with many internal stages involved. The various stages involved in the flow, concepts of frame and toggle rate, recommended target power range, and the performance/capacity impacts of the flow are explained in this section.

Figure 36. PCVS Flow

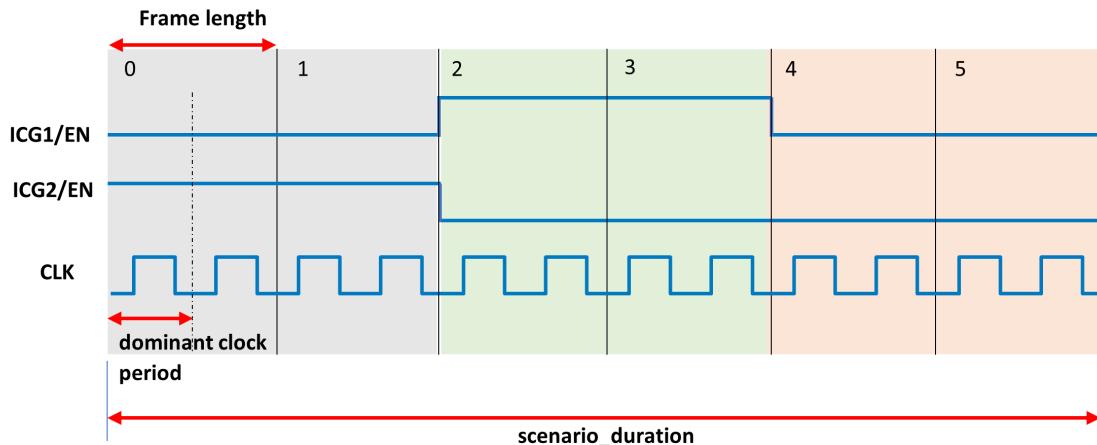
Concept of Frame

In PCVS, different set of ICGs are selected for each frame, which is similar to the NPV frame. You can use multiple clock cycles as one frame, giving chance for more flops to switch.

If you consider that there are 10 ICGs in the design, icg1, icg2, ...icg10. PCVS finds the sets of ICGs, that are switched together to meet the target power. The possible sets are icg1, icg4 and icg6 as set1, icg2, icg7 as set 2, icg3, icg5, icg8, icg10 as set 3, and so on. Each of these sets are turned on for one frame.

Therefore, frame is the duration for which a particular set of ICGs are kept on. Frame is a user input which can be one or two clock cycles of dominant clock.

More number of frames can enable higher ICG coverage. In the following figure, frame length is twice the dominant clock period and scenario_duration is five times the frame. PCVS enables a different set of ICGs starting at interval 0, 2, 4, and so on. Each ICG is on for two frames to cover rise and fall at flop Q pin.

Figure 37. Frame Size

Setting the Right Target Power

Setting the right target power is important as it gives the best result. High target power keeps all ICGs on and it moves away from user defined toggle rates. A low value causes modifying user defined toggle rates and also less number of ICGs on. Extremely low power can also end up being less than non-scalable power. No extra adjustment of sequential toggle rate is required. Following is an example of setting the right target power:

- All ICGs and all registers on all the time -> maximum power for Scenario: 100mW
- Leakage power: 5mW
- Clock power, including clock pin: 25mW
- Power from data, that is flop + combo: 70mW
- Activity factor of 0.2 -> power from data scale down one-fifth: 14mW
- $25+14 = 39\text{mW}$ -> controllable power.
- Target power within this range of 5-44mW (adding leakage power) is achieved by turning off some ICGs.
- Target power of 25mW is good -> achieved by turning off roughly half of ICGs.
- Target power of 50mW -> most ICGs turned on and raise flop activity.
- Target power of 10mW -> most ICGs off and lower flop activity.

Macro Settings

Typically, macros have high number of input pins. Power consumed by macros are dependent on the state of these input pins which would be coming from event propagation of upstream combinational cloud. The states of inputs are different between initial power estimation stage and the later power calculation stage. Therefore, power estimation can get skewed for designs with high percentage of macro power. Hence, it is advisable to set explicit modes for macros in the design using object settings.

Performance/Capacity Considerations

PCVS is typically run for a longer duration than most of the other vectorless runs. 5-10 frames are recommended to have all ICGs and sequential instances switching atleast once. This coupled with the two

internal scenarios get created in the flow, makes PCVS an expensive feature with respect to performance and capacity. In general, a PCVS ScenarioView takes ~2.5X times to run compared to a simple vectorless ScenarioView of the same scenario duration.

You should run PCVS flow for blocks and use event replay flow from these block level scenarios to do full chip power integrity runs.

6.5.3. PCVS Inputs

To specify target power for a PCVS scenario, use the `target_power` key under the `settings` dict of the `create_scenario_view` command. For example,

```
settings = {
    'target_power' = [ { 'power_targets': { Instance(<block_name>) :
        {<domain_name>:{ 'target_power' : <target_power>} },
        'toggle_rate_only':
        True|False, 'force_user_toggle_rate':True|False, 'icg_coverage_settings':
        { 'ensure_coverage':True|False, 'coverage_interval_duration':<value>,
        'power_threshold_for_dominant_frequency':<value>},
        'power_interval_duration': <value> }, ... ]
}
scn = db.create_scenario_view (tv, ev, ..., settings=settings)
```

Specifying `power_targets` is mandatory. The different types of keys are explained in the following tables:

Table 5: PCVS Input Keys

Key	Description
<code>power_targets</code>	A dict of type { <Block#> : { <DomainNet#> : {<target_power>:<power_in_watts>},...} } <Block#>: Specifies the target block. Use Instance ("") to specify target power for top block. <DomainNet#>: Specifies domain net to be targeted for <Block#>. Use '*' to specify overall target power for all domains in <Block#>.
<code>toggle_rate_only</code>	States if the scenario is True/False. The default value is False If True, ICG control is disabled and the target power is achieved only by tweaking state point toggle rate. If False, ICG control is enabled and state point toggle is the secondary method to achieve target power.
<code>force_user_toggle_rate</code>	States if the scenario is True/False. The default value is False If True, tool does not tweak user toggle rate. Hence, only ICG control is used. If False, ICG control is used and state point toggle is the secondary method to achieve target power. It should not be True if ' <code>toggle_rate_only</code> ' is True.

Key	Description
power_interval_duration	<p>value_type : float</p> <p>This specifies the duration for which these settings can be applied. If this key is given, 'scenario_duration' key is ignored. This key is optional if only one dict of target_power is given. This argument is must if more than one dicts are given in the target_power, in which case sum of 'power_interval_duration' in each dict would be the scenario duration.</p>
icg_coverage_settings	These settings are used for achieving ICG coverage.

Table 6: icg_coverage_setting Details

Key	Description
ensure_coverage	<p>States if the scenario is True/False. The default value is True.</p> <p>If True, the tool tries to achieve ICG coverage. Whole scenario_duration or power_interval_duration (if given) is divided into frames (or, coverage_intervals), where tool gates different set of ICGs in each frame to achieve ICG coverage and target power over complete scenario_duration. Duration of each frame is determined by key 'coverage_interval_duration'.</p> <p>If False, the tool does not try to achieve ICG coverage, and only one set of ICGs are gated for the entire scenario_duration to achieve target power. It should not be True if 'toggle_rate_only' is True.</p>
coverage_interval_duration	You can directly give value of frame duration here. The default value is calculated internally.
power_threshold_for_dominant_frequency	<p>The value is in range of 0 to 1. The default value is 0.9. This value is used to calculate coverage_interval_duration internally. This value represents the percentage of power tool should consider for determining the dominant frequency.</p> <p>For example, if the value is 0.9, the tool gathers the frequencies which controls 90% of the total power. Out of these frequencies, the maximum period is used as coverage_interval_duration. This is ignored if the 'coverage_interval_duration' is given.</p>

Top level Target Power Setting

Use Instance ("") to specify target power for top block and '*' to specify overall target power for all domains as shown in the following example:

```
settings = {'target_power' = {
    'power_targets' : {Instance('') : {'*' : {'target_power': 3.5 }}}}}
```

Block Level and Power Domain Specific Target Power Setting

You should use Instance (<block_name>) to specify the target blocks and <domain_name> to define domain specific target power as shown in the following example:

```
settings = {'target_power' =
{'power_targets':{Instance('u1'): {Net('VDD1'): {'target_power': 0.1},
Net('VDD2'): {'target_power': 0.3}},
Instance('u2'): {Net('VDD3'): {'target_power': 0.05}}}}}
```

Controlling Frame Size

The coverage_interval_duration key is related to frame size. By default, this is automatically determined based on the dominant clock period. Following are the two ways to control this:

1. By setting the key coverage_interval_duration

```
settings = {'target_power' = {
'power_targets':{Instance(''): {'*':{'target_power':0.4}}},
'icg_coverage_settings': {'coverage_interval_duration':1e-9 } }}
```

2. By changing the key power_threshold_for_dominant_frequency

This value represents the percentage of power tool considers for determining the dominant frequency. The tool gathers all frequencies which control the given threshold of total power. Out of these frequencies, the maximum period is used as coverage_interval_duration. This setting is ignored if coverage_interval_duration key is explicitly set.

Example:

```
settings = {'target_power' = {
'power_targets':{Instance(''): {'*':{'target_power':0.4}}},
'icg_coverage_settings': {'power_threshold_for_dominant_frequency':0.8
} }}
```

If there are three frequencies in a design, 100MHz, 500 MHz, and 900MHz and their powers are 5%, 35% and 60%, respectively. In this scenario, 900MHz is the most common frequency. However, dominant frequency as per definition is the frequency above which power_threshold% (80%) of power resides. In this case, it is 500MHz, as adding up powers of frequencies above 500MHz (that is, 500MHz and 900MHz) gives 95% power.

Power Transient Analysis

Rather than one single target power for a level (say top level), PCVS provides the flexibility to come up with a transient power Scenario where different power numbers can be given across time. You can come up with a ScenarioView where power/currents go from high to low or low to high or even high, low, and then high again.

Multiple target powers can be specified as a list of dicts, as shown in the following example:

```
settings = {'target_power' =
[{'power_targets':{
Instance(''): {
Net('VDD1') : {'target_power':0.1},
Net('VDD2'):{'target_power' : 0.3} },
```

```

        Instance('u2'):{
            Net('VDD3'):{'target_power':0.05}},
        'power_interval_duration': 2.5e-9 },
        {'power_targets':{
            Instance(''):
                Net('VDD1'):{'target_power' : 0.5},
            Instance('u2'):{
                Net('VDD3'):{'target_power' :0.5}}},
        'power_interval_duration': 3e-9 },
        {'power_targets':{
            Instance(''):
                Net('VDD1'):{'target_power' : 2.5},
            Instance('u2'):{
                Net('VDD3'):{'target_power' :0.5}}},
        'power_interval_duration': 3.5e-9 }]
    }
}

```

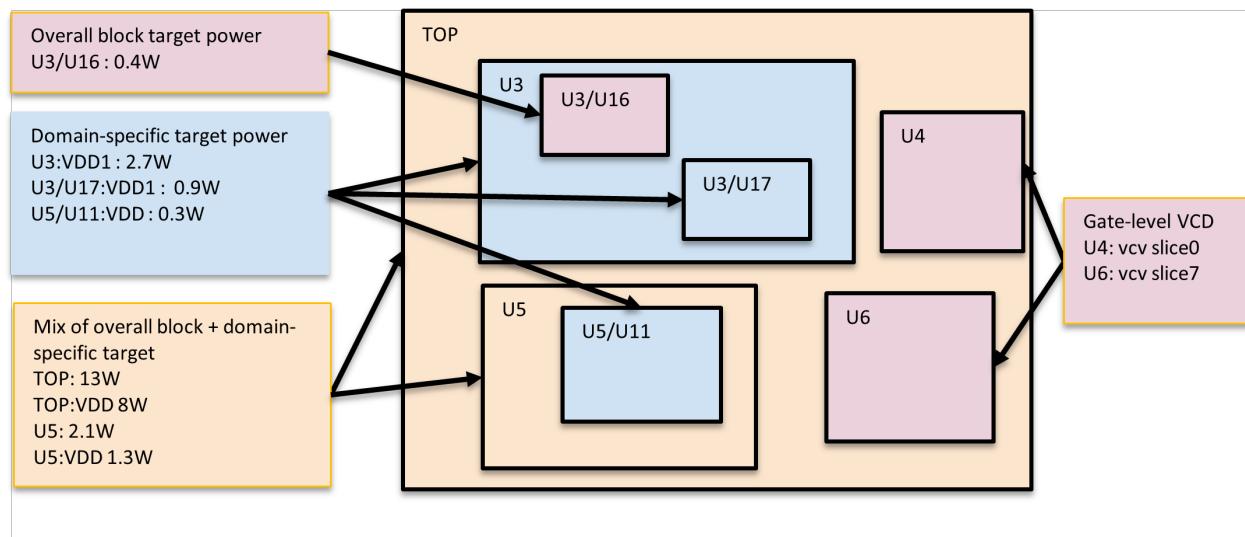
The `power_interval_duration` key specifies the duration for which each `power_targets` settings are applied. This key is optional if only one dict of `power_targets` is specified.

The `power_interval_duration` key is mandatory if more than one dict is specified in `target_power`. The sum of `power_interval_duration` in each dict is the `scenario_duration` for `ScenarioView`. When you specify this key, the tool ignores the specified `scenario_duration`.

Mixed Flows

PCVS supports mixed flows, that is, gate-level VCD/FSDB for some blocks and power constraints for the others. If a block has both VCD as well as target power setting, then the target power is ignored. Following diagram shows the mixed flow followed by an example:

Figure 38. Mixed Flows in PCVS



```

'power_targets' : {
    Instance('') : { '*' : { 'target_power': 13},
                    Net('VDD') : {'target_power':8} },
    Instance('U3') : {Net('VVDD1') : { 'target_power': 2.7}},
    Instance('U3/U16') : { '*' : { 'target_power': 0.4 } },
    Instance('U3/U17') : {Net(VDD1') : { 'target_power': 0.4}},
}

```

```

Instance('U5') : {'*' : { 'target_power': 2.1},
                  Net('VDD') : {'target_power':1.3} },
Instance('U4') : {'*' : { 'target_power': 1.1}}, # ignored since U4 has VCD
}

```

Target power is assigned from bottom to top. Target power given for a block is the cumulative target for that block and all its child blocks. If a child block is a VCD block, then its power is subtracted from the target power. Also, the tool expects the VCD blocks to have full coverage. If the VCD block has VCD for only 5-10% instances and rest 90-95% are vectorless, then PCVS results may not be achieved. For large designs, you should run PCVS on blocks, event replay flow, and then use these block FSDBs in full chip run.

6.5.4. PCVS Outputs

This section describes some of the outputs from ScenarioView that are useful from PCVS context. This includes current waveforms, script reports, log messages, and so on.

Power Summary

The first information that comes from a ScenarioView with the target power is the power numbers. You can use the API `emir_reports.write_instance_power_report_and_summary` to get this power summary.

Example:

```
>>> emir_reports.write_instance_power_report_and_summary(scn)
```

Figure 39. Power Summary

```

**** RedHawk-SC Power Summary Report ****

Created: Thu Mar 19 04:51:16 2020

A total of 685824 instances were summarized for this report while 311463 were omitted due to missing power data (54.59% coverage).

A total of 0 pins were omitted because they were not attached to a power domain.

***

grouping          clock_pin_power(W)  internal_power(W)  leakage_power(W)  switching_power(W)  total_power(W)  percent_power(%)  pin_count  instance_count
*** Power Domains:
VDD              0.20151        0.40421        0.081053        0.4834        0.96866       100.00      374361     374361
Total            0.20151        0.40421        0.081053        0.4834        0.96866       100.00      374361     374361
*** Frequency Domains:
2.5e+09          0.2004        0.39659        0.079985        0.47976       0.95634       98.73      368317     368317
0                0.0011036    0.0076103    0.0010681    0.0036337    0.012312      1.27       6044      6044
Total            0.20151        0.40421        0.081053        0.4834        0.96866       100.00      374361     374361
*** User Defined Groups:
combinational logic 0.034191    0.18627        0.048797        0.35148       0.58655       60.55      321479     321479
sequential logic   0.16732       0.21794        0.032256        0.13192       0.38211       39.45      52882      52882
Total             0.20151        0.40421        0.081053        0.4834        0.96866       100.00      374361     374361
**** End Report ****

```

Switching Coverage

You can query frame-wise switching coverage using the following script. The information on the percentage of ICGs and register switching per frame is also available. Cumulative numbers are also available from this script. You can even plot the switching coverage heatmap in GUI and save it as a UserView in the database.

Syntax:

```
import scenario_utils
scenario_utils.get_switching_coverage(scn, file_name='switching_coverage.rpt',
plot_gui=False, frame_timings=None, frame_length=None,
macro_disable_state_names=[],
db=None, tag='sw_maps', block_domain_filter=None, return_dict=False)
```

The arguments are explained in the following table:

Table 7: Switching Coverage Arguments

Argument	Description
Scn	Input ScenarioView (type=ScenarioView, required=True)
file_name	The default value is switching_coverage.rpt, file name of output file (type=object, default_value='switching_coverage.rpt')
plot_gui	The default value is False (type=bool, default_value=False)
frame_timings	The default value is None. This is used to find the coverage for a particular interval. For example, <i>frame_timing = [(1e-9, 1.3e-9)]</i> (type=list, default_value=None)
frame_length	The default value is None. (type=list, default_value=None) <p style="text-align: center;">Note: frame_timings and frame_length should not be given together</p>
macro_disable_state_names	The default value is []. List of string of state names which are considered as disabled state for macros (type=list, default_value=[])
Db	The default value is None, saves Mpas to db (type=object, default_value=None)
Tag	The default value is sw_maps (type=str, default_valuetag=)
block_domain_filter	Dict of list of block/domain names as strings for which coverage is reported (type=dict, default_value=None)

Example:

```
scenario utils.get switching coverage(scn,plot gui=True, frame length=1.2e-9)
```

The following example shows the script output:

```
+-----+ Switching Coverage Report +-----+
Switching Coverage for Block: '' Domain: '*'
+-----+-----+-----+-----+
| Cell Type | Total | No Logic | Power Disconnect |
+-----+-----+-----+-----+
| Clock | 617 | 0 | 0 |
+-----+-----+-----+-----+
```

ICG	8450	0	0
Sequential	52882	0	0
Combinational	312412	0	0
Total	374361	0	0

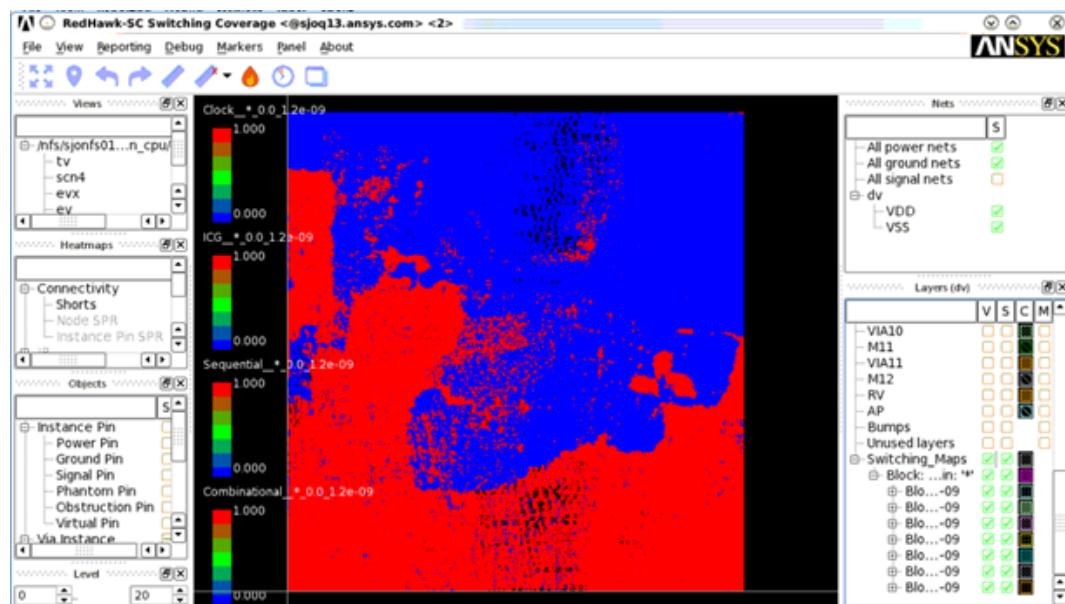
Note: Instances reported under 'No Logic' have empty scn.get_instance_events due to various reasons like logical disconnects, missing lib, STA coverage issues (for NPV), etc. So, 0 instances won't be considered in the switching coverage reported below:

Cell Type Wise Coverage for Frame: 0.00s - 1.20ns

Cell Type Cumulative %	Total	Switching	Switching %	Cumulative
Clock 79.9 %	617	493	79.9 %	493
ICG 45.33 %	8450	3830	45.33 %	3830
Sequential 35.56 %	52882	18805	35.56 %	18805
Combinational 23.03 %	312412	71947	23.03 %	71947
Total 25.4 %	374361	95075	25.4 %	95075

This is for frame duration (0-1.2ns). The script gives similar details for all frames. Following figure shows the GUI snapshot of the same (enabled by 'plot_gui' argument):

Figure 40. GUI Snapshot of Switching Coverage



Frame-wise Power

User can check frame-wise power numbers using the following script.

Syntax:

```
import scenario_utils
scenario_utils.get_power_data(scn, file_name='power_data.rpt',
frame_timings=None,
frame_length=None, cumulative=False, block_domain_filter=None,
return_dict=False)
```

Table 8: Power Arguments

Argument	Description
Scn	Input ScenarioView (type=ScenarioView, required=True)
file_name	The default value is power_data.rpt, file name of the output file is (type=object, default_value='power_data.rpt')
frame_length	The default value is None (type=list, default_value=None) <div style="background-color: #ffffcc; padding: 10px; margin-top: 10px;"> Note: frame_timings and frame_length should not be given together </div>
Cumulative	The default value is False. It shows cycle by cycle coverage as cumulative (type=bool, default_value=False)
block_domain_filter	Dict of list of block/domain names as strings for which coverage is reported (type=dict, default_value=None)
return_dict	The default value is False. It returns the stats as dict when set to True (type=bool, default_value=False)

Example:

```
scenario_utils.get_power_data(scn, frame_length=1.429e-09)
```

The following is the script output:

```
+----- Power Report -----+
Power Report for Block: '' Domain: '*'
Cell Type Wise Power for Frame: 0.00s - 1.20ns
+-----+-----+-----+
| Cell Type | Power | Percentage % |
+-----+-----+-----+
| Clock | 3.9835e-02 | 4.24 % |
| ICG | 9.8451e-02 | 10.49 % |
| Sequential | 3.4678e-01 | 36.95 % |
| Combinational | 4.5357e-01 | 48.33 % |
| ----- | ----- | ----- |
```

Total	9.3856e-01	100 %
Cell Type Wise Power for Frame: 1.20ns - 2.40ns		
Cell Type	Power	Percentage %
Clock	4.0252e-02	4.09 %
ICG	1.0128e-01	10.28 %
Sequential	3.8213e-01	38.79 %
Combinational	4.6153e-01	46.85 %
Total	9.8512e-01	100 %

Demand Current Plots

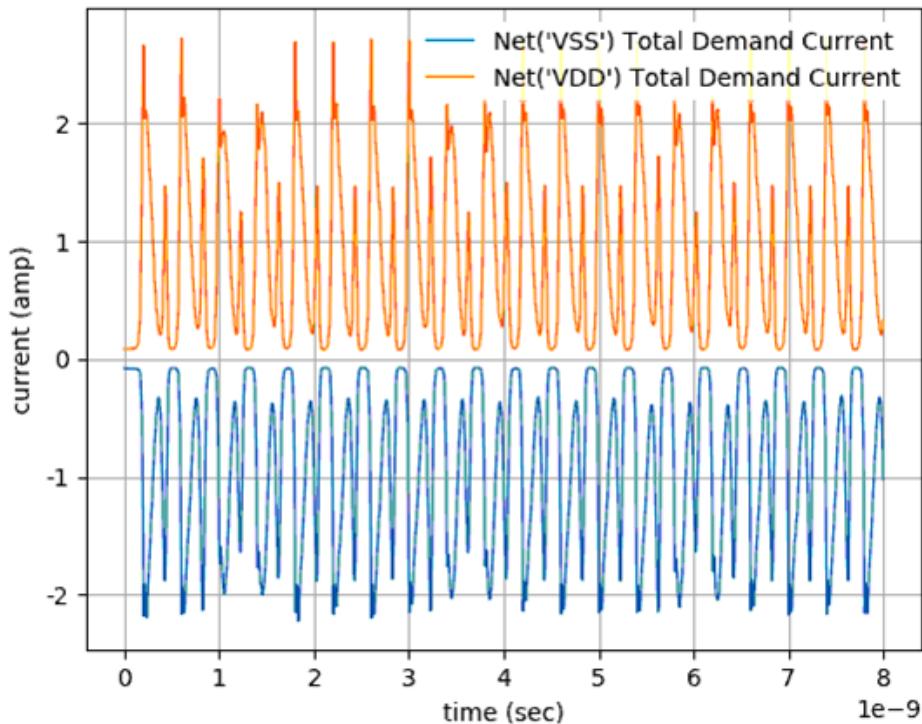
<ScenarioView>.get_total_demand_currents API returns total demand current waveform per net. The output is a Waveform object which can be viewed with the function plot.

Example:

```
plot(scn.get_total_demand_currents)
```

The following figure shows the total demand current:

Figure 41. Total Demand Current



Further, the script `scenario_utils.get_block_currents` returns the demand current category wise. For the input ScenarioView, per domain, per block, and per cell type demand currents are available as output.

Syntax:

```
import scenario_utils
scenario_utils.get_block_currents(scn, nets=[Net('VDD')], rollup_virtual=True)
```

This returns a dict of dicts with format:

```
{Net('<net_name>') : {Instance('<hierarchycy_name>') : {'<type>' : Waveform
, }, }, }.
<Type> will be : 'comb', 'seq', 'clk', 'icg', 'macro' or 'total'
```

The following table lists the arguments:

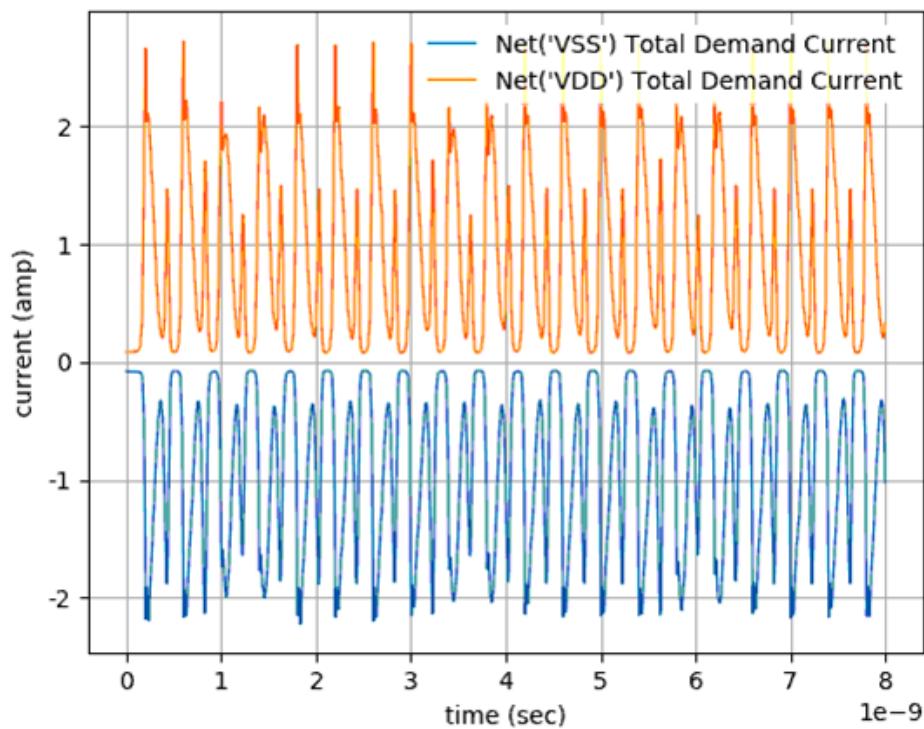
Table 9: Demand Current Arguments

Argument	Description
Scn	Input ScenarioView (type=ScenarioView, required=True)
Nets	List of power or ground net objects (type=list, default_value=[Net('VDD')])
rollup_virtual	Choose whether to rollup the virtual net current into its parent net's total at the top (type=bool, default_value=True)

Example:

```
block_curr = scenario_utils.get_block_currents(scn)
plot(block_curr[Net('VDD')][Instance('')].values())
```

The following figure shows the demand current plots:

Figure 42. Demand Current Plots

Reviewing Log Messages

The target power and other related inputs of PCVS can be bad, which causes desired power not being met or low ICG/flop coverage. The messages in the log file can help in understanding such bad inputs. The following are some of the log messages:

INFO Messages regarding power, frame size, and so on

The INFO message PCV.119 gives information about the non-scalable power and also the maximum power of the design as shown in the following example:

```
INFO<PCV.119> Non scalable power is 6.63128441996
and maximum power is 26.5731958327.
```

The message PCV.120 gives information about the frame size, number of frames, and so on as shown in the following example:

```
INFO<PCV.120> Frame size: 4.00000005341e-10,
number of frames: 20, scenario_duration: 8.00000010682e-09.
```

Common Warning Messages

Case 1: When target power is more than maximum achievable power

The tool issues the following warning message when the target power given is more than the maximum achievable power, which is by turning on all ICGs and registers.

```
"PCV.123 Target_power_more_than_base_power", 'msg' : "Target power $target_power
is
more than max_power $base_power, for block Instance($inst_id)"}
```

There is also a related error PCV.103, which appears when a domain specific target power is more as shown in the following example:

```
'target_power' = { 'power_targets': { Instance('') : { '*' : { 'target_power'
: 1.9} } } }
```

Warning in log:

```
WARNING<PCV.123>      Target power 1.9 is more
than max_power 1.85655166065, for block Instance('<block_name>').
```

Case 2: When target power is small

The tool issues the following warning:

```
PCV.113 Target_power_less_than_non_scalable", 'msg' : "Overall top level target
power:$target_power is less than overall non scalable
power:$non_scal_pwr.",
```

In this case, the input target power goes in the opposite direction, rather than too high, it became too less. Non-scalable power refers to leakage power plus the power of clock network not controlled by any ICG as well as powers for macros. You can have similar messages of PCV.104 and PCV.107, which deals with less target power for blocks and domains.

Example:

```
'target_power' = { 'power_targets': { Instance('') : { '*' :
{ 'target_power' : 0.08 } } } }
```

Warning in log:

```
WARNING<PCV.113>      Overall top level
target power:0.08 is less than overall non scalable
power:0.0963413987105..
```

Case 3: When multiple levels of target power are specified, but powers set are not consistent

```
"PCV.125 Target_power_less_than_child", 'msg' : "Target power $target_power
is less than
sum of child blocks/domains target power $sum_child_pwr for block
Instance($inst_id)",
```

Here, the block level target powers add upto XX and the top level target power is YY. You can have similar message of PCV.126 when domain specific target powers add upto more than top level domain power.

Example:

```
'target_power' =[ { 'power_targets': {
Instance('') : { '*' : { 'target_power' : 8 } },,
Instance('U1/a') : { '*' : { 'target_power' : 3 } },
Instance('U1/b') : { '*' : { 'target_power' : 3 } }}
```

```

} } ]
ERROR<PCV.125> Target power 8 is less than sum of child blocks/domains
target power 9.63224619187 for block Instance(0).

```

6.6. Vectorless Scan Flow

Overview

Typically, IR drop analysis is done for the normal functional modes. DFT or scan mode is not considered for DvD as the frequency of operation is lesser. However, in scan mode, simultaneous switching of large number of registers give rise to high peak current and di/dt effects, which when coupled with package inductance is bound to introduce IR drop issues.

Note: ATPG creates high level of activity by design (not only on registers but in the combinational logic) as it is trying to test the maximum number of faults in the fewest number of vectors.

Scan mode analysis is required to capture high peak current and di/dt effects seen when activating scan chains in an SoC design. It can also capture secondary effects such as unintentional switching in the data path. Typically, this flow requires a scan mode gate VCD to annotate the activity on the scan flops. However, RedHawk-SC offers the option to perform vectorless scan mode analysis, where the flow is set up with early-stage design data (no need of gate VCD).

Gate-level timing-annotated VCD with scan activity is usually available late in design cycle. Evaluating potential issues such as IR drop due to simultaneous clock switching and scan functionality can be achieved (based on initial placement) using the scan methodology. Here, RHSC generates the scenario from constraint file specifications, can avoid many cases of chip failure and yield reduction. It is, therefore, advisable to run scan mode analysis on top of regular functional analysis.

The vectorless scan mode in RedHawk-SC uses the pattern provided in the scan chain file to come up with the activity at the scan chain outputs. The delay offered by each cell in the scan chain is picked up from the liberty file. It must be noted that the pattern is treated as golden and will be honored at every point in the chain. This means, for example, that inversion logic in the path will not flip the pattern for cells downstream.

For registers with output pins for scan mode operation, the current computation for simultaneous switching of scan-only output pin with other regular pins uses the clock to scan-only output pin arc current from APL. This applies to both APL current characterization and ScenarioView demand current creation.

6.6.1. Scan Constraint Files

Vectorless scan mode analysis requires one or more scan chain files and also the regular input collateral needed for dynamic analysis. Scan constraint files contain a detailed description of each scan chain in the design, which includes the chain name, a list of instance-pins per chain in sequential order, the pattern to be used in this chain, the scan clock name and other modifiers that affects the output sequence. The modifiers that may be specified are:

- Pattern: The pattern to be shifted into the registers in this scan chain (a string, for example, '11010').
- Start Shift: The number of shifts already completed before starting the simulation. If negative, there is a delay before the first pattern bit is introduced.
 - Typically, this is expected to be a negative value indicating the number of cycles to be completed before the first shift-in operation.

- o Additionally, if the start shift is a negative number having a higher magnitude than the length of the scan chain (scan chain is 10 flops long, start_shift is -15), it will be reset to 0.
- Previous Pattern: The bit/bits used until start_shift is completed. This is applicable when start_shift is a negative number.

RHSC scan chain files are expected to be defined in a JSON file. You can provide separate JSON files for each scan chain, or a single JSON file with multiple chains defined in it, or multiple JSON files with many scan chains described in each file. In case of a broken chain, the breakpoint instance will be assigned some random activity based on the scenario activity settings. If the scan chains are accurate, it is advisable to keep activity level to 0 to avoid unwanted switching.

Format of JSON File

The JSON syntax is strict regarding the placement of commas, double-quotes, and braces. Refer to the following example:

```
# import json
# with open('scan_chain.json', 'r') as scan_chain_file:
#     json.load(scan_chain_file)

{
    "chain_normal": {
        "instance_pins": [
            "reg1/Q",      # each element is separated by a comma
            "reg2/Q",
            "reg3/Q",
            "reg4/Q",
            "reg5/Q"       # no comma after the last element
        ],
        "pattern": "101",
        "previous_pattern": "0",
        "scan_clock": "clk_scan",
        "start_shift": 0           # no comma after the last element
    },
    "chain_start_shift": {
        "instance_pins": [
            "reg6/Q",
            "reg7/Q",
            "reg8/Q",
            "reg9/Q",
            "reg10/Q"
        ],
        "pattern": "101",
        "previous_pattern": "0",
        "scan_clock": "clk_scan",
        "start_shift": -1
    }
}
```

Converting Third-party Scan Chain Files

AE scripts are available to convert scan chain files from third-party tools to an RH-style scan chain format. Another script is available to convert an RH-style scan constraint file to the RHSC json file.

Following script is used to convert 3rd party format to RedHawk-SC compatible format:

```
<RHSC_build>/gps/pythonpkgs/thpkgs/ae_utils/convert_to_rh_scan_format.py
```

```
# Usage: python convert_to_rh_scan_format.py -option <input_file> [-o
<output_file>]
#           option: -syn
#           -tmx
#           -scan_def
#           -pnr
#           -mgc [-front_path <front_path>] [-block_name <block_name>]
# setting within [] is optional
```

Following script is used to convert RH compatible format to RHSC compatible format:

```
<RHSC_build>/gps/pythonpkgs/thpkgs/ae_utils/convert_rh_to_rhsc_scan_format.py
```

The following function is used to get the usage details:

```
>>> from thpkgs.ae_utils import convert_rh_to_rhsc_scan_format
>>> help(convert_rh_to_rhsc_scan_format.convert_rh_to_rhsc_scan_format)
```

6.6.2. Flow Setup in RedHawk-SC

Vectorless scan can be done in RHSC by providing the relevant arguments and inputs to ScenarioView and TimingView. No other changes are required as compared to a regular vectorless dynamic run.

Required Inputs

The following are the required inputs:

- 1.** Scan chain files in JSON format
- 2.** STA or SDC file with scan clock defined
 - a.** If you require to honor the STA arrival times, it is mandatory to set the keys of the `event_time_precedence` dict to `['sta']` in ScenarioView.

Typically, scan flops are operated using a lower frequency clock. Customers are expected to provide a scan mode STA file with different clock definitions (or, alternatively, a scan mode SDC file with the necessary clock definitions). Running this flow using the normal STA file does not give any error/warning. However, depending on the pattern, there may be a massive overshoot from the expected demand current and severe di/dt effects.

In addition to the input collaterals mentioned above, the flow requires the addition of three arguments while creating scenario view. These are:

- 1.** `scan_operation`: Specifies the scan operation desired – currently, only the ‘shift’ operation is supported.
- 2.** `num_shifts`: Specifies the number of shifts to be performed. This number must be in sync with the `scenario_duration` key.
 - a.** If the `scenario_duration` is less than the time required for `num_shifts` cycles of the scan clock, then the scenario has only as many shift operations as are accommodated in the specified duration.
 - b.** If `scenario_duration` is more than the time required for `num_shifts` cycles of the scan clock, then the extra scenario duration will have only regular clock events and other vectorless activity (if any such activity is specified).

For example, if the scan clock period is 4 ns, `scenario_duration` is 20 ns, and `num_shifts` is 8, only five shift cycles are possible. However, if `num_shifts` is 3, the scan-shift operation happens only for 12 ns. If non-zero activity is set by using the `toggle_rate` key, events are initiated at flop outputs according to the activity input.

3. `scan_chain_data`: Pointers to the scan chain json files, in a list of dicts format. An example is shown in [Scan Shift Operation](#) on page 193.

The vectorless scan flow requires the use of a normal vectorless propagated scenario. The scenario types such as No-propagation vectorless scenarios cannot be used for this case.

Example run.py

The following is an example of `run.py`:

```
< .. launcher setup, db and previous view creation .. >

#list of dicts with pointers to the scan chain files
scan_chains = [{`file_name': `./rhsc_scan_1.json'},
    {'`file_name': `./rhsc_scan_2.json', `previous_pattern': 1,
     'pattern' : `01011', `start_shift' : -2 }] #override the parameters given in
json file

#create the scenario view
scn_scan = db.create_scenario_view(...,
timing_view = tv_scan, #TimingView using scan mode STA
scan_operation = "shift", #Specify scan operation
num_shifts = 4, #Number of shifts to be done
scan_chain_data = scan_chains, #list of dicts as above
...)

< .. analysis view, reporting scripts if any .. >
```

6.6.3. Scan Shift Operation

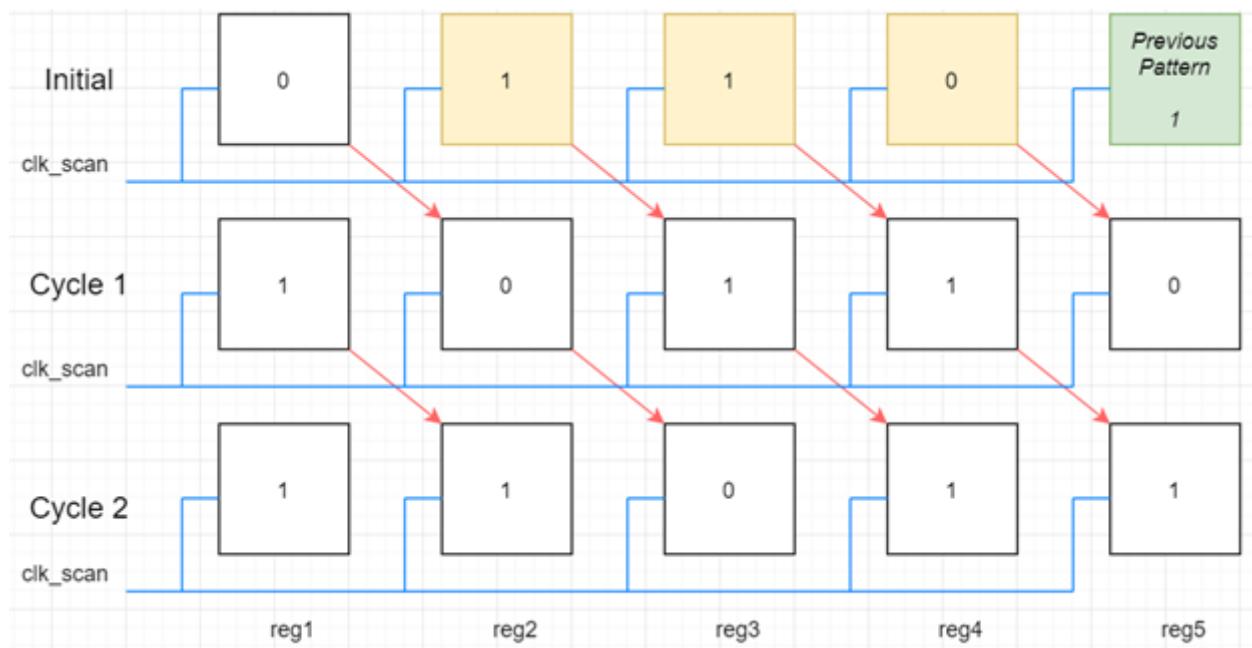
Scan shift is a highly deterministic scenario mode. This means that, for a given combination of scan chains, scan pattern, and other modifiers, the events at the output of each flop can be calculated beforehand. Consider the following case:

Num shifts = 2, Start shift = -1, Pattern = 110, Previous Pattern = 1

This chain can be specified in the scan chain file using the following lines:

```
{
  "chain_normal": {
    "instance_pins": [
      "reg1/Q",
      "reg2/Q",
      "reg3/Q",
      "reg4/Q",
      "reg5/Q"
    ],
    "pattern": "110  ",
    "previous_pattern": "1",
    "scan_clock": "clk_scan",
    "start_shift": -1
  }
}
```

In this case, the output at each flop at each cycle is as follows:

Figure 43. Sequence of Shifts in a Scan Chain

Here, Cycle N refers to the Nth clock edge. For fall edge triggered flops, the shift occurs at the falling clock edge. The process to determine the state of each flop can be roughly summarized as follows:

- 1.** Fill the last -(*start_shift*) flops with *previous_pattern*
 - a.** reg5 is filled with '1'
- 2.** Fill the next *pattern_length* flops (going towards the head of the chain) with the actual pattern
 - a.** reg4, reg3, reg2 are filled with '0', '1', '1'
 - b.** The LSB of the pattern will be shifted out first (after cycle 1)
- 3.** Continue to fill the rest of the chain with the pattern, until the first instance is reached. This gives you the initial conditions for all the flops.
 - a.** reg1 is filled with '0'
- 4.** At each clock triggering edge, shift the data from Flop_N to Flop_N+1. For Flop 0, insert the correct bit such that the pattern is maintained.
 - a.** '1' is inserted at reg1 in cycle 1, '1' is inserted at reg1 in cycle 2

The working of each of the modifiers can be understood from the above example.

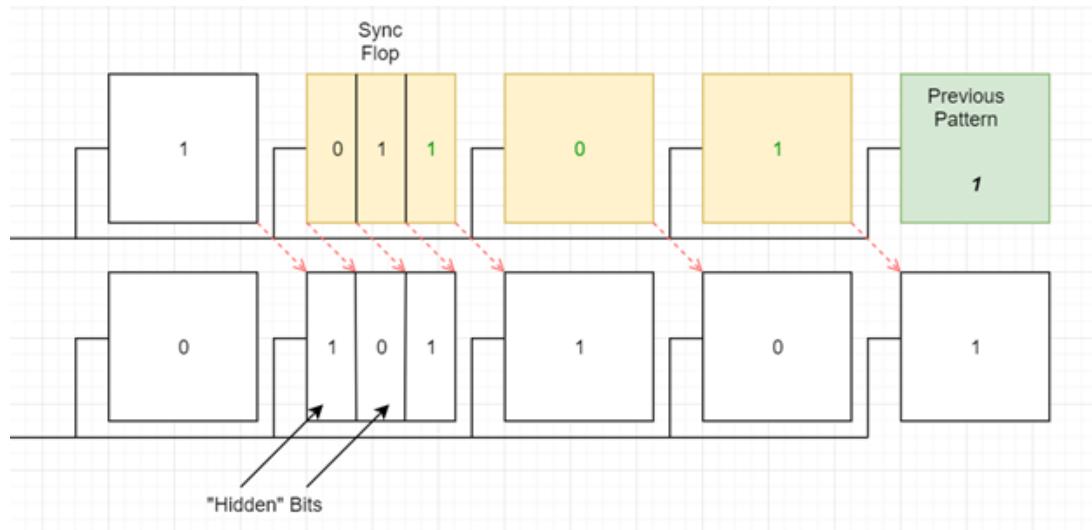
Synchronizer Flops

Synchronizer flops are cells made of multiple flops connected in series internally with one output pin. An N-bit synchronizer flop has 'N' flops connected in series. This means N clock cycles are needed to shift one bit as compared to a normal flop. In RHSC, you need to mark synchronizer flops by specifying the 'extra cycles' parameter in the scan chain file. Typically, an N-bit synchronizer flop requires N-1 extra cycles. Further, at any given moment, you can only see the state of the ending flop - therefore, synchronizer flops can have apparent hidden bits.

Synchronizer flops need to be specified using a separate dict within the scan chain file. The syntax for marking a synchronizer flop is as follows:

```
{
  "chain_synchronizer": {
    "instance_pins": [
      "reg1/Q",
      {"ip": "reg2_sync/Q", "extra_cycles": 2}, #3-bit sync flop
      "reg3/Q",
      "reg4/Q",
      "reg5/Q"
    ],
    "pattern": "101",
    "previous_pattern": "1",
    "scan_clock": "clk_scan",
    "Start_shift": -1
  }
}
```

Figure 44. Hidden Bits in a Sync Flop



Multi-bit Flops

Each flop in an MBFF is treated as an individual flop. This means it takes one cycle to shift one bit for each output. MBFF output bits need to be covered in the scan chain. The syntax for a 4-bit MBFF is as follows:

```
{
  "chain_mbff": {
    "instance_pins": [
      "reg_mb1/Q_0_",
      "reg_norm1/Q_",
      "reg_norm2/Q_",
      "reg_mb2/Q_3_",
      "reg_mb3/Q_4_"
    ],
    "pattern": "101",
    "previous_pattern": "1",
    "extra_cycles": 2
  }
}
```

```

        "scan_clock": "clk_scan",
        "start_shift": -1
    }
}

```

6.7. Controlling Macro Switching in RedHawk-SC

Macros, which include memories, hard IPs, third party IP's, other analog parts and so on, form an important part of SoC chips. They occupy much space and draws high power/current. Most of the power consumed by macro comes out as internal power and therefore, depends on the logic state of input pins of the macro. So, when a dynamic ScenarioView is created, the logic state of different pins governs the power/current consumed by macros at each clock cycle. This becomes difficult to control even if a gate FSDB is provided as event input. RedHawk-SC allows fine grained control over the modes considered for macros so that user can come up with desired modes/sequences needed for analysis.

Following are the different methods to derive current signature for a macro/memory:

- **Sim2iprof:** This utility can generate the current signature for a macro using the ANSYS Power Library (APL) format. You can capture current signatures for various modes of a macro in the APL format. Each current signature can be associated with a specific mode name and a Boolean equation of input pin logic state. Currents are present for the whole clock cycle, together for both active and inactive edges.
- **AVM:** Library vendors provide the specifications for memory power consumption in ANSYS Virtual Memory (AVM) format. AVM can be input as such to RedHawk-SC. Further, AVM can also be converted to APL and then supplied. Also, each current signature can be associated with a specific mode name and a Boolean equation of input pin logic state. AVM is also considered equivalent to an APL input and the currents are for the entire clock cycle as in Sim2iprof.
- **Liberty:** In the absence of APL generated by Sim2iprof or AVM, RedHawk-SC can automatically derive current signatures from liberty input using energy tables. This can be composite current source (CCS) power format or nonlinear power model (NLPM) format. Here also, each current signature can be associated with a specific mode name and a Boolean equation of input pin logic state. For Liberty-based inputs, currents are present separately for active and inactive edges.

There are different ways to create a dynamic scenario for doing voltage drop analysis, such as VCD and vectorless methods. In both logic propagation and NPV dynamic scenarios, mode control syntax is available in the `object_settings` dict under the `settings` argument. Macro control settings are input using '`mode_control`' key of `object_settings`. The syntax and examples are explained in the following sections.

6.7.1. Input Settings

The key '`mode_control`' is the key or parameters used for controlling modes of macros including those that are part of the clock network.

Macro mode control input settings includes the following topics:

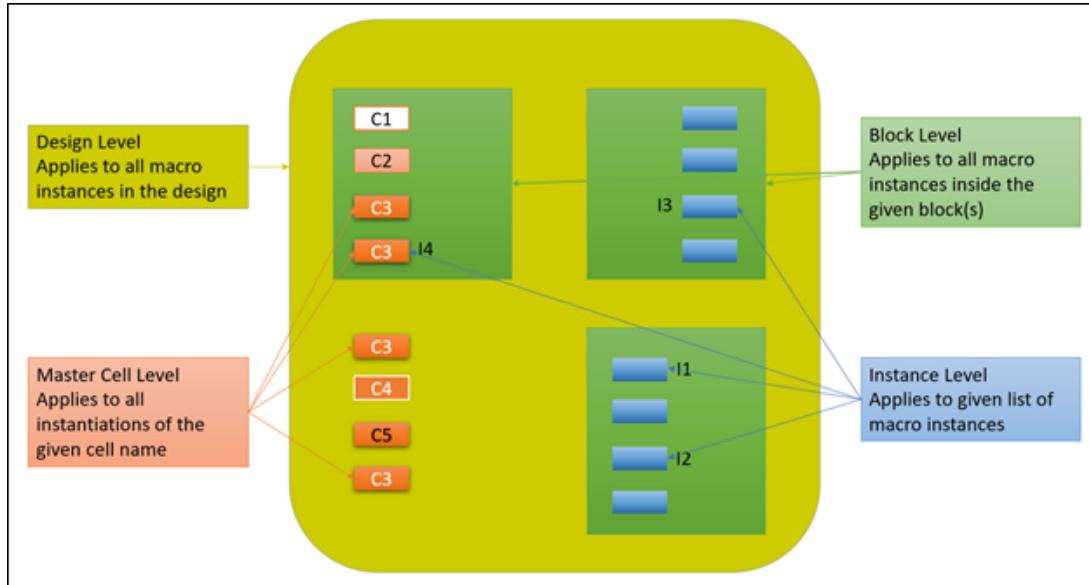
- [Scope or Levels of Control](#) on page 197
- [Mode Control](#) on page 197
- [Mode Sequence](#) on page 197
- [Mode Probabilities](#) on page 198
- [Implicit Modes](#) on page 199
- [Custom Modes](#) on page 200
- [Custom Clock Period and Offset](#) on page 201
- [Using mode_change period](#) on page 201

- [Controlling Active Pins of Multi-Clock Macro Instances](#) on page 202

Scope or Levels of Control

The following figure shows different scopes available for macro control.

Figure 45. Scopes Available for Macro Control



The first level keys taken into `object_settings` are:

- `design_values`
- `block_values`
- `cell_values`
- `leaf_instance_values`

These correspond to the different scope, with high priority for the most specific input. The priority order is `leaf_instances_values > cell_values > block_values > design_values`.

The `design_values` takes in a dict and the rest three inputs as a list of dict as there can be multiple blocks, master cells, or leaf instances to be controlled. The following sections describe the modes in detail with examples.

Mode Control

`mode_control` is the key in `object_setting` that is used to control switching mode of macros. Mainly, there are two modes of input, `mode_sequence` and `mode_probabilities`. One of these keys is mandatory.

Mode Sequence

A sequence of mode names (a python list), that is applied to each macro instance. By default, the sequence will start repeating after applying all the modes in the sequence and there are clock cycles left for the macro. If `mode_sequence_repeats` is set to False, the last mode in the sequence is applied for rest of the clock cycles. The following are examples for this scenario:

Example1

```
object_settings_macro={
    'cell_values' : [
        {
            'patterns' : 'MEM_CELL_NAME',
            'mode_control' : {
                'mode_sequence' : [
                    'ACTIVE_write',
                    'ACTIVE_read',
                    'standby_ntrig',
                    'standby_trig',
                    'SLEEP'] }}, ]}
```

If the macro is having a frequency of 500MHz for the input clock, you would have each clock cycle of 2ns. A mode gets applied for each of the clock cycles. If the input scenario duration is more than 10ns (if the example is having list of 5 modes), the mode sequence cycle gets repeated.

Example2

```
object_settings_macro={
    'cell_values' : [
        {
            'patterns' : 'MEM_CELL_NAME',
            'mode_control' : {
                'mode_sequence' : [
                    'MEM_write',
                    'MEM_read', ] }}, ],
    'leaf_instance_values':[
        {'instances': [Instance('mem1'), Instance('mem2')], 'mode_control' : {
            'mode_sequence' : [
                'MEM_read',
                'MEM_write',
                'standby'], },
            'mode_sequence_repeats' : False}, ], }
```

Since `mode_sequence_repeats` is set to false, the sequence that is applied for the second set in the example is `MEM_read, MEM_write, standby, standby`.

For **Example1**, it would be `MEM_write, MEM_read, MEM_write, MEM_read`.

Example3

```
object_settings={
    'block_values' : [
    {'patterns' : 'block_A0_inst*', 'mode_control' : {
        'mode_sequence' : [
            'ACTIVE_write',
            'ACTIVE_read',
            'standby_ntrig',
            'standby_trig',
            'SLEEP'] }, ]}
```

Mode sequence is mainly used for cell/leaf instance level inputs where the exact macro instance and the exact mode sequence is known.

Mode Probabilities

Here, rather than exact sequence, probability of occurrence of each mode is input. Mode probabilities style expects a python dict with keys as mode names, and values as their probability of applying in the events for

the scenario duration. Probability values for the modes must sum up to 1.0 for each scope. If it doesn't add up to 1, then `_leakage_only_mode_` is applied (this mode will be discussed in upcoming section). If probability value exceeds 1 then scaling will happen internally to adjust it to 1.0. Following are the examples for the scenario:

Example1

```
object_settings_macro={  
    'cell_values' : [  
        {'patterns' : 'MEM3X*',  
         'mode_control' : { 'mode_probabilities' : {  
             'MEM_write' : 0.5, 'standby' : 0.5} },},  
        {'patterns' : 'MEM2X*',  
         'mode_control' : { 'mode_probabilities' : {  
             'MEM_write' : 0.75, 'standby' : 0.25} },},],  
    'block_values': [  
        {'patterns' : 'core2',  
         'mode_control' : { 'mode_probabilities' : {  
             'MEM_read' : 0.4, 'MEM_write' : 0.3, 'standby' : 0.25} },},],
```

The probabilities are applied both to the instances belonging to given scope and also to each clock cycle of the instance, for example consider 'READ' mode with 0.1 as probability. If input to only one instance with 20 cycles, you can expect 0.1% of cycles, for example two cycles to have 'READ' mode. If input to 20 macros with only one clock cycle of scenario duration, you can expect two macros to have 'READ' mode.

Example2

```
object_settings={  
    'leaf_instance_values':[  
        {'instances' : 'inst_name',  
         'mode_control' : {  
             'mode_probabilities' : {  
                 'ACTIVE_write':0.6,  
                 'ACTIVE_read' :0.4}}}]}
```

Mode probabilities is typically used for design/block level inputs where the exact macro instance and exact mode sequence is not known but a control on overall mode distribution is desired.

Implicit Modes

The previous section dealt with examples where the mode names in input files are known. When the current source is APL, the modes have names that you can use. When input is Liberty (NLPM or CCS), it might be difficult to interpret the modes. During DesignView generation, the tool identifies the following pre-defined modes for each cell based on the energy consumption. You can use these implicit energy modes to annotate `mode_sequence` or `mode_probabilities` in macro `mode_control`.

Energy Mode	Description
<code>_low_energy_mode_</code>	Lowest Energy Mode (in Liberty file)
<code>_median_energy_mode_</code>	Median Energy Mode
<code>_high_energy_mode_</code>	Highest Energy Mode
<code>_leakage_only_mode_</code>	Only Leakage is Consumed
<code>_off_mode_</code>	Off Mode, that is, not even Leakage

The following example shows how to use the implicit modes.

```
object_settings_macro = {
    'design_values' : {
        'mode_control' : { 'mode_probabilities' : {
            '_high_energy_mode' : 0.2,
            '_median_energy_mode' : 0.6,
            '_leakage_only_mode' : 0.2}, }, },
    'leaf_instances_values' : [
        {'instances' : [Instance('ip345')], 'mode_control' : { 'mode_sequence' : [
            'MEM_write', '_low_energy_mode', '_median_energy_mode']}, },
        {'instances' : [Instance('mem3'), Instance('mem4'), Instance('mem5')], 'mode_control' : { 'mode_probabilities' : {
            'MEM_read' : 0.4,
            '_low_energy_mode' : 0.3,
            '_leakage_only_mode' : 0.3}}, },
        {'instances' : [Instance('macro_XYZ')], 'mode_control' : { 'mode_sequence' : ['_off_mode']}}, ], }
```

You can also use implicit modes when APL is present, for example, when several APL modes exist in the input current file and the highest energy mode is considered for `mode_control` annotation.

Custom Modes

You can define a mode name and associate it with a `when` Boolean condition defined in the Liberty file. This is useful when current sources are CCS power or NLPM and the current for a specific Boolean condition is applied for analysis. For example, `new_ram_mode_def` is a list of custom-defined modes.

```
new_ram_mode_def = {
    'my_WRITE' : { 'when' : '!we&cs&!byp&!se'},
    'my_READ' : { 'when' : 'we&cs&!byp&!se'},
    'my_STANDBY' : { 'when' : '!cs'}, }
my_modes_def = [{ 'name' : 'my_ram_modes', 'modes' : new_ram_mode_def}]
```

You can now reference these modes in `mode_control` to annotate `mode_sequence` or `mode_probabilities`.

You must specify mode definition information in both `mode_control` of `object_settings` and with the `mode_definitions` argument of the `ScenarioView` creation command.

```
settings_macro = {
    object_settings = {
        'cell_values' : [{ 'patterns' : 'ram*', 'mode_control' : { 'mode_sequence' : [
            'my_READ', 'my_WRITE', 'my_READ', 'my_STANDBY'],
            'mode_definition' : 'my_ram_modes'} }], }
    mode_definitions = my_modes_def
}

npv_args = dict(..., settings = settings_macro)
scn_args = dict(..., settings = settings_macro)
```

Different cells might have different `when` conditions to specify the same mode, such as READ). In such cases, multiple mode definitions can be defined and applied to different cell patterns.

Custom Clock Period and Offset

The previous settings dealt with establishing which all modes need to be applied to the macros and in what order or priority. There is also the question of when to apply the modes, that is what time the modes should come in. In general, clock pins are the active pins for modes and the modes/currents get applied when these clocks(active pins) switches. Clock switching happens according to the settings for the run (different ways of FSDB, Vectorless, sta event time, propagated event time and so on). If you want to control switching, `mode_change` can be used to specify the exact time of switching. This is a dict, and can have the following keys:

Table 10: mode_change Keys

Key	Description
period	This is a float value that inputs clock period for the macro. So, if period is 3ns, the modes happen at each 3ns cycle.
offsets	This is a list tuple with tuple entries being : float, string with clock name and 'r' or 'f'. This indicates the offsets from periods edge on which the clock is to switch.
use_sta	This is a Boolean value. This is not important for NPV ScenarioView where clock period information is input from the STA file. For logic-propagation ScenarioView, if the propagated clock and the clock from the STA file are different, this key is useful.

If these values are given, the new mode is applied at ($i * \text{period} + \text{offset}$) where i starts from 0 until the time reaches scenario duration. This is useful when there is no STA coverage for the instance or when you want the current for the mode to be applied at a particular time in the cycle. For the following settings, modes are applied at 3e-9, 19e-9 and so on if the cell is rise-edge triggering cell.

```
object_settings={
    'leaf_instances_values' : [
        {'instances' : [Instance('ip345')],
         'mode_control' : { 'mode_sequence' : [
             '_MEM_write', '_low_energy_mode_', '_median_energy_mode_']},
         'mode_change' : { 'use_sta' : True }, },
        {'instances' : [Instance('mem3'), Instance('mem4')], 
         'mode_control' : { 'mode_sequence' : [
             '_MEM_write', '_leakage_only_mode_']},
         'mode_change' : {
             'period' : 16e-09,
             'offsets' : [(3e-09, 'CLK', 'r'), (2e-09, 'CLK', 'f')]}}
```

In the example, clock rise occurs at 3ns and clock fall at 10ns (half cycle of 8ns and given value 2ns). For nonlinear power model (NLPM) or composite current source (CCS) power source, mode is applied according to the active edge. The other edge has clock-only current as per the input Liberty file. For APL current source, mode is applied only at the active edge.

Using mode_change period

In the `mode_control` specification (for any scope) of the `object_settings` dict, you can specify the `mode_change` key with `period` with or without the `mode_sequence` or `mode_probabilities` keys.

In the following example, `mode_change` is specified without `mode_sequence` or `mode_probabilities`. This enables you to input the clock period for a specific scope (such as a design, block, cell, or instance) in

multi-clock macro scenarios. The tool determines a reaching clock period that is closest to the specified period and uses the clock pin to create events.

```
object_settings = {'leaf_instance_values': Instance('u1'),
'mode_control' : {'mode_change' {'period' : 4e-9}}}
```

When you use mode_sequence or mode_probabilities, the tool creates events on all the clock pins and applies currents based on mode names. For example,

```
object_settings = {'leaf_instance_values': Instance('u1'),
'mode_control' : {'mode_sequence': ['READ', 'WRITE'],
'mode_change' {'period' : 4e-9} } }
```

Controlling Active Pins of Multi-Clock Macro Instances

To select the clock pin for application of modes to a macro cell or instance with multiple clock pins, use the macro_multi_clock_behavior key with the calculation key under the settings argument as shown in the following example:

```
settings = {'calculation' : {
    'macro_multi_clock_behavior' : 'fastest'}
scn = db.create_scenario_view(..., settings = settings, ...)
```

The macro_multi_clock_behavior key has the following values:

- fastest : One clock pin from all the clock pins is chosen, and input to output events are created for that clock pin. The fastest among non-quiet clock pins is picked.
- first : First pin in the related clock pin list for each output pin is picked to create input to output events.
- all : Events are created on all clock and non-clock input pins with associated current tables.
- default : Default flow applies modes to the fastest clock. In the PCVS flow, the first clock pin is considered so that consistent clocks are considered for the initial estimation and the final event creation stages.

Note: For mode_control based current application, the clock pins that are related to the specified modes are considered. This setting applies only to logic propagation ScenarioView. No propagation ScenarioView always takes fastest clock or considers clocks based on mode_control input.

6.7.2. Reviewing Results

In this section, you need to discuss the ways to understand the modes that got used for the macro. The input could have been in any combination explained in the previous section. There are APIs which help in understanding the various modes present for a macro instances as well as the mode that has been implemented in the scenario that is generated.

<DesignView>.get_cell_mode_attributes

This is a DesignView API. It returns the details of implicit states. The Boolean combination that corresponds to the low, median and high-power modes are there here. The information is returned for each liberty corner

included in the DesignView. Even if APUs are included, the highest/lowest among them is also shown. Following are the examples:

```
>>> dv.get_cell_mode_attributes(Cell('ABC'))
{'corner1': {'apl': {(0.8299999833106995, 125.0): {
        '_high_energy_mode_': ('ACTIVE_write', 'we * cs'),
        '_low_energy_mode_': ('standby_ntrig', '!cs'),
        '_median_energy_mode_': ('ACTIVE read', '!we * cs')}},
    'ccsp': {(0.8299999833106995, 125.0): {}},
    'nlpm': {(0.8299999833106995, 125.0): {
        '_high_energy_mode_': ('', 'we * cs * !byp * !se'),
        '_low_energy_mode_': ('', '!cs * !byp * !se'),
        '_median_energy_mode_': ('', 'byp * !se * !cs')}},
    'corner2': {'apl': {(0.8299999833106995, 125.0): {}},
        'ccsp': {(0.8299999833106995, 125.0): {}},
        'nlpm': {(0.8299999833106995, 125.0): {
            '_high_energy_mode_': ('', 'we * cs * !byp * !se'),
            '_low_energy_mode_': ('', '!cs * !byp * !se'),
            '_median_energy_mode_': ('', 'byp * !se * !cs')}}}}}
```

<ScenarioView>.get_mode_usage_attributes

The API is common for both regular dynamic ScenarioView and no-propagated vectorless (NPV) ScenarioView. For the power pin and active pin (mostly clock pin), the API returns the mode applied and the time associated with each of them. That way, one can easily understand about the mode, whether the input was sequence/probability or implicit/explicit.

Modes considered for both rise and fall of the active pin are returned by the API. For Liberty-input based flows, the other edge consumes only leakage power unless it is dual-edge triggered. There are two mode entries per period.

Note: An APU input has current for the entire clock cycle. It is only applied once during the active or the triggering edge. For APU input, the `get_mode_usage_attributes` output reports modes only at active edges, that is, once in a clock cycle. For Liberty-based inputs, separate currents applied at active and non-active edges have two entries for each clock cycle.

Example1

When a mode sequence is provided and `mode_sequence_repeat` is not made `False`, the sequence is repeated.

```
object_settings_mem_inst_transient2 = {
    'cell_values': [
        {'cell_pattern': 'cellB',
        'mode_definition': 'ram_cellB',
        'mode_sequence': ['OFF_STATE', 'ON_STATE'],
        'mode_change': {'use_sta': True}}]}

>>> scn.get_mode_usage_attributes(Instance('inst_of_cellB'))
{Pin('clk'): {Pin('VDD'): [(2.649999664150471e-10, 'OFF_STATE'),
                            (6.349999970289844e-10, 'OFF_STATE'),
                            (1.0059999500100503e-09, 'ON_STATE'),
                            (1.3759999806239875e-09, 'ON_STATE'),
                            (1.7469998780939022e-09, 'OFF_STATE'),
                            (2.1169999087078395e-09, 'OFF_STATE'),
                            (2.488000028222359e-09, 'ON_STATE'),
                            (2.8580000588362964e-09, 'ON_STATE')]}}
```

Example2

Custom offset and period values can be given. This is much useful when there is no STA file coverage for the macro instance.

Here, period is 700ps and offsets are 200ps and 300ps. You have modes at 200ps, 450ps, 900ps ($700*1 + 200$), 1150ps ($700*1 + 350+100$), and so on.

```
object_settings_mem_inst_transient3 = {
    'cell_values' : [
        {'patterns' : 'cellB',
         'mode_probabilities' : {
             '_high_energy_mode_': 0.5,
             '_low_energy_mode_': 0.25,
             '_median_energy_mode_': 0.25},
            'mode_change' : {'period':700e-12, 'offsets' : [(200e-12, 'clk', 'r'), (100e-12, 'clk', 'f')]}]}
]

>>> scn3.get_mode_usage_attributes(Instance('inst_of_cellB'))
{Pin('clk'): {Pin('VDD'): [(2.000000267e-10, 'we * cs * !byp * !se'),
                           (4.499999706e-10, 'we * cs * !byp * !se'),
                           (9.0000001895e-10, '!cs * !byp * !se'),
                           (1.1499997171e-10, 'byp * !se * !cs'),
                           (1.60000002131e-09, 'we * cs * !byp * !se'),
                           (1.85000002962e-09, 'we * cs * !byp * !se'),
                           (2.299999968284e-09, '!cs * !byp * !se'),
                           (2.54999997655e-09, 'we * cs * !byp * !se'),
                           (3.000000026176508e-09, 'byp * !se * !cs')]}}
```

Example3

Leakage-only mode is applied if none of the valid states match with APL. In this example, the invalid `_high_energy_modejj` state is specified so that only the leakage mode is applied. Other implicit modes are honored.

```
object_settings = {
    'leaf_instance_values': [
        {'instances' : Instance('e/f/g'),
         'mode_probabilities':{
             '_high_energy_modejj':0.3,
             '_low_energy_mode_':0.3,
             '_median_energy_mode_':0.4},
            'mode_change' : { 'use_sta' : True }},]}
}

>>scn.get_mode_usage_attributes(Instance('inst_with_pattern_e/f/g'))
{Pin('CK'): {Pin('CVCC'): [(3.66e-10, '_leakage_only_mode_'),
                           (1.275e-09, '_low_energy_mode_'),
                           (2.184e-09, '_median_energy_mode_'),
                           Pin('VCC'): [(3.66e-10, '_leakage_only_mode_'),
                           (1.275e-09, '_low_energy_mode_'),
                           (2.184e-09, '_median_energy_mode_')]}}}
```

<ScenarioView>.get_demand_current

The API is common for both regular dynamic scenario and no propagated vectorless scenario. Given instance and power/ground pin, the API returns the current waveform. This is as `Waveform` object which can be viewed with the function `plot`. Following are the examples for mode control usages and the corresponding current waveforms:

Example1

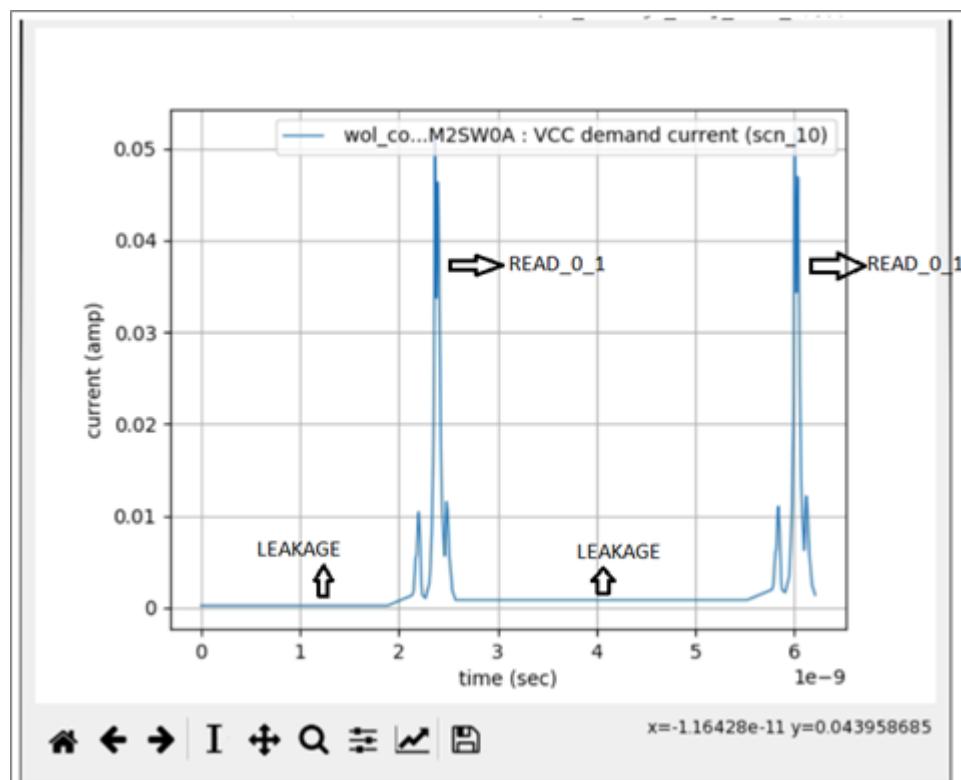
When mode probability entries do not add up to 1.0, leakage-only mode is used to obtain a probability of 1.0. In the following example, the `_high_energy_mode` value is 0.3. So, `_leakage_only_mode_value` is specified as 0.7 to cover the remaining probability.

```
object_settings = {
  'leaf_instance_values': [
    {'instances': Instance('a/b/c'),
     'mode_probabilities': {
       '_high_energy_mode': 0.3,
       '_mode_change': { 'use_sta' : True } }, ] }

i=Instance('a/b/c')
>>plot_waveforms(scn.get_demand_current(i,Pin('VCC')))
```

The following figure shows scenario demand current with implicit mode and leakage:

Figure 46. Scenario Demand Current with Implicit Mode and Leakage



In this example, leakage is present for 70% and the high energy mode of `READ_1_0` happens for 30% of total number of clock cycles. `READ_1_0` is seen with current of 0.05A in this case and leakage is mostly a DC.

Example2

When implicit modes are used with 0.5 `mode_probability` for high and low energy modes. In the following example, `mode_probability` for `_high_energy_mode` and `_low_energy_mode` is given 0.5 probability each.

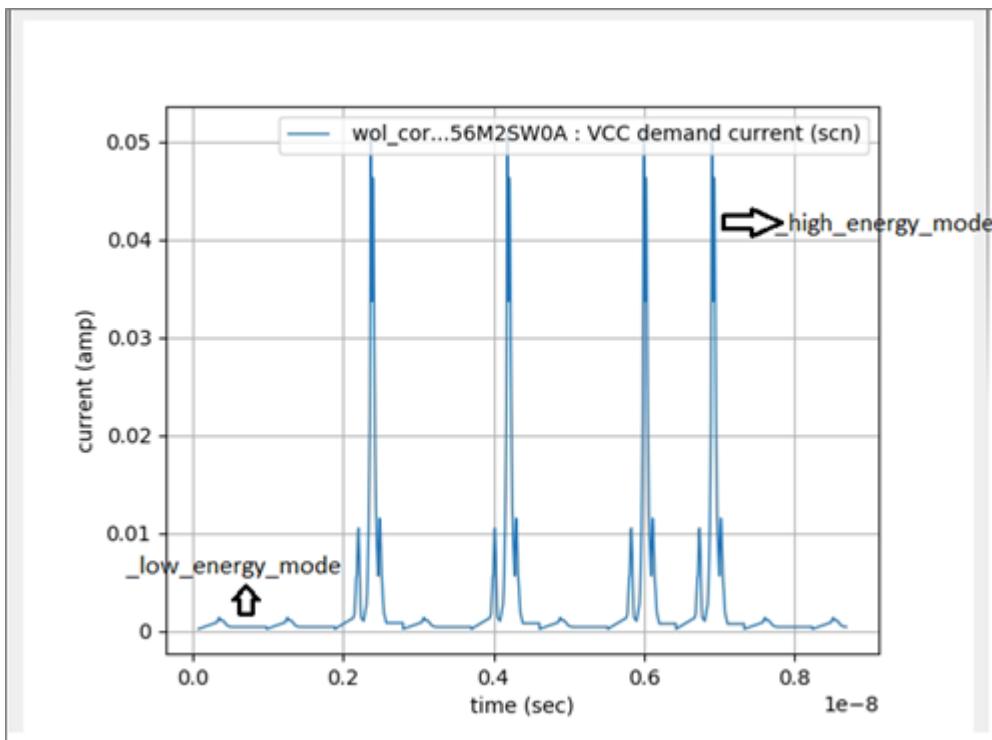
```
object_settings = {
  'leaf_instance_values': [
```

```
{
  'instances': Instance('x/y/z'),
  'mode_probabilities':{
    '_high_energy_mode':0.5,
    '_low_energy_mode':0.5},
  'mode_change': { 'use_sta' : True }},
]
```

```
i=Instance('x/y/z')
>>plot_waveforms(scn.get_demand_current(i,Pin('CVCC')))
```

The following figure shows scenario demand current with high and low implicit modes:

Figure 47. Scenario Demand Current with High and Low Implicit Modes



Example3

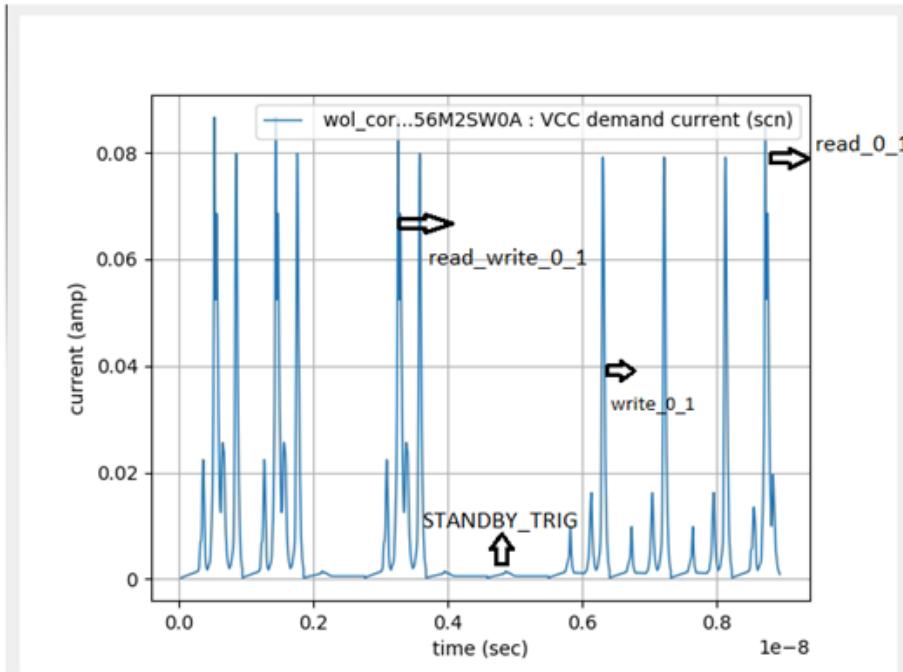
When explicit modes are used with 0.3 mode_probability each for read_write_0_1, STANDBY_TRIG, and write_0_1 modes, and 0.1 mode_probability for read_0_1 mode.

```
object_settings = {
  'leaf_instance_values': [
    {'instances': Instance('my_ram/inst1'),
     'mode_probabilities':{
       'read_write_0_1':0.3,
       'STANDBY_TRIG':0.3,
       'write_0_1':0.3,
       'read_0_1':0.1}}]
```

```
'read_0_1':0.1},
'mode_change' : { 'use_sta' : True },]]}

i=Instance('my_ram/inst1')

>>plot_waveforms(scn.get_demand_current(i,Pin('VDD')))
```

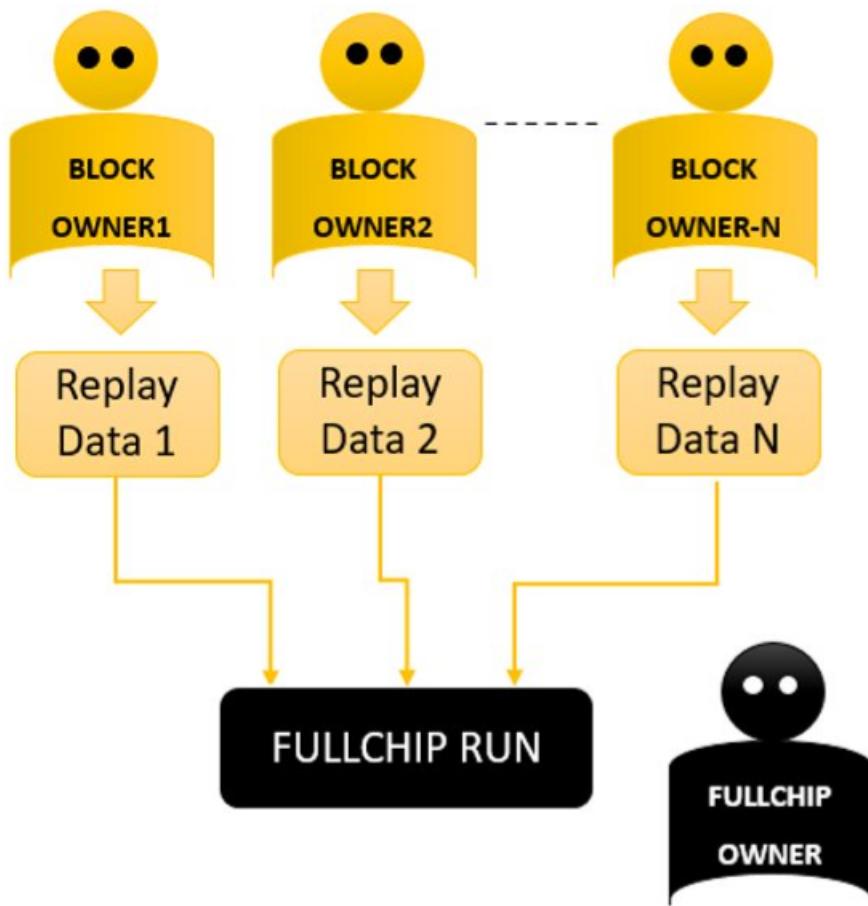
Figure 48. Scenario Demand Current with APL State Names

6.8. Event Replay Flow

The following are the concepts behind the event replay flow:

- Reuse events from one ScenarioView to another.
- Integrating/replicating block level setup/events to a top-level run.
- Developing scenarios by the block owner that are representative/good for DVD, that is you need to use these exactly when the block is instantiated at higher level.
- Used across ScenarioView types (Logic Propagation/NPV/PCVS and so on).

The following figure shows the event replay flow:

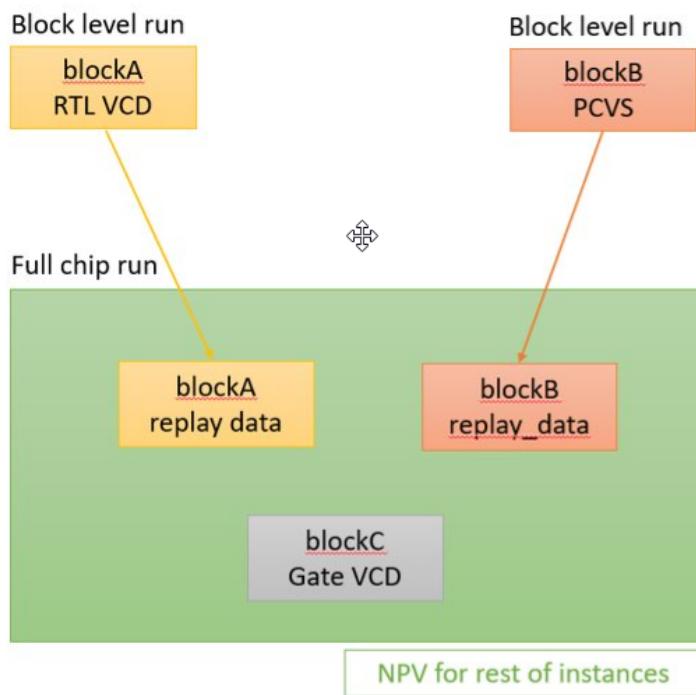
Figure 49. Event Replay Flow

6.8.1. Methodology

The following are the methodologies involved in the event replay flow:

- Outputs event replay data (a directory) from one ScenarioView.
- Reads it into the next ScenarioView.
- The following are the possible combinations:
 - block -> top, top -> block, top -> top, block -> block,
 - No Prop-SCN->Logic Prop-SCN, PCVS->NPV, VCD->NPV

The following figure shows the methodology for the event replay flow:

Figure 50. Methodology

6.8.2. Examples

Writing Event Replay Data

The following example shows how API writes out event replay data. This is used with any type of ScenarioView:

```
scn_npv_30tr.write_event_replay_files(
    file_dir_location = 'npv_40ns_30%act_fullchip_replay')

scn_vcd_rtl.write_event_replay_files(
    file_dir_location = 'blockC_replay_data_rtl_vcd_modem_63ns',)
```

The default value is top level. The input directory location for the whole data is written in the following example.

```
scn_npv_300mW.write_event_replay_files(
    file_dir_location = 'npv_36ns_300mW_core3_replay',
    instance = Instance('core3'))

scn_pcvs_400mW.write_event_replay_files(
    file_dir_location = 'block1_replay_data_pcvs_400mW_89ns',
    instance = Instance('hier3/block1'))
```

The above example is used for writing out replay for block level.

Note: It is a good practice to have file name with all details.

Reading Event Replay Data

The argument `event_replay_data` takes in the directory that is written out.

```
event_replay_core3 = [
    {'file_dir': 'core3_replay_data_pcvs_400mW_40ns',
     'instances': ['core3']}
]
npv_args = dict(
    scenario_duration=40e-09,
    settings = {
        'pvt' : {voltage_levels=voltage_levels},
        'frame_length' : 8e-09,
        'object_settings' : object_settings_npv,
        'event' : {default_clock = {'policy': 'custom', 'period': 8e-9}}
    },
    event_replay_data = event_replay_core3,
    tag='npv',
    options=options)
scn_npv = db.create_scenario_view(timing_view=tv, external_parasitics=evx,
extract_view = ev, **npv_args)
```

NPV takes in event replay for a block from PCVS. Event replay data takes higher priority over other settings that is, VCD, toggle rate, target power and so on.

Example for Logic propagation SCN taking in event replay for two blocks from NPV. Event replay data takes higher priority over other settings, that is VCD, toggle rate, target power and so on.

Specifying Event Replay Time Shift

To input a time shift with event replay data to ScenarioView, use the `time_delay` key. When you specify this key, all events are shifted by this time shift value. This is useful in situations where the event replay data is reused for multiple instantiations of a block, and the instantiations have some phase shift implemented to avoid high peak current.

The following example shows how to specify the `time_delay` key:

```
blockU_events = [
    {'file_dir': 'block_replay_data_U', 'instances' : ['U_block1'],
     'time_delay' : 2e-11},
    {'file_dir': 'block_replay_data_U', 'instances' : ['U_block0']}]

scn = db.create_scenario_view(..., event_replay_data = blockU_events)
```

6.9. Performing Dynamic Voltage Drop Analysis

The dynamic AnalysisView takes a SimulationView and a dynamic ScenarioView as inputs and stores the results of simulation. All voltage and current queries are available after simulation from the AnalysisView. You can query these by using the `emir_reports` commands listed in [Reporting Dynamic Analysis Results](#) on page 215.

The dynamic current that an instance sinks from the PG network while switching is called its demand current. The tool stores piece-wise linear demand current values in ScenarioView. The AnalysisView samples the

demand current values from ScenarioView by the AnalysisView time step (`step_size`) and stores these as **used demand current**. You can query both the demand current and used demand current.

The following example shows how to create AnalysisViews for two different dynamic vectorless scenarios.

```
av_dynamic_pcvs_args = dict(
    duration=40e-09,
    step_size=20e-12,
    tag='av_dynamic_pcvs',
    options=options)

av_dynamic_npv_args = dict(
    duration=24e-09,
    step_size=20e-12,
    tag='av_dynamic_npv',
    options=options)
```

The `step_size` argument specifies the sampling time duration of simulation. You select the `step_size` based on frequency and delay values of the design. Reducing the step size improves the accuracy of simulation while increasing the run time. By default, the tool automatically determines the step size.

The following sections describe the detail level of statistics and the DVD parameters available in AnalysisView.

- [Specifying Level of Statistics](#) on page 211
- [DVD Reporting Parameters](#) on page 211

6.9.1. Specifying Level of Statistics

The `keep_stats_level` argument specifies the detail level of simulation statistics and heatmap creation information stored in the AnalysisView. Default is `Medium`, that is, AnalysisView stores a single data set of DVD parameters (such as `minTW` and `effTW`) for the entire analysis duration.

- To store DVD parameters for each clock cycle, set `keep_stats_level` to `High`.
- To store instance voltage and current waveforms, set `keep_stats_level` to `Full`. For example,

```
keep_stats_level='Full'
```

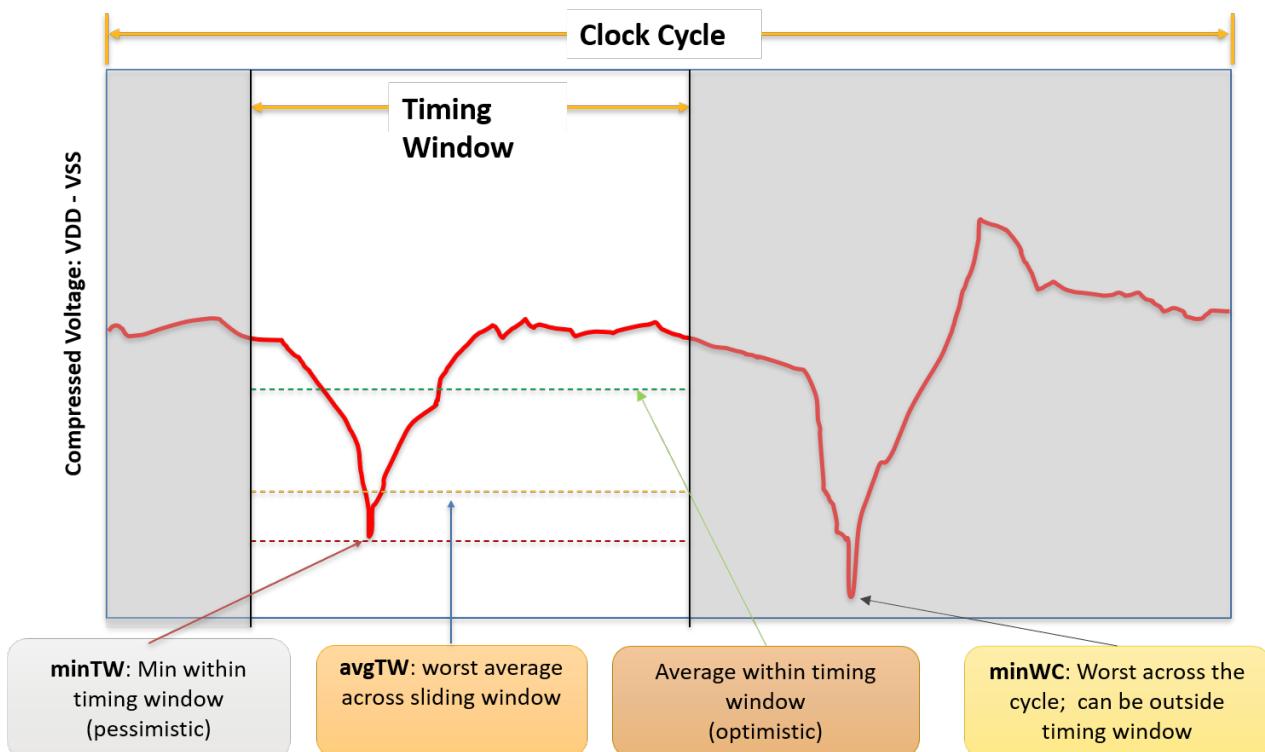
6.9.2. DVD Reporting Parameters

AnalysisView stores several parameters that measure the extent of dynamic voltage drop across design instances. The tool reports these dynamic voltage drop parameters for your analysis. These include parameters based on the timing window and the effective window. The following sections describe these parameters.

- [Timing Window Based Parameters](#) on page 211
- [Effective DVD Window Based Parameters](#) on page 213

6.9.2.1. Timing Window Based Parameters

The following figure shows a typical dynamic voltage drop waveform across an instance for one clock cycle. The instance switches for the duration of the timing window.



- **minWC**

Minimum or worst dynamic voltage across the clock cycle. This can lie outside the timing window and can occur because of neighboring instance switching. Therefore, this parameter is less used.

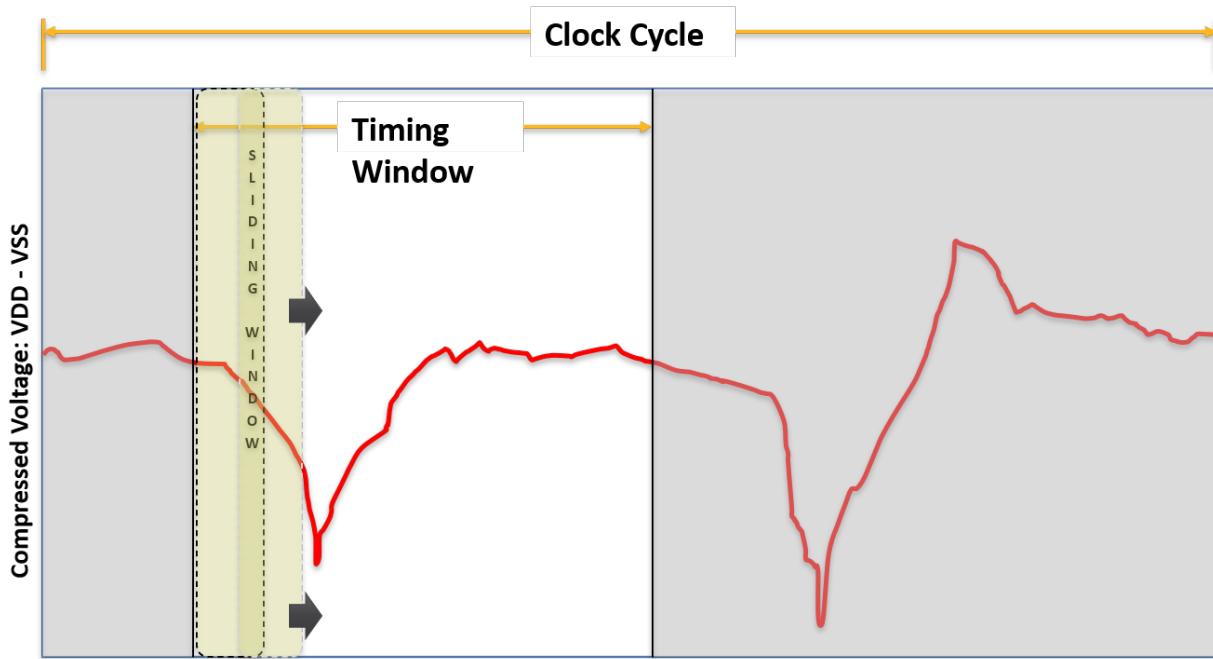
- **minTW**

Minimum voltage in the timing window. Using this parameter can be pessimistic because such a large voltage drop might rarely occur.

- Average voltage in the timing window. Using this parameter can be optimistic because such a low voltage drop might be an incorrect representation of the actual DVD, specifically if the timing window is long.

- **avgTW**

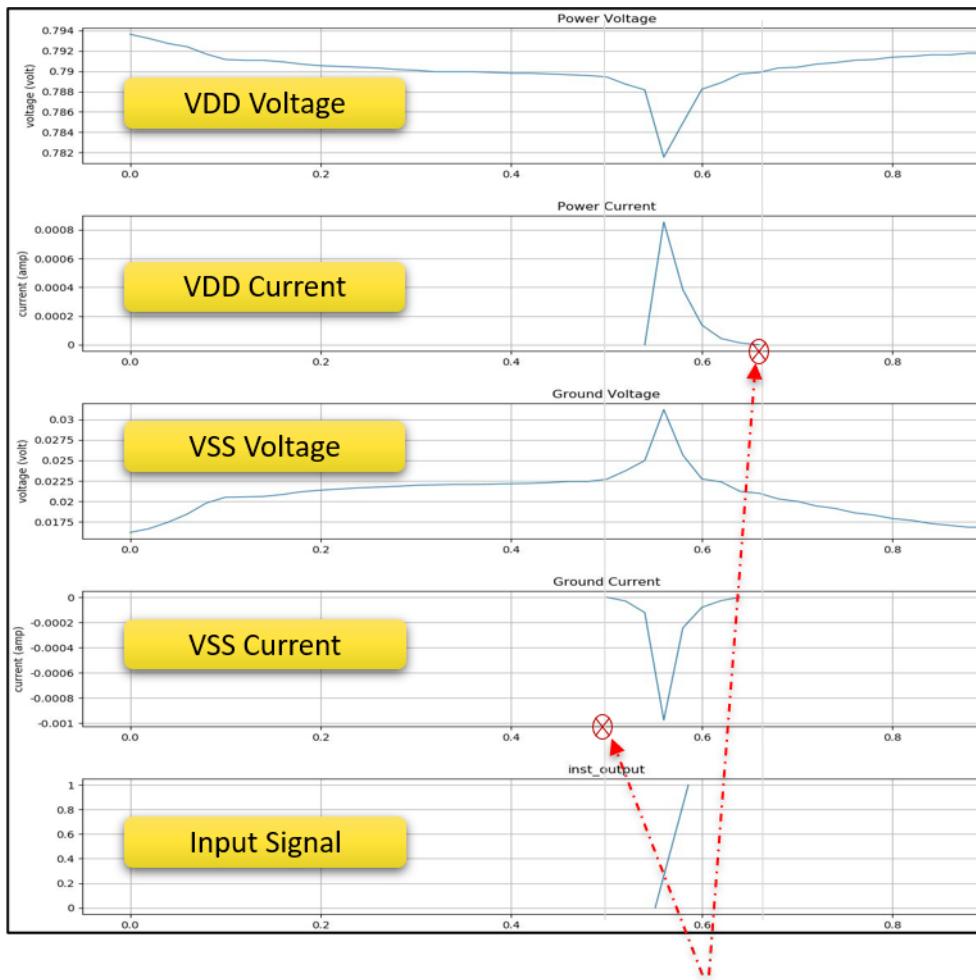
Worst average voltage across the sliding window. The sliding window represents the switching duration when the instance is sensitive to voltage drop. It is the sum of cell delay and the average transition time. The sliding window is moved across the total timing window using the time step (`step_size`) from AnalysisView and the average voltage drop in each sliding window is measured. AvgTW is the worst voltage across all the sliding windows. The minTW value can be pessimistic while the average voltage in the timing window can be optimistic if the timing window is long. avgTW is the preferred representation of the instance voltage drop.



By default, the tool calculates the average voltage drop across the entire timing window. To enable the use of sliding window for average voltage drop computation, set the `timing_window_stats_mode` key to `sliding_window` with the `keep_stats_level` key under the `settings` dict of the `create_analysis_view` command. The default is `full_tw`.

6.9.2.2. Effective DVD Window Based Parameters

If there is a mismatch between VCD data and STA data, DVD parameters reported using timing window measurements might not correctly represent the actual voltage drop values. For such cases, the tool stores another set of DVD parameters based on the effective DVD window. Effective DVD window is the window of time during which the demand currents for all PG pins of the instance are active, that is, the demand current (curve) values are greater than the leakage current.



"Effective DvD Window": overlap window of currents of PG pins of instance

The following table specifies the reported DVD parameters for the effective DVD window.

Parameter	Description
Effective DVD	Average VDD-VSS voltage in a current stamping window
Rise effdvd	Effective DVD when any of the output signals rises
Fall effdvd	Effective DVD when any of the output signals falls
in_only effdvd	Effective DVD when there is no transition on output signals
fullsim effdvd	Minimum of the Effective DVD for each type across all clock cycles
fullsim avgTW	Minimum of the cycle AvgTW rise and fall data across all clock cycles

6.10. Reporting Dynamic Analysis Results

The following commands are available to report dynamic voltage drop results:

```
emir_reports.write_all_instance_voltages(av_dynamic_npv,
reports_dir+'/inst_voltage.rpt')
emir_reports.write_bump_currents(av_dynamic_npv,
reports_dir+'/bump_current.rpt')
emir_reports.write_bump_voltages(av_dynamic_npv,
reports_dir+'/bump_voltage.rpt')
emir_reports.write_demand_currents(av_dynamic_npv,
reports_dir+'/demand_current.rpt')
emir_reports.write_layer_voltage_report(av_dynamic_npv,
reports_dir+'/layer_voltage.rpt')
emir_reports.write_node_voltage_report(av_dynamic,
reports_dir+'/node_voltage.rpt')
emir_reports.write_supply_currents(av_dynamic,
reports_dir+'/supply_currents.rpt')
emir_reports.write_switch_report(av_dynamic, reports_dir+'/switch_report.rpt')
```

The following are typical report examples.

inst_voltage.rpt

The `write_all_instance_voltages` command outputs the instance voltage across each instance of the design.

#	loc_x	loc_y	eff_Vdd	max_pg_tw	min_pg_tw	min_pg_sim	max_pg_sim	min_vdd_tw	max_vss_tw	pg_arc	instance
#	(u)	(u)	(v)	(v)	(v)	(v)	(v)	(v)	(v)	pwr/gnd	
1075.615	484.505	1.054	1.1	0.9762	0.961	1.113	0.9763	8.716e-05	core3/VDD_INT/VSS core3/cts_inv_527224759		
1082.455	484.505	1.099	1.104	1.099	0.9621	1.113	1.1	0.0003504	core3/VDD_INT/VSS core3/HFSINV_46_15798		
1083.215	484.505	1.092	1.113	1.083	0.9623	1.113	1.089	0.006373	core3/VDD_INT/VSS core3/regfile_program_memory.AOI21_X1_923		
1080.365	481.705	1.096	1.113	1.094	0.9641	1.113	1.096	0.00238	core3/VDD_INT/VSS core3/regfile_program_memory.NOR2_X1_978		
1082.265	481.705	1.1	1.1	1.097	0.9649	1.113	1.098	0.000928	core3/VDD_INT/VSS core3/HFSINV_31_13751		
1082.645	481.705	1.085	1.1	1.093	0.965	1.113	1.095	0.001559	core3/VDD_INT/VSS core3/HFSINV_31_16038		
1083.215	481.705	1.099	1.1	1.099	0.9651	1.113	1.1	0.0008513	core3/VDD_INT/VSS core3/HFSINV_40_14888		

The following table describes the columns from the report example.

Column name	Description
loc_x	The lower left x-coordinate for the instance bounding box
loc_y	The lower left y-coordinate for the instance bounding box
eff_Vdd	Effective Vdd for the Instance
max_pg_tw	Maximum voltage across the instance in the timing window
min_pg_tw	Minimum voltage across the instance in the timing window
min_pg_sim	Minimum voltage across the instance for the duration of simulation
max_pg_sim	Maximum compression voltage across the instance for the duration of simulation

Column name	Description
min_vdd_tw	Voltage at the instance power pin when the voltage across the instance is minimum in the timing window
max_vss_tw	Voltage at the ground pin of the instance when the voltage across the instance is minimum in the timing window
pg_arc	Pair of the applicable power and ground nets of the instance
instance	The name of the instance

bump_current.rpt

The `write_bump_currents` command outputs bump current waveforms.

```
TitleText: Bump Currents
# Time (ps) I (A)
"VSS_1
    0.00  -0.0000005
    20.00  -0.0000804
    40.00  -0.0002011
    60.00  -0.0003409
    80.00  -0.0005351
   100.00  -0.0008380
```

bump_voltage.rpt

The `write_bump_voltages` command outputs bump voltage waveforms.

```
TitleText: Bump Voltages
# Time (ps) V (V)
"VSS_1
    0.00  0.0000002
    20.00  0.0003660
    40.00  0.0011909
    60.00  0.0029072
    80.00  0.0060383
   100.00  0.0085382
```

demand_current.rpt

The `write_demand_currents` command outputs the per voltage domain demand current waveforms.

```
TitleText: Demand Currents
# Time (ps) I (A)

"VSS
 0.00 -0.1367024
 20.00 -0.1656296
 20.00 -0.1656296
 20.00 -0.1656296
 20.00 -0.1656296
 20.00 -0.1656297
 20.00 -0.1656297
 40.00 -0.1997199
 40.00 -0.1997199
 40.00 -0.1997199
```

```
TitleText: Demand Currents
# Time (ps) I (A)

"core3/VDD_INT
 0.00 0.0054279
 40.00 0.0051139
 80.00 0.0051307
 100.00 0.0058439
 120.00 0.0084662
 140.00 0.0090562
 140.00 0.0090562
 160.00 0.0078603
 200.00 0.0052586
 220.00 0.0053284
```

layer_voltage.rpt

The `write_layer_voltage_report` command reports the minimum and maximum node voltage per layer for each net of the design.

#	min_x	min_y	min_voltage_drop	max_x	max_y	max_voltage_drop	layer	net
#	(u)	(u)	(v)	(u)	(u)	(v)	(v)	
660.885	432.645	0.01067	1081.78	483.165	0.1299	0.1192	core3/VDD_INT	metal1
1226.355	569.8	0.006487	176.96	1057.0	0.07906	0.07257	VSS	metal1
1226.07	0.0	0.008334	836.085	1136.8	0.07319	0.06486	VDD	metal1
1175.14	628.6	0.006742	31.615	1194.2	0.03627	0.02952	VSS	metal2
1174.94	628.6	0.006764	31.74	1194.2	0.03549	0.02873	VSS	metal3

The following table describes the columns from the report example.

Column	Description
min_x	The x-coordinate for the minimum voltage on the layer
min_y	The y-coordinate for the minimum voltage on the layer
min_voltage_drop	The minimum voltage drop seen on the layer

Column	Description
max_x	The x-coordinate for the maximum voltage on the layer
max_y	The y-coordinate for the maximum voltage on the layer
max_voltage_drop	The maximum voltage drop seen on the layer
layer_drop	The absolute difference between the max_voltage_drop and min_voltage_drop per layer for each net
net	The net name for the reported layer
layer	The layer name of the design

node_voltage.rpt

The `write_node_voltage_report` command outputs the voltage at each node.

```
# contents are sorted in decending order by layer, net, node_voltage
# loc_x    loc_y    layer   net    node_voltage
# (u)      (u)      (v)
71.6    64.8    metal12  VSS    0.00698
76.4    64.8    metal12  VSS    0.006974
71.6    63.2    metal12  VSS    0.00697
73.2    64.8    metal12  VSS    0.006967
74.8    64.8    metal12  VSS    0.006965
76.4    63.2    metal12  VSS    0.006964
73.2    63.2    metal12  VSS    0.006956
74.0    64.8    metal12  VSS    0.006954
74.8    63.2    metal12  VSS    0.006954
71.6    66.4    metal12  VSS    0.006954
76.4    66.4    metal12  VSS    0.006953
```

The following table describes the columns from the report example.

Column	Description
loc_x	The x-coordinate of the node
loc_y	The y-coordinate of the node
layer	The layer name of the reported node
net	The net name for the reported node
node_voltage	The minimum voltage seen at the node over the course of simulation

supply_currents.rpt

The `write_supply_currents` command outputs the per package domain supply current waveforms.

```
TitleText: Supply Currents
# Time (ps) I (A)

"VSS
    0.00      -0.1554655
    30.00     -0.1557408
    60.00     -0.1563817
    90.00     -0.1574274
   120.00     -0.1591184
   150.00     -0.1614533
   180.00     -0.1643164
   210.00     -0.1677248
   240.00     -0.1715997
   270.00     -0.1756741
   300.00     -0.1800714
   330.00     -0.1848136
   360.00     -0.1898976
```

For more information about the reporting commands, use the `help` command at the tool command prompt. For example,

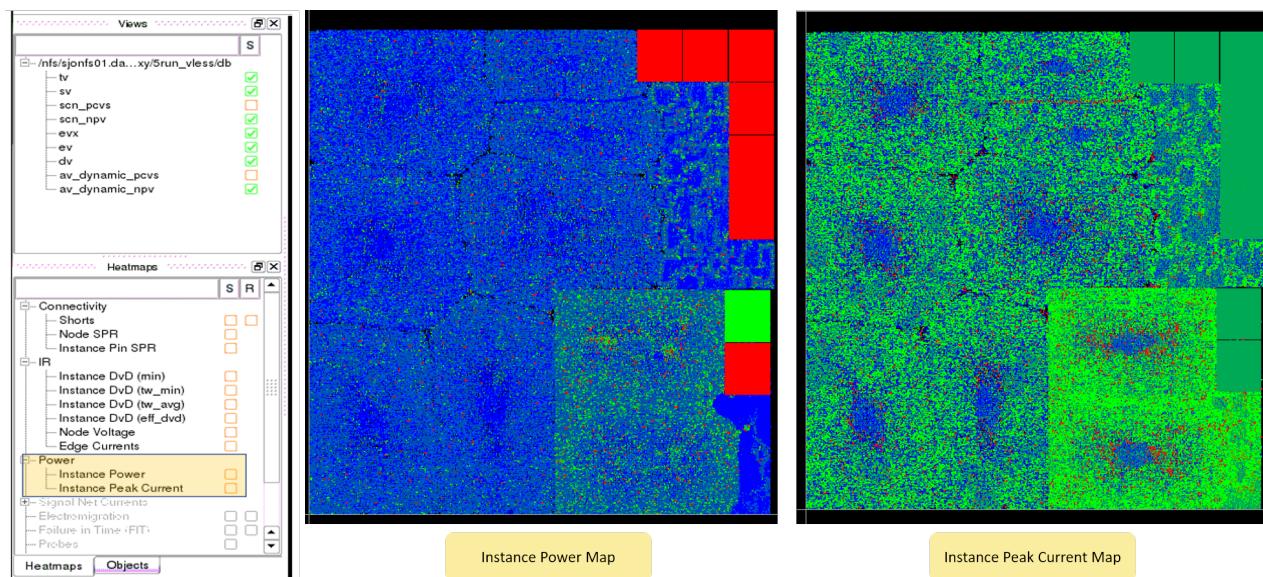
```
help(emir_reports.write_all_instance_voltages)
```

6.11. Viewing Dynamic Analysis Results in GUI

When the AnalysisView is completed, you can view various current and voltage heatmaps for dynamic voltage drop analysis. You generate these heatmaps in a manner similar to static IR drop analysis. For more information, see [Viewing Static Analysis Results in GUI](#) on page 129.

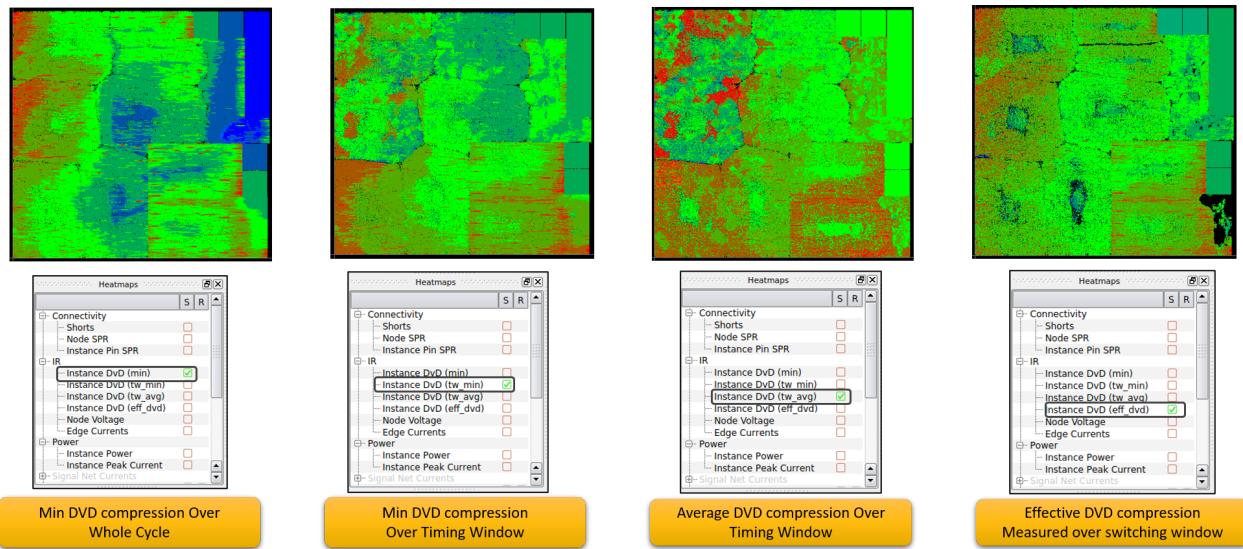
Instance Power and Peak Current Maps

The following figures show the instance power and corresponding peak current heatmaps.



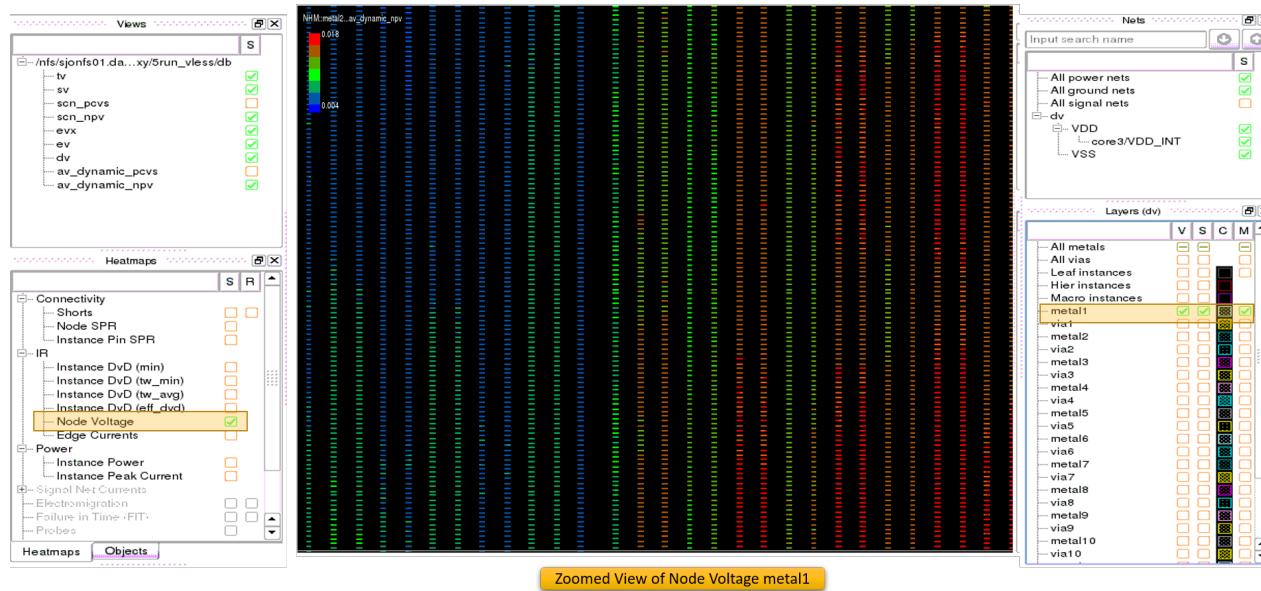
Instance Dynamic Voltage Drop Heatmaps

The **Instance DvD(min)**, **Instance DvD(tw_min)**, **Instance DvD(tw_avg)** and **Instance DvD(eff_dvd)** maps are available under **IR** in **Heatmaps**.



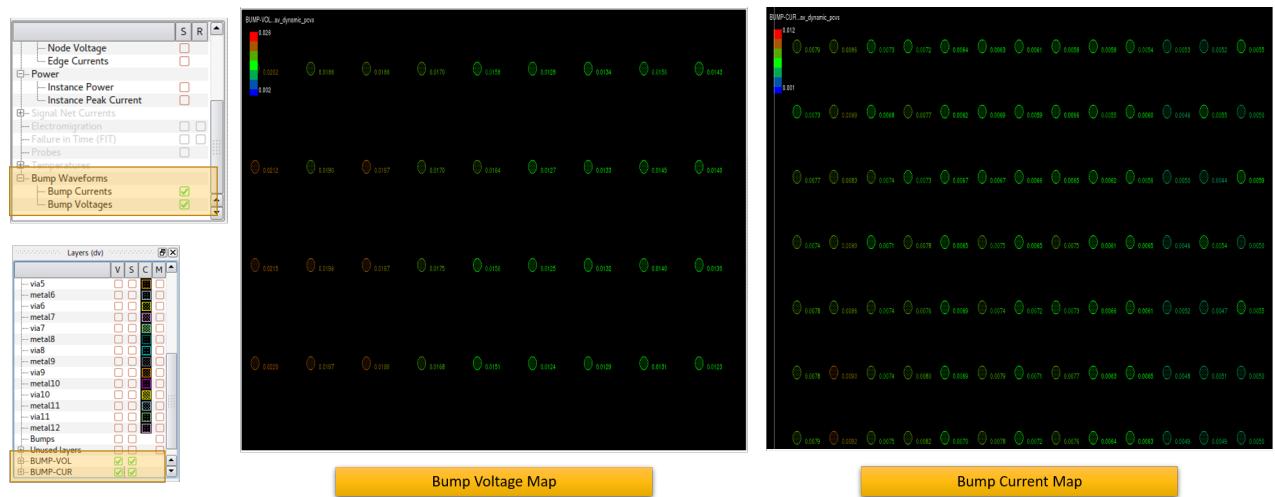
Node Voltage Drop Heatmap

The following is a magnified view of the node voltage drops.



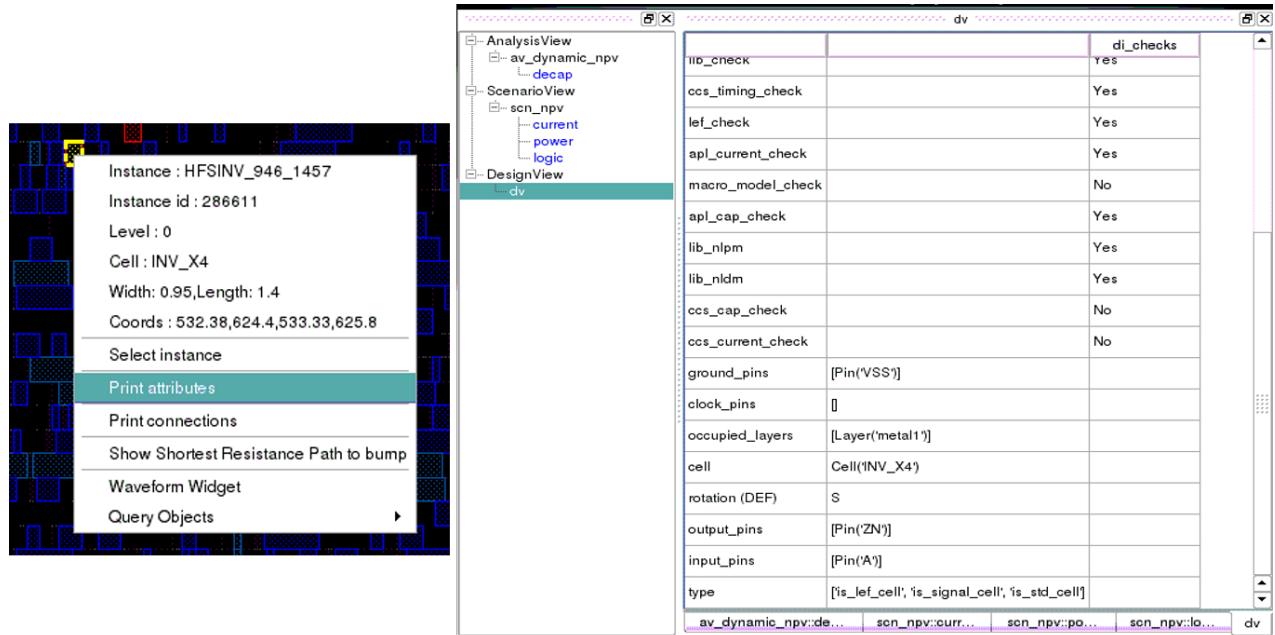
Bump Voltage and Current Heatmap

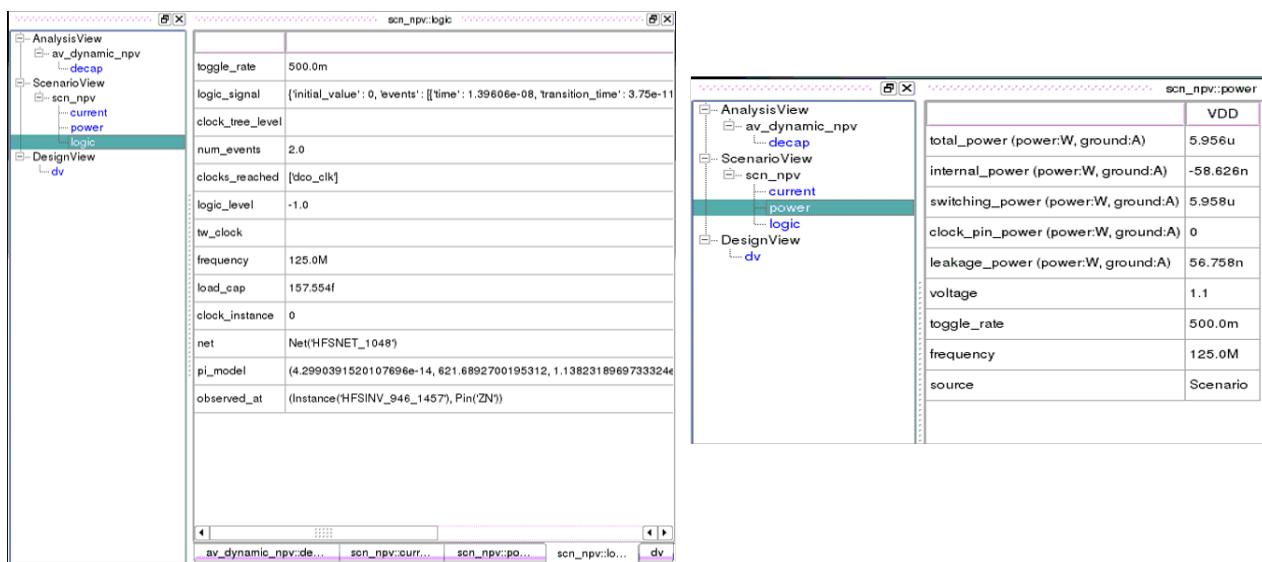
The following show the bump voltage and bump current maps.



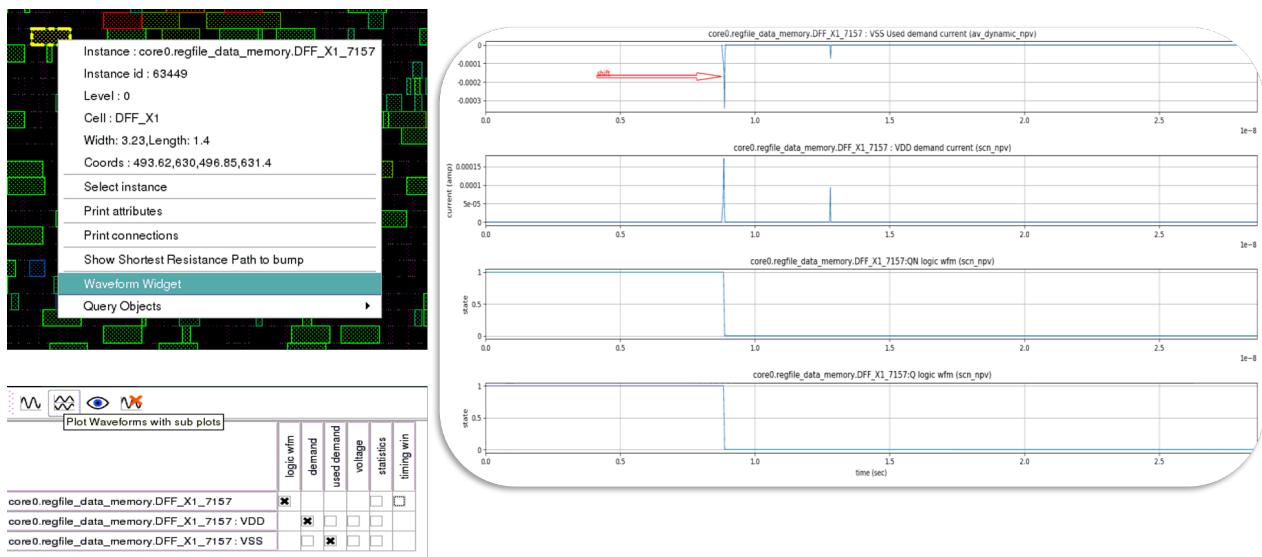
Instance Attributes and Waveforms

You can view the tables of power and logic attributes of any instance of the design using **Print attributes**.





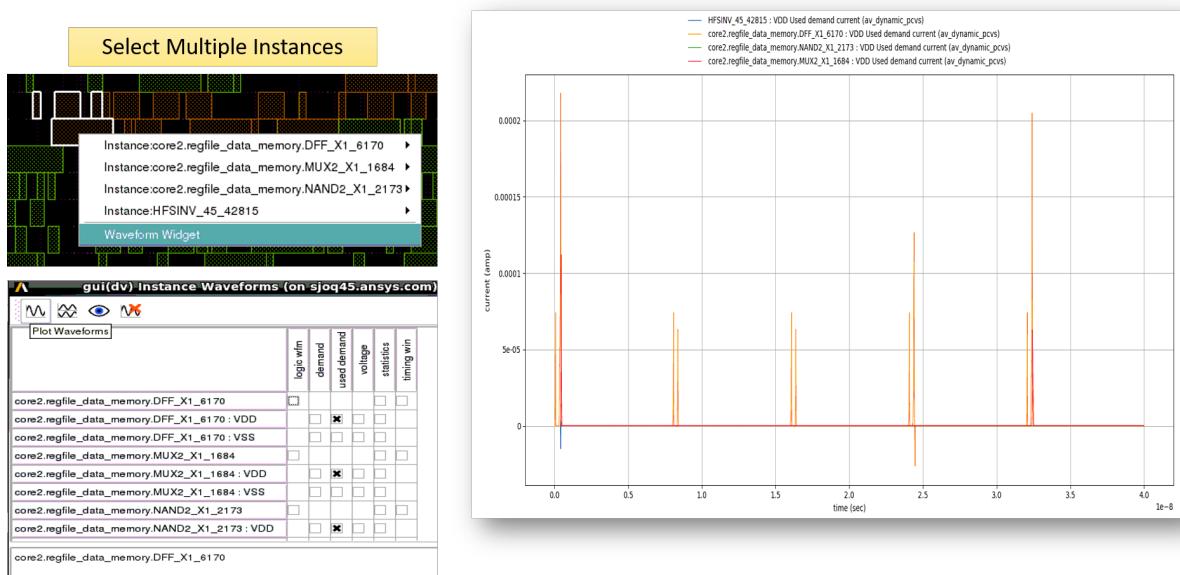
You can view waveforms of any instance of the design by using the **Waveform Widget**. The following figure shows the instance waveforms of used demand current at VSS, demand current at VDD, and the output logic. For information about the difference between demand current and used demand current, see [Performing Dynamic Voltage Drop Analysis](#) on page 210.



Viewing Simultaneous Waveforms of Multiple Instances

You can view simultaneously switching effects of a group of instances.

1. To select multiple instances, click the instances while holding the **Ctrl** key in the Layout window.
2. To invoke the waveform widget, right-click and select the **Waveform widget** and then view the current waveforms. The following figure shows co-located instances switching together.



6.12. Querying Analysis Data

Post AnalysisView, you can query design DVD data and plot corresponding waveforms. The following table shows the design-level query commands and the applicable databases.

Query Command	Database to Query
get_total_demand_currents()	ScenarioView
get_total_used_demand_currents()	AnalysisView
get_average_bump_voltages()	AnalysisView
get_total_bump_currents()	AnalysisView

The following examples show how to use these commands.

- The `get_total_demand_currents` command queries all the three voltage domains of the design. The `plot` command only plots the demand current for the VDD net.

```
scn_npv.get_total_demand_currents().keys()
[Net('core3/VDD_INT'), Net('VSS'), Net('VDD')]
plot(scn_npv.get_total_demand_currents() [Net('VDD')])
```

- The `get_total_used_demand_currents` command queries the total demand current from the `AnalysisView`.

```
plot(av_dynamic_pcvs.get_total_used_demand_currents())
```

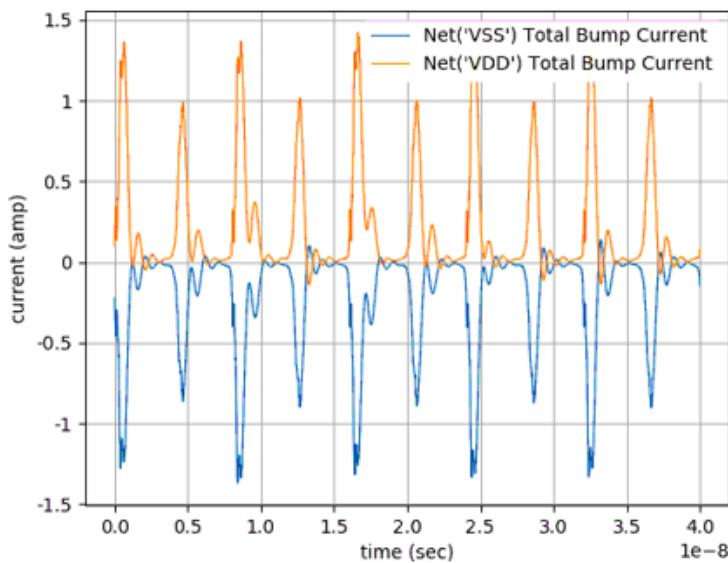
- The `get_average_bump_voltages` command queries both the VDD and VSS nets. The `plot` command plots the average bump voltages for the VDD net.

```
av_dynamic_npv.get_average_bump_voltages().keys()
[Net('VSS'), Net('VDD')]
plot(av_dynamic_npv.get_average_bump_voltages() [Net('VDD')])
```

- The `get_total_bump_currents` command queries all the PG nets.

```
plot(av_dynamic_pcvs.get_total_bump_currents())
```

A typical plot is as follows:



6.13. Querying Instance Data

To query instance data for DVD analysis, use the following commands. For more information, see respective command help.

Command with applicable database	Output
<code>tv.get_attributes()</code>	Attributes of the queried instance and specified output pins, such as, transition time, arrival time, clock root
<code>scn.get_attributes()</code>	Logic propagation details and events, power and current details
<code>scn.get_mode_usage_attributes()</code>	For macro, modes that got applied
<code>dv.cell_mode_attributes()</code>	Macro master cell, high/low/median mode details
<code>scn.get_logic_signals()</code>	Logic signal for each signal pin
<code>scn.get_instance_events()</code>	Event arcs considered for each instance
<code>scn.get_demand_current()</code>	Demand current waveform
<code>av.get_voltage_stats()</code>	Voltage drop parameters for the cell
<code>av.get_voltage()</code>	Voltage waveform. Stored if keep_stats is kept as Full
<code>av.get_bump_voltages()</code>	Bump voltage waveform
<code>av.get_current()</code>	Bump current waveform

The following examples show the usage and results of some of the instance-level query commands.

Instance Logic Details

In the following example, the reported `clock_instance` attribute value of `False` means that the instance is a data instance. `frequency` value is the applied clock frequency. `logic_level` is the logic depth of the combinational network.

Note: Logic level is valid only for logic propagation scenario, and not valid for a no-propagation vectorless scenario.

`num_events` and `toggle_rate` are the number of events and the toggle rate at the output pin.

```

scn_pcvs.get_attributes(Instance("core1.regfile_program_memory.OAI21_X1_845"))

{'logic': {
    'clock_instance': False,
    'clock_tree_level': None,
    'clocks_reached': ['dco_clk'],
    'frequency': 125000000.0,
    'logic_level': 10,
    'output_pins' : {Pin('ZN') : {'load_cap': 6.97e-14,
                                    'logic_signal': {'initial_value' : 0,
                                                    'events' : [
                                                        {'time' : 1.39e-08, 'transition_time' : 2.17e-10},
                                                        {'time' : 2.09e-08, 'transition_time' : 1.27e-10},
                                                        {'time' : 4.50e-08, 'transition_time' : 2.17e-10},
                                                       ],
                                                    'net': Net('core1.regfile_program_memory._13887_'),
                                                    'num_events': 3,
                                                    'pi_model': (1.12e-16, 321.74, 6.95e-14),
                                                    'toggle_rate': 0.5,
                                                    'tw_file': {'tw_clock': 'dco_clk', 'tw_max': 6.19e-09, 'tw_min': 4.99e-09}
                                                   }
                                 }
   },
    'dir' : 'r'},
    'dir' : 'f'},
    'dir' : 'r'},],},
    'net': Net('core1.regfile_program_memory._13887_'),
    'num_events': 3,
    'pi_model': (1.12e-16, 321.74, 6.95e-14),
    'toggle_rate': 0.5,
    'tw_file': {'tw_clock': 'dco_clk', 'tw_max': 6.19e-09, 'tw_min': 4.99e-09}
  },
  }
}

```

Instance Events

The following example reports transition time, load capacitance, and input to output pin combinations. The tool determines the demand current based on these values.

```
scn_npv.get_instance_events(Instance("core1.regfile_program_memory.DFF_X1_5510"))

{0: [ {'events': [ {'CK': 'f',
    'Q': '0',
    'input_event_time': 2.490399886312389e-08,
    'input_transition_time': 5.999999802552836e-11},
    {'CK': 'r',
    'Q': 'r',
    'input_event_time': 2.8903997417728533e-08,
    'input_transition_time': 6.499999843923021e-11}],
    'net_cap': 5.047869979272427e-15,
    'out_pin': 'Q',
    'total_cap': 7.538433988369189e-15},
    {'events': [ {'CK': 'f',
```

```

    'QN': '1',
    'input_event_time': 2.490399886312389e-08,
    'input_transition_time': 5.999999802552836e-11},
    {'CK': 'r',
    'QN': 'f',
    'input_event_time': 2.8903997417728533e-08,
    'input_transition_time': 6.499999843923021e-11}],
    'net_cap': 1.5443945109842758e-16,
    'out_pin': 'QN',
    'total_cap': 1.5443945109842758e-16}]}

```

Macro Modes

The output pin switching attributes are insufficient for macro instances because most of the power dissipation is due to the modes or input pin states. The following example queries and reports details of modes applied in each clock cycle.

```

scn_npv.get_mode_usage_attributes(Instance("core3/program_memory"))

{Pin('clk0'): {Pin('vdd'): [(4.6815000764866e-09, 'MEM_READ'),
                             (1.2681500294320358e-08, 'MEM_READ'),
                             (2.0681499179886487e-08, '_leakage_only_mode_'),
                             (2.8681498065452615e-08, 'MEM_WRITE')]}}

```

The following example reports the NLPM Boolean states or APL modes corresponding to different energy modes.

```

dv.get_cell_mode_attributes(Cell('sram_16_512_freepdk45'))

{'default_process': {
    'apl': {(1.0, 25.0): {}, (1.10000023841858, 25.0): {Pin('vdd'): {0: {
        '_high_energy_mode': ('MEM_WRITE', '1'),
        '_low_energy_mode': ('MEM_READ', '1'),
        '_median_energy_mode': ('MEM_WRITE', '1')}}}},
    'ccsp': {(1.0, 25.0): {}, (1.10000023841858, 25.0): {}},
    'nlpm': {(1.0, 25.0): {Pin(''): {61: {
        '_high_energy_mode': ('', '!csb0 * !clk0 * web0'),
        '_low_energy_mode': ('', 'csb0'),
        '_median_energy_mode': ('', '!csb0 * clk0 * !web0')}}}},
    (1.10000023841858, 25.0): {}}}}

```

Voltage Statistics

The following example reports a summary of the DVD data. By default, the tool does not store per cycle data, that is, `get_cycle_over_tw()` and `get_cycle_effdvd()` data is not stored. To store this data, you must set `keep_stats_level` to High.

```

av_dynamic_npv.get_voltage_stats(Instance('core2.regfile_data_memory.DFF_X1_2013'))

full_sim: Min: 1.08176 Max: 1.11268 Mean: 1.09976 Count: 1201
vss_at_min_: 0.00675058
full_sim_over_tw: Min: 1.08251 Max: 1.09684 Mean: 1.08915 Count: 39
vss_at_min_: 0.0082631
full_sim_effdvd: Min: 0 Max: 0 Mean: 0 Count: 0 vss_at_min_:

```

For more details, see command help as follows:

```
vs =
av_dynamic_npv.get_voltage_stats(Instance('core2.regfile_data_memory.DFF_X1_2013>
type(vs)
<class 'gp_export_py.SMInstanceArcVoltageStats'>
help(vs)
```

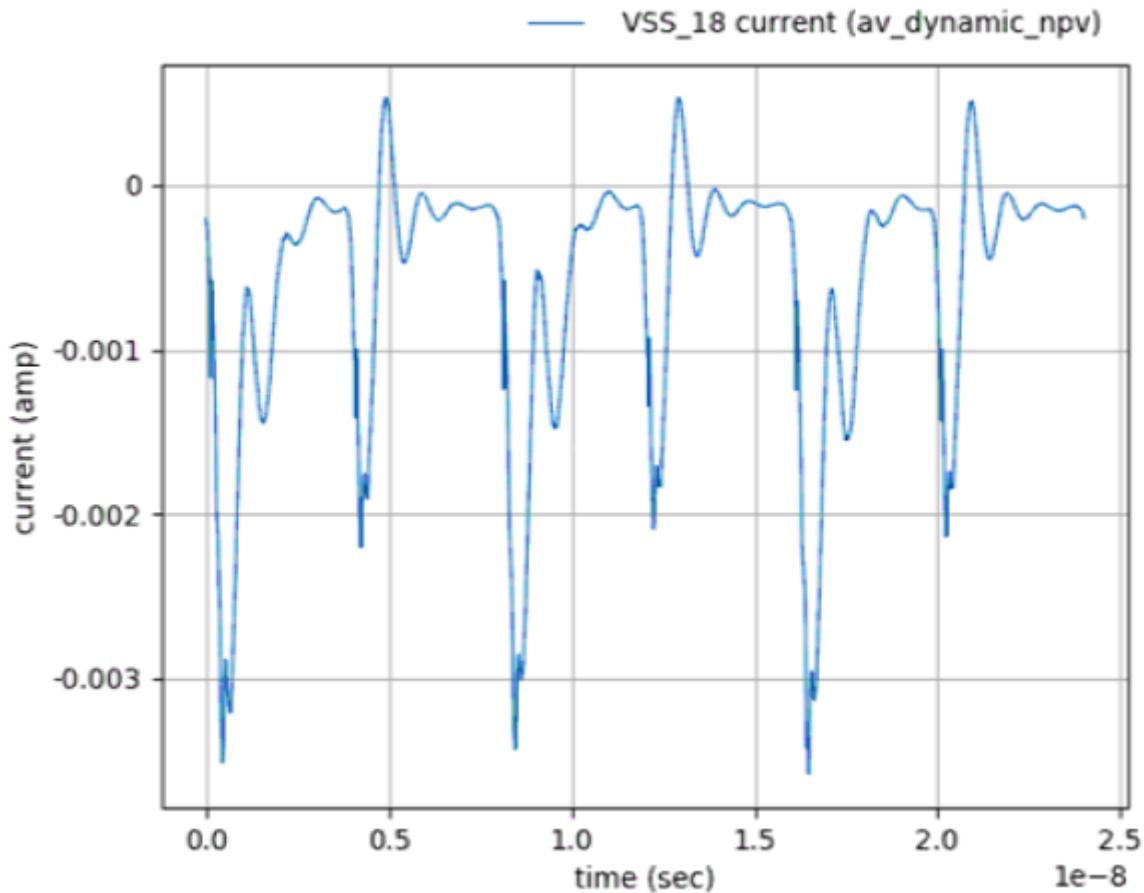
```
FUNCTIONS
get_cycle_effdvd()
    Get effective dvd statistics for one cycle of the simulation.

get_cycle_over_tw()
    Get statistics for one cycle of the simulation within timing window.
...
```

Bump Currents

The following example reports the per instance current waveform. Reports bump currents when Instance(''), that is, the top design. The pins are then the bumps of the design.

```
plot(av_dynamic_npv.get_current(Instance(''), Pin('VSS_18')), legend_location='top
    right')
plot(av_dynamic_pcvs.get_current(Instance(''), Pin('VDD_17')))
```



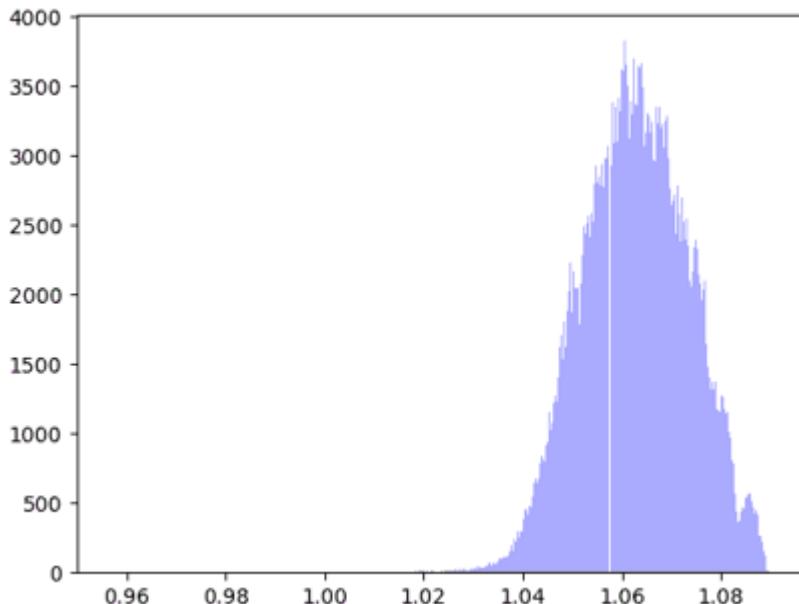
6.14. Generating DVD Histograms

Post AnalysisView, you can create and plot histograms for DVD analysis.

The following example shows how to generate and plot an instance voltage histogram. By default, the histogram y-axis scale is logarithmic. To use a decimal y-axis scale, set `log` to `False`. To view a bar histogram and not continuous data, specify `hist_type='bar'`.

```
vd_histogram = av_dynamic_pcv.get_instance_voltage_histogram()
plot(dvd_histogram,hist_type='bar',log=False,)
```

The following is a typical plot of instance voltage drop (x-axis) versus the number of design instances (y-axis) and shows how the voltage drop is spread across the instances.



The following example shows how to generate and plot a node voltage histogram. You can generate a histogram for each voltage domain and each layer of the PG grid.

```
nvh = av_dynamic_npv.get_node_voltage_histograms()
plot(nvh[Net('VDD')] [Layer('metal1')])
```

6.15. Recommended Flow

Ansys recommends the following flow for dynamic voltage drop analysis.

- Build Quality Metric (BQM)
 - Measure relative weakness per PG domain across entire power grid.
 - Run from power planning through final implementation to ensure consistent resistance gradients as design matures.
- PeakTW
 - Find placement issues by modeling possible peak current due to local simultaneous switching.
 - Run from early placement onward to identify issues as design is closed.

- Merge with BQM results to determine a hazard metric for every instance PG pin.
- Power-Constrained Vectorless Scenario (PCVS) at block level
 - Scenario for specific operating mode(s) with realistic propagated switching.
 - Run transient AnalysisView (typically 10 to 100ns duration each).
 - Recommended for block level run's signoff.
 - Optional: analyze timing effect of voltage drop.
- No-Propagation Vectorless (NPV) Scenario
 - Switch almost all instances in relatively few clock cycles.
 - Run transient AnalysisView with this scenario and fix outliers.
 - Excellent frequency domain power coverage for long duration simulations with full chip and package response.

6.16. Dynamic PG Electromigration Analysis

This section describes the electromigration checks to compare the RMS and peak currents flowing through a wire or via against foundry-specified limits and report the violations. The current values are obtained from the AnalysisView. The electromigration limits or rules are input from the technology file information stored in the TechnologyView. For information about static electromigration analysis, see [Static Electromigration Analysis](#) on page 138.

Dynamic electromigration analysis is described in the following topics:

- [Performing RMS and Peak Electromigration Analysis](#) on page 229
- [Reporting RMS and Peak Electromigration Results](#) on page 230
- [Viewing DVD Electromigration Results in GUI](#) on page 231

6.16.1. Performing RMS and Peak Electromigration Analysis

To perform electromigration analysis of dynamic currents in PG nets, use the `create_electromigration_view` command.

In the following example, the command checks for electromigration violations of RMS currents on all the metal layers and vias of the design.

```
pem_rms = db.create_electromigration_view(av_dynamic_npv, **pem_rms_args)

pem_rms_args = dict(
    tag='pem_rms',
    settings={'mode' : 'rms', 'delta_t_rms' : 10}
    options=options)
```

`tag` is the name of the ElectromigrationView database, `'mode' : 'rms'` means that the electromigration check is for RMS currents, and `delta_t_rms` specifies the temperature tolerance for RMS electromigration checking (in Celsius).

In the following example, the command checks for electromigration violations of peak currents on all the metal layers and vias of the design.

```
pem_peak = db.create_electromigration_view(av_dynamic_npv, **pem_peak_args)

pem_peak_args = dict(
    tag='pem_peak',
    settings={'mode' : 'peak'}
    options=options)
```

mode: 'peak' means that the electromigration check is for peak currents.

For more details, see `help(SeaScapeDB.create_electromigration_view)`.

6.16.2. Reporting RMS and Peak Electromigration Results

To report electromigration violations, use the `write_em_metal_report` and `write_em_via_report` commands after the ElectromigrationViews (`pem_rms` and `pem_peak`) are created, as shown in the following example:

```
reports_dir = 'reports/powerem_analysis'
gp_util.makedirs(reports_dir)
for em_type,em_view in { 'RMS_EM' : pem_rms , 'PEAK_EM' : pem_peak }.items():

    emir_reports.write_em_metal_report(em_view,
    reports_dir+"/"+em_type+'_metal_report.rpt')
    emir_reports.write_em_via_report(em_view,
    reports_dir+"/"+em_type+'_via_report.rpt'
```

The following are typical examples of metal and via reports. For more information about electromigration metal and via reports, [Reporting Power Electromigration Results](#) on page 139.

Peak Electromigration Metal Report

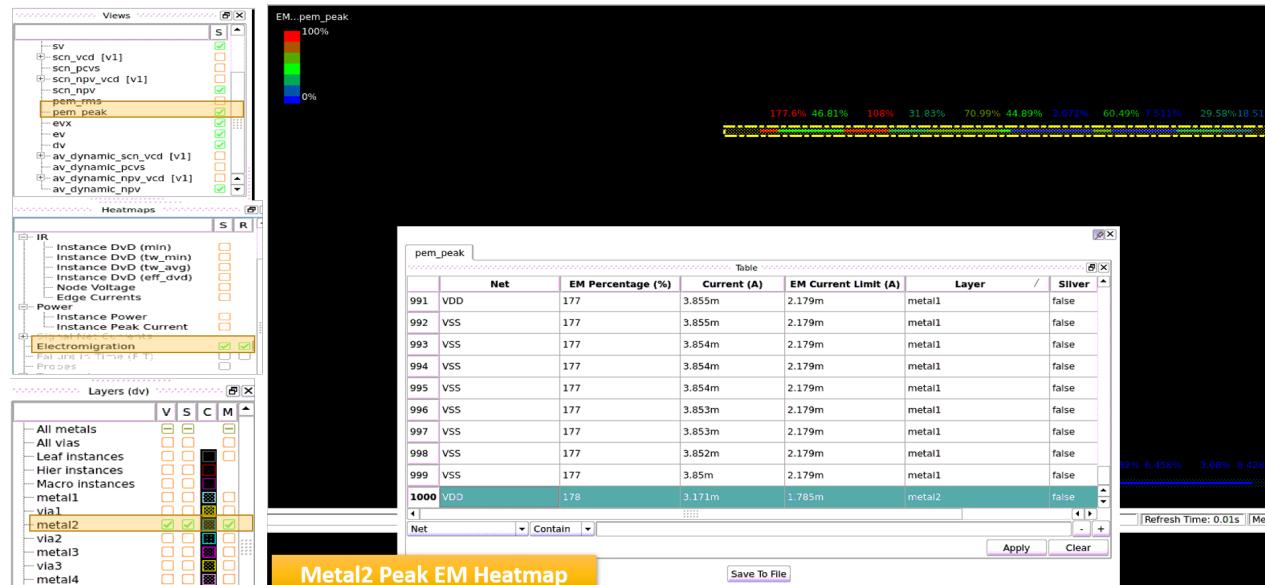
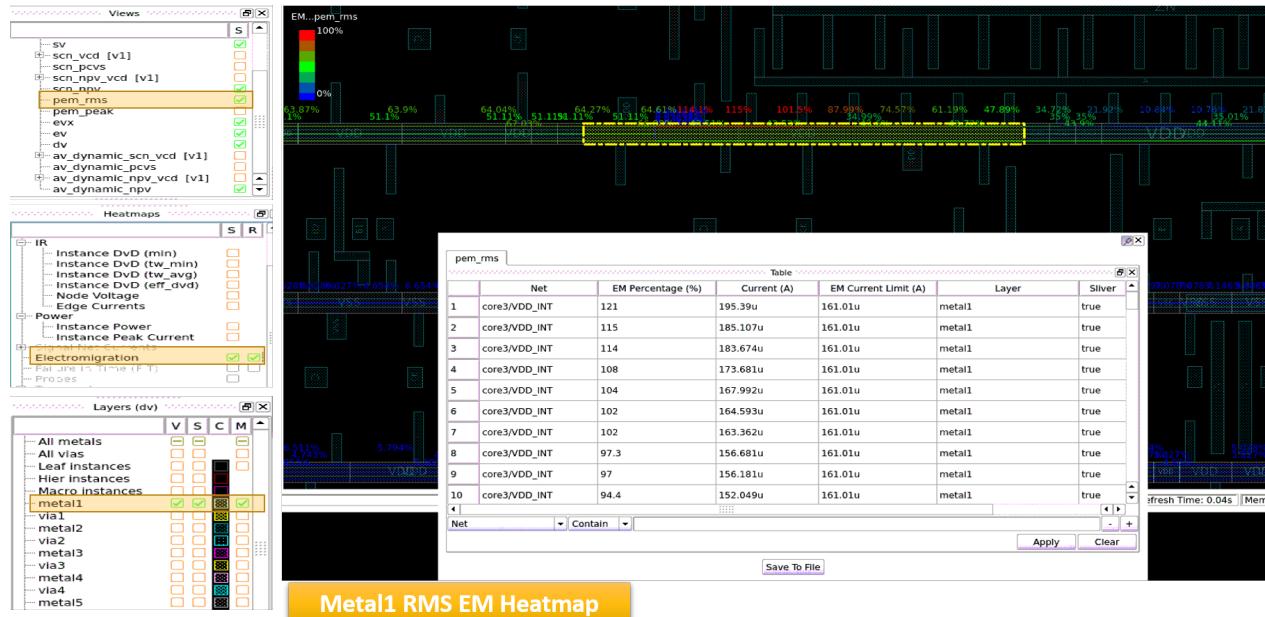
#	em_type	layer	from	to	length	width	current	constraint	violation	status	net
#			(u)	(u)	(u)	(u)	(A)	(A)	(%)		
		metal1	(678.0775,420.105)	(678.25,420.105)	153.3	0.17	0.01111	0.002179	509.7	FAIL	VSS
		metal1	(678.605,420.105)	(678.615,420.105)	153.3	0.17	0.0111	0.002179	509.3	FAIL	VSS
		metal1	(677.46,420.105)	(677.84,420.105)	153.3	0.17	0.01041	0.002179	477.8	FAIL	VSS
		metal1	(677.31,420.105)	(677.46,420.105)	153.3	0.17	0.009715	0.002179	445.8	FAIL	VSS
		metal1	(1075.89,543.305)	(1076.09,543.305)	295.1	0.17	0.009612	0.002179	441.1	FAIL	VSS
		metal1	(1076.09,543.305)	(1076.255,543.305)	295.1	0.17	0.0096	0.002179	440.6	FAIL	VSS
		metal1	(1076.945,543.305)	(1077.065,543.305)	295.1	0.17	0.009594	0.002179	440.3	FAIL	VSS
		metal1	(1077.065,543.305)	(1077.6,543.305)	295.1	0.17	0.009589	0.002179	440	FAIL	VSS
		metal1	(1077.825,543.305)	(1078.3675,543.305)	295.1	0.17	0.009581	0.002179	439.7	FAIL	VSS

RMS Electromigration Via Report

#	em_type	layer	loc_x	loc_y	via_length	via_width	current	constraint	violation	status	net
#			(u)	(u)	(u)	(u)	(A)	(u)	(%)		

6.16.3. Viewing DVD Electromigration Results in GUI

The following figures show typical electromigration heatmaps with browsers showing the violations in descending order of RMS or peak current. For more information about how to view power electromigration heatmaps in GUI, see [Viewing Electromigration Results in GUI](#) on page 140.



6.17. Handling Decoupling Capacitance

Decoupling capacitors have a significant impact on the final dynamic voltage drop of leaf instances. Sources, such as PG or signal wire capacitors, intrinsic cell capacitors or intentional decoupling capacitor cells contribute to dynamic simulation results. Decoupling capacitors improve the dynamic voltage drop on neighboring

switching instances by augmenting the instantaneous current demand. It is also important to accurately model on-chip capacitance for system-level analysis, such as Chip Power Model (CPM) generation.

6.17.1. Stamping Decoupling Capacitance

The following two main types of capacitance are used in dynamic simulation and CPM generation:

- PG grid capacitance

Wire capacitance extracted from the PG grid.

- Capacitance contributed by leaf level instances

This is of three types:

- Intentional device capacitance

Contributed by on-chip intentional decoupling capacitors. You input these to the tool in APL Cdev (.cdev) format.

- Intrinsic or device capacitance of functional instances

Contributed by internal nodes of functional instances. You input the intrinsic capacitance (ESC) information contained in APL Cdev models to the tool through LibertyView.

- Load Capacitance connected to instance output pin

When an instance does not switch in CMOS designs, the load is directly connected to the power or ground pin through the R_{on} resistance. This acts like a decoupling capacitor to the neighboring switching instances. The total load capacitance is equal to sum of net capacitance and the receiver instance input pin capacitance. The tool uses the SPEF file to derive the net capacitance. You input the SPEF file to the tool by using the `create_extract_view_from_files` command. If SPEF files are not available, the tool extracts the net capacitance. The receiver input capacitance is computed by using Liberty files.

This section explains how the device and load capacitance are stamped in dynamic simulation. For intentional decoupling capacitor instances, the intrinsic capacitance always connected to the power grid are replaced with effective series capacitor (ESC) and effective series resistor (ESR) network during simulation. The functional cells consume current only when they switch. Otherwise, they act as decoupling capacitors. The intrinsic capacitance of these non-switching instances are connected to the power grid through effective series resistors (ESR).

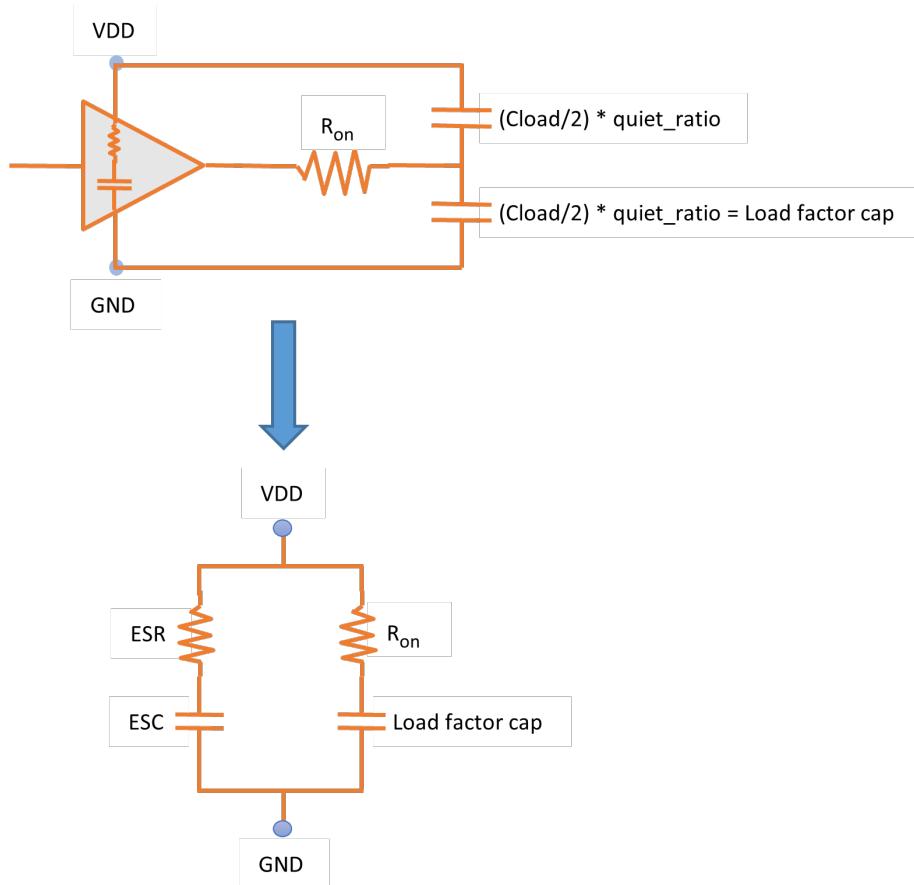
The tool uses the following methodology to stamp load capacitance:

RedHawk-SC assumes load capacitors are coupled 50% to power and 50% to ground pins during the transient analysis, hence the factor of load capacitance is stamped to VDD and GND nets and it is computed using the following formula:

```
quiet_ratio = 1 - (total_transition_time/total_time)
load_factor_cap = (C_load/2) * quiet_ratio
```

`total_transition_time` is the time during simulation that the instance transitions from 0 to 1 and 1 to 0. This is computed from the rise and fall transition times and load capacitance of that instance.

For example, consider a buffer driving the load capacitor (C_{load}) with a series resistance R_{on} as shown in the figure. R_{on} is the net parasitic resistance computed from ExtractView. The first part of the figure shows the schematic view of the buffer.

Figure 51. Schematic to Circuit Representation of Decoupling Capacitor Stamping

The second part is a circuit representation of the same which goes to the transient analysis. Here, the intrinsic effective series capacitance (ESC) is connected to the grid through the effective series resistance (ESR). This circuit is in parallel to the load capacitance factor that is connected to the power grid through the series resistance, R_{on} .

The two methods of stamping are `scale_toggle` (default and more accurate) and `legacy`. The following table shows the differences in stamping the signal load capacitance between the two methods.

<code>legacy</code>	<code>scale_toggle</code>
The load capacitance is always stamped only for power network only based on the driver instance output state.	The load capacitance is stamped for both the power and ground networks.
Load capacitance is not stamped if the instance switches in the simulation duration.	The load capacitance is stamped based on the activity factor.
Load capacitance is stamped for instances with constant output states of 1.	50% of the load capacitance is stamped to both the power and ground network with output at 1.
Load capacitance is not be stamped for instances with constant output state of 0.	50% of the load capacitance is stamped to both the power and ground network with output at 0.

6.17.2. Querying Decoupling Capacitance Data

The following functions are available to query decoupling capacitance data from AnalysisView.

- To query the total decoupling capacitance statistics of the design for dynamic simulation, use the `AnalysisView.get_decap_stats` function. The following example shows a typical output of the `AnalysisView.get_decap_stats` function:

```
>>> av.get_decap_stats()

{'VDD': {'num_instances': 1000,
          'num_instances_with_decap': 999,
          'total_decap': 6.493843344540603e-08,
          'total_intrinsic_cap': 7.987977807451808e-08,
          'total_pg_route_cap': 1.898468582339774e-08,
          'total_pin_load_cap': 3.1149608088928624e-08,
          'total_signal_load_cap': 9.220013386564688e-08},
 'GND': {'num_instances': 1000,
          'num_instances_with_decap': 999,
          'total_decap': 6.493843344540603e-08,
          'total_intrinsic_cap': 7.987977807451808e-08,
          'total_pg_route_cap': 1.898468582339774e-08,
          'total_pin_load_cap': 3.1149608088928624e-08,
          'total_signal_load_cap': 9.220013386564688e-08},
 'total': {'average_instance_coverage': 0.999149583838286,
           'cdie': 1.489970840253679e-07,
           'total_decap': 3.151667488160064e-08,
           'total_intrinsic_cap': 4.315952683403479e-08,
           'total_pg_route_cap': 8.746573199152142e-09,
           'total_pin_load_cap': 1.6263880521223528e-08,
           'total_signal_load_cap': 4.93104285893568e-08}}
```

The `get_decap_stats` function also returns the decoupling capacitance statistics per domain and reports the total decoupling capacitance statistics with components.

In the example,

- `total_load_cap` is sum of `total_pin_load_cap` (total receiver pin capacitance) and `total_signal_load_cap` (signal routing capacitance).
- `total_decap` is the total intentional decoupling capacitance.
- `total_intrinsic_cap` is the intrinsic capacitance of other functional instances.
- `cdie` is the total on-chip capacitance at low frequency close to the DC point.

```
cdie = total_decap + total_intrinsic_cap + total_pg_route_cap +
      total_pin_load_cap + total_signal_load_cap
```

The final total decoupling capacitance values represent the effective decoupling capacitance between the power and ground network.

- To query the decoupling capacitance statistics of an instance, use the `AnalysisView.get_attributes` function. The function returns the load capacitance and intrinsic capacitance of the instance with their respective series resistances that are stamped in transient simulation of the instance.

```
>>> av.get_attributes(Instance('Xbuff1'))

{'decap': {Pin('VDD'): {'driver_res': 1626.16,
                        'driver_total_cap': 2.014e-13,
```

```

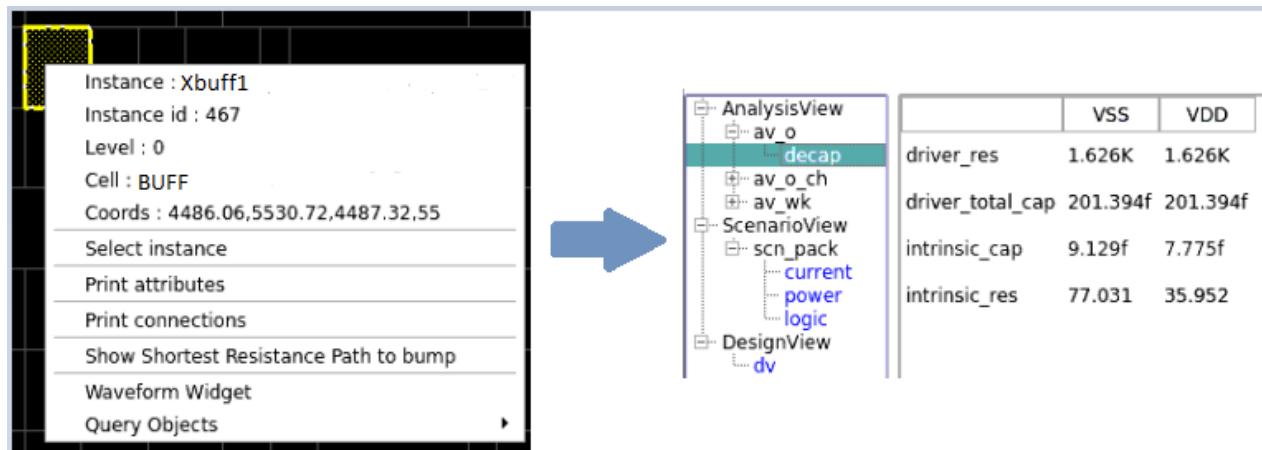
        'intrinsic_cap': 7.775e-15,
        'intrinsic_res': 35.95},
    Pin('VSS'): {'driver_res': 1626.16,
        'driver_total_cap': 2.014e-13,
        'intrinsic_cap': 9.129e-15,
        'intrinsic_res': 77.03}}}

```

You can also view the decoupling capacitance statistics of an instance in the GUI. See [Viewing Instance Decoupling Capacitance Statistics in GUI](#) on page 235.

Viewing Instance Decoupling Capacitance Statistics in GUI

To view the decoupling capacitance statistics of an instance in the GUI, select the instance and right-click, and then click **Print attributes**. Select **decap** under **AnalysisView**. This prints the decoupling capacitance statistics, as shown in the following figure:



7: Dynamic Analysis With Vectors

Vector files contain switching or event information for a pin or net of the design. Typically, vector files are available in the Value Change Dump (VCD) and Fast Signal DataBase (FSDB) formats. This chapter discusses various ways in which vector files are read into the RedHawk-SC tool, different aspects of file processing, and settings to process vector files for dynamic voltage drop (DVD) analysis.

Note: The RedHawk-SC tool supports both the VCD and FSDB input file formats. FSDB file versions 5.9 and earlier are supported. In the chapter, the term VCD refers to both VCD and FSDB files.

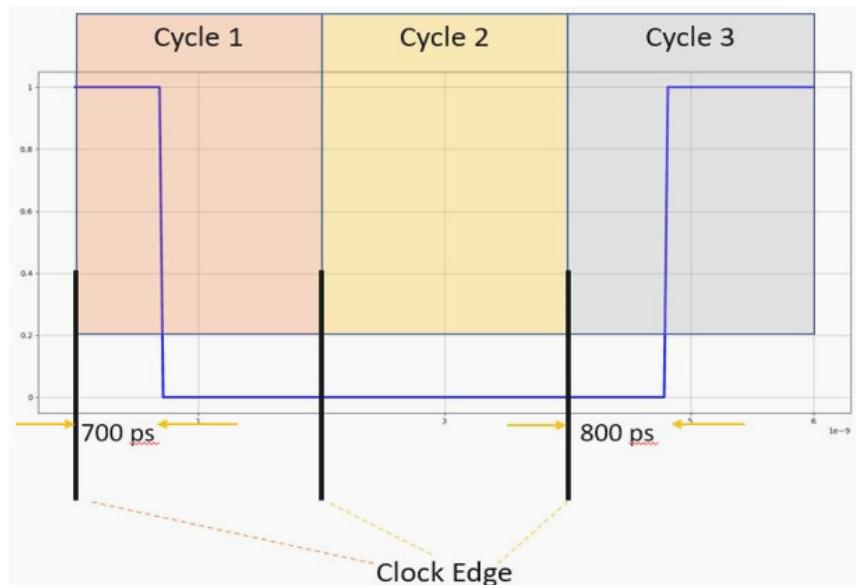
The following topics are discussed in this chapter:

- [Switching Scenario](#) on page 237
- [Processing VCD Files](#) on page 238
- [ValueChangeView](#) on page 238
- [Vector-Based Dynamic Analysis](#) on page 249
- [Checking VCD Annotation](#) on page 259

7.1. Switching Scenario

In dynamic voltage drop analysis, ScenarioView supplies currents at PG pins to the next stage. To do this, ScenarioView relies on switching or event information at logic pins. When a rise or fall event occurs at an input or output pin, dynamic current is drawn from VDD or supplied to VSS pins. The type of these logical events (such as, input-only rise, clock-only rise, output fall, output rise) and the exact time at which they occur are determined using various means or flows. This section describes using VCD input to derive these events.

The following figure shows the events for a simple buffer with clock period of 2 ns. For an analysis duration of 3 clock cycles or 6 ns, two events occur at the output pin, a fall at 0.7 ns (700ps) and a rise at 4.8 ns.



Considering clock edges at 0 ns, 2 ns, and 4 ns, the tool considers a fall event at the first clock cycle and a rise event at the third clock cycle as shown in the following table.

Table 11: Switching Events and Time

Cycle	Cycle1	Cycle2	Cycle3
Switching event	Fall	Low	Rise
Time of switching	700 ps	-	800 ps

VCDs can be of different types. RTL VCDs are available at early RTL stage of the design. These have events only at sequential instances. The delay values are also not exact.

Gate VCDs are available post synthesis and include events for almost all instances of the design. Gate VCDs can be of two types, non-delay annotated and delay annotated.

Though events occur correctly within each clock cycle in non-delay annotated gate VCDs, the exact time of their occurrence cannot be considered from the VCD. There can be zero-delay VCD, unit-delay VCD, or a VCD with unreliable delay values.

Delay-annotated VCD files are gate VCD files output by annotating accurate time delays from an SDF file. The exact time of events can be considered from delay-annotated VCD files. For non-delay gate VCDs and RTL VCDs, there is a need to determine event times. This can be done by using either RedHawk-SC's event propagation methodology or arrival time values from the STA file.

7.2. Processing VCD Files

To load the VCD files for processing, use one of the following views:

- ValueChangeView (VCV)

VCV can store the complete temporal information, that is, each event, preferably for a short duration. Because VCV stores events and SwitchingActivityView stores only toggle information, VCV is generally used for dynamic voltage drop analysis. See [ValueChangeView](#) on page 238.

- SwitchingActivityView

SwitchingActivityView stores the activity factor, that is, the number of toggles and corresponding high or low probability, typically for a long duration. In general, SwitchingActivityView is used for static voltage drop analysis. For more information, see [Setting Switching Activity](#) on page 105.

Alternative flows where you use SwitchingActivityView data for dynamic analysis and VCV data for static analysis, are possible but less likely. Therefore, such flows are not described in this chapter.

7.3. ValueChangeView

ValueChangeView (VCV) holds value changes, that is, events seen from VCD file at a corresponding pin or net. No other processing happens in the view. It reads VCD file, maps it to the pin or the net of the design, and then stores events. All VCD files or slices are read into the same (or different) VCV as different time slices.

The following concepts are covered under this section:

- [Need and Concept](#) on page 239
- [Name Mapping](#) on page 240
- [Concept of Time Slices](#) on page 248

- [X State Handling](#) on page 248
- [Parsing Long Duration FSDB](#) on page 249

7.3.1. Need and Concept

In RedHawk-SC, you can only read and store the VCD file. Events are not processed in this case. The same settings are applicable for RTL, gate, delay, or no-delay VCD. There is no concept of clock or power in VCV.

Syntax

VCV can be created using the command `<SeaScapeDB>.create_value_change_view`. The following table lists the arguments:

Table 12: VCV Arguments

Argument	Description
<code>design_view</code>	DesignView object. This is mandatory argument.
<code>vcd_files</code>	List of VCD files.
<code>settings</code>	Dictionary of settings used for various controls.
<code>options</code>	View creation options.
<code>tag</code>	Name of the view.

For detailed description on the settings, see [Name Mapping](#) on page 240. Other important arguments are explained in the following section:

VCD Files Input

`vcd_files` argument takes in a list of dicts. Each of these dicts are for one VCD file. The following table lists the keys inside each dict:

Table 13: vcd_files Setting

dict	Description
<code>'file_name'</code>	States the path to the VCD file.
<code>'instances'</code>	States the list of hierarchical instance names that are applied to the file. If the VCD is given to the entire design, empty quotes are enough (""). If the VCD is given to multiple hierarchies, you need to list them out.

dict	Description
'time_slices'	States the value change information that is stored as time slices. You can read in different time windows or time duration from a VCD file. These are specified as a list of dict. The following sub-keys are used: <ul style="list-style-type: none"> • 'slice_name': States the name of the slice. Data from VCV is accessed by slice name and hence each slice name has to be unique per VCV. • 'start_time': States the starting time of the VCD slice that is considered. Default value is the start of the VCD. • 'stop_time': States the ending time of the VCD slice that is considered. Default value is the end of the VCD. So, you need to omit 'start_time' and 'stop_time' if the entire VCD is read into the slice. • 'instances': States how to further restrict a slice to specific instances. • 'cell': States how the slice is applicable to all instantiations of the master cell/block.
'preamble'	States the front path or strip path that has to be removed from signal names in VCD file. Typically, this is the test bench name.
'cell'	Specifies the cell name, if the VCD is applied to all instantiations of a master block.
'top_only'	Reads only the signals at the top level of hierarchy.

The following is an example of the VCD flow.

```
vcf_files = [
    {'file_name': '../design_data/vcd/top.fsdb',
     'instances': [''],
     'preamble' : 'label0/label1/top0/',
     'time_slices' : [
         {'slice_name' : 'fsdb1',
          'stop_time' : 7.99920188e-05,
          'start_time' : 7.99845e-05, }],
    },]
```

7.3.2. Name Mapping

Name mapping refers to mapping or matching done between names in the design data and the names coming from VCD. The design data is setup from DEF file and each of the other input files need to be mapped correctly to instance, pin, or net of the design data.

Name mapping can be done for other inputs also, such as STA and SPEF files. For the gate VCD cases, mapping is smooth, the way it's done for STA or SPEF file. The presence of RTL VCD makes name mapping an onerous task. The names in RTL VCD are from RTL stage itself and they need to be mapped to the DEF or gate level names, which are created in synthesis stage.

The names (or difference in names) between RTL or VCD and DEF or gate level netlist generally follow some common patterns. By default, the tool automatically maps the names. This advanced name mapping feature improves VCD annotation for certain type of designs and RTL VCD naming styles. When large design data is to be mapped to the VCD information, the tool creates multiple jobs. In particular, a large block of the design is divided into small chunks and an independent parallel job is launched for each chunk. It reduces the peak memory consumed to read VCD or FSDB files, especially for large designs and large VCD files. Some cases might take more runtime.

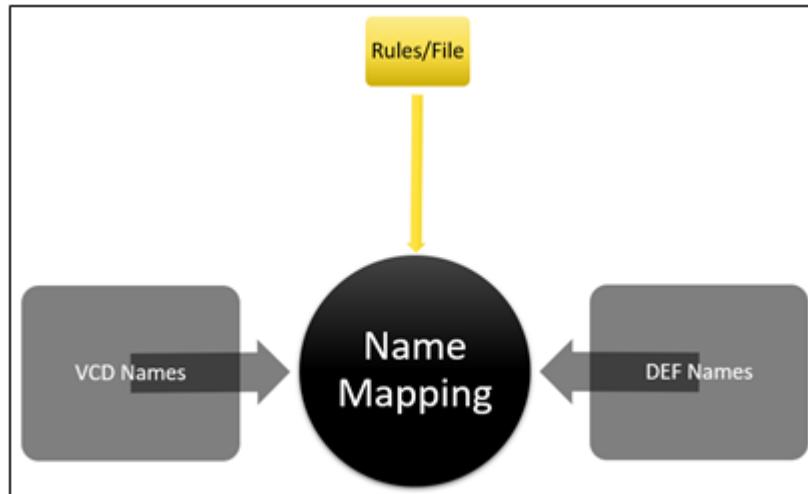
To disable advanced name mapping, set the `use_basic_name_mapping` key to `True`:

```
vcv_settings = {'namemap' : {'use_basic_name_mapping' : True}}
vcv = db.create_value_change_view(..., settings = vcv_settings, ...)
```

However when the names have uncommon patterns, it becomes difficult to do automatic mapping. Then the name mapping requires additional help, such as name mapping files or name mapping rules. This section describes the name mapping concept through name mapping files or name mapping rules.

The terms RTL and VCD are used interchangeably. Also, the terms DEF and gate are used interchangeably.

Figure 52. Concept of Name Mapping



Preamble Settings

For dumping in FSDB or VCD, you need to instantiate the design in a test bench. Preamble is the path that must be taken out so that the name in VCD matches with the name in RedHawk-SC.

The following example shows that 'X/Y/Z' is the preamble and it is removed from the VCD full name, so that both the names match.

VCD full name: **X/Y/Z/core3/net1_one**

Name seen from RedHawk-SC: **core3/net1_one**

Preamble: **'X/Y/Z'**

VCD is given for the top level.

The following is another example in which preamble is 'tb_JKL/chip_core_inst/C_wrapper_inst'.

VCD Full Name: **tb_JKL/chip_core_inst/C_wrapper_inst/count_reg_3/_Q**

Name seen from RHSC: **C_block_2/count_reg_3/_Q**

preamble: **'tb_JKL/chip_core_inst/C_wrapper_inst'**

VCD is given for block '**C_block_2**'

Setting Annotation Type

Before going into the details of name mapping, an important setting to understand in the VCD based flow is the 'annotation_type' setting. The default 'annotation_type' is 'all'. When the VCD file contains only nets or only pins, set the 'annotation_type' key to one of the following:

- 'net': VCD is mapped only to net names in the design or DEF.
- 'pin': VCD is mapped only to the signal pin names in the design.

The following example shows how to specify 'annotation_type'.

```
settings = {
    'annotation_type' : 'net',}
vcv = db.create_value_change_view(..., settings = settings, ...)
swa = db.create_switching_activity_view(..., settings = settings, ...)
```

7.3.2.1. Name Mapping File

Name mapping file shows which signal in VCD is mapped to which instance pin or net in DEF. Typically, this can be dumped from the synthesis setup on its own or you can run a script. The format and input settings for VCV and SWA are exactly the same. The file is given as a separate setting and there is the flexibility to provide multiple files for block/top.

A typical format of the file is shown in the following example:

```
rtl_namemap_files = [
{'file_name' : 'example.txt',
'rtl_preamble' : 'RTLP1.RTLP2',
'gate_preamble' : 'GATEP1.GATEP2',
'delimiter' : '/',
'map_symbol' : '=>',
'instances' : ['']]}
settings = {
    'annotation_type' : 'All',
    'namemap' : {
        'namemap_files' : rtl_namemap_files }}
vcv = db.create_value_change_view(..., settings = settings, ...)
swa = db.create_switching_activity_view(..., settings = settings, ...)
```

The following table describes the keys used for name mapping:

Table 14: Name Mapping Keys

Value	Description
'rtl_preamble'	The string gets removed from the RTL entry in name map file. This is useful if there are extra hierarchies in the name map file.
'gate_preamble'	The string gets removed from the gate entry in name map file. For example, when the name map file is for the whole design and it must be applied for a small block inside.
'delimiter'	The string '/' is used as hierarchy delimiter in DEF formats. It is '.' in netlists and RTL level data. The right delimiter used in the map file is specified here.

Value	Description
'map_symbol'	In this string, name map file could have a map symbol between gate and RTL names. It could also be a simple space.
'instances'	The string lists the (block) instances to which the name map is applied. Each entry of this list gets added to both RTL entry and gate entry after the removal of preamble.
'map_type'	The default value is 'gate_to_rtl', where name map file has the gate level name first and then the RTL name. In the reverse case, where name map file has the RTL name first and then the gate level name, you can set this to 'rtl_to_gate'.

7.3.2.2. Examples of Name Mapping File

The following examples describe the name mapping file settings:

Simple Single File for Whole Design

In this example, there is only one file used. The file has name mapping information for the whole design. There are no extra strings or hierarchies in the file.

The `map.txt` file contains:

```

qr_hier_1_qr2_hier2_0_s_namea_reg_27/q qr_hier_1/qr2_hier2_0/s_namea[27]
qr_hier_1_qr2_hier2_0_s_namea_reg_28/q qr_hier_1/qr2_hier2_0/s_namea[28]
qr_hier_1_qr2_hier2_0_s_namea_reg_29/q qr_hier_1/qr2_hier2_0/s_namea[29]

rtl_namemap_files = [ {'file_name' : 'map.txt',
                      'rtl_preamble' : '',
                      'gate_preamble' : '',
                      'delimiter' : '/',
                      'map_symbol' : '',
                      'instances' : ['']} , ]

vcv_settings = {
    'annotation_type' : 'All',
    'namemap' : {
        'namemap_files' : rtl_namemap_files } }
vcv = db.create_value_change_view(
design_view=dv,
vcd_files=vcd_files,
tag='vcv',
settings = vcv_settings,
options=options)

```

The symbol for `delimiter` is '/'. So, in this case, the symbol for `map_symbol` is ''. Preambles are empty.

Extra Hierarchies for RTL Name

In this example also, a single name map file is used for the whole design. There is an extra string to the VCD part.

The map.txt file contains:

```

qr_hier_1_qr2_hier2_0_s_namea_reg_27/q =>
tb_test/DUT/qr_hier_17qr2_hier2_0/s_namea[27]
qr_hier_1_qr2_hier2_0_s_namea_reg_28/q =>
tb_test/DUT/qr_hier_17qr2_hier2_0/s_namea[28]
qr_hier_1_qr2_hier2_0_s_namea_reg_29/q =>
tb_test/DUT/qr_hier_17qr2_hier2_0/s_namea[29]

rtl_namemap_files = [ {'file_name' : 'map.txt',
    'rtl_preamble' : 'tb_test/DUT', 
    'gate_preamble' : '',
    'delimiter' : '/',
    'map_symbol' : '=>',
    'instances' : ['']},]
vcv_settings = {
    'annotation_type' : 'All',
    'namemap' : {
        'namemap_files' : rtl_namemap_files } }
vcv = db.create_value_change_view(
    design_view=dv,
    vcd_files=vcd_files,
    tag='vcv',
    settings = vcv_settings,
    options=options)

```

The symbol for delimiter is '/'. The symbol '=>' comes in between the names and it denotes map_symbol. The extra string is marked as rtl_preamble.

Extra Hierarchies for RTL and Gate Name

In this example also, a single name map file is used for the whole design. There are extra strings to both the names.

```

chip/t1/qr_hier_1_qr2_hier2_0_s_namea_reg_27/q =>
tb_test/DUT/qr_hier_17qr2_hier2_0/s_namea[27]
chip/t1/qr_hier_1_qr2_hier2_0_s_namea_reg_28/q =>
tb_test/DUT/qr_hier_17qr2_hier2_0/s_namea[28]
chip/t1/qr_hier_1_qr2_hier2_0_s_namea_reg_29/q =>
tb_test/DUT/qr_hier_17qr2_hier2_0/s_namea[29]

rtl_namemap_files = [ {'file_name' : 'map.txt',
    'rtl_preamble' : 'tb_test/DUT',
    'gate_preamble' : 'chip/t1', 
    'delimiter' : '/',
    'map_symbol' : '=>',
    'instances' : ['']},]
vcv_settings = {
    'annotation_type' : 'All',
    'namemap' : {
        'namemap_files' : rtl_namemap_files } }
vcv = db.create_value_change_view(
    design_view=dv,
    vcd_files=vcd_files,
    tag='vcv',
    settings = vcv_settings,
    options=options)

```

To make things more clear:

Full name of signal in VCD: tb_test/DUT/qr_hier_1/qr2_hier2_0/s_namea[27]

Name in DEF: qr_hier_1_qr2_hier2_0_s_namea_reg_27/_q

In this example, name map file is for the whole (bigger) design and analysis is done for a small block inside it (chip/t1).

Name Map File for a Block

In this example, you must look at block-specific name map files. The setup has following two name map files:

The `cell_top` file contains:

```
p_qrst_uv0_i_wx0_reg_3/q p_qrst_uv0_i_wx0[3]
p_qrst_uv0_i_wx12_reg_13/q p_qrst_uv0_i_wx12[13]
p_qrst_uv0_i_wx3_reg_10/q p_qrst_uv0_i_wx3[10]
```

The `blockM_wrapper_0_map` file contains:

```
chip_core_inst/blockM_top_wrapper_0_inst/blockM_top_inst/p_qrst_uv0_
i_wx0_reg_3/q
chip_core_inst/blockM_top_wrapper_0_inst/blockM_top_inst/p_qrst_uv0_i_wx0[3]
chip_core_inst/blockM_top_wrapper_0_inst/blockM_top_inst/p_qrst_uv0_
i_wx12_reg_13/q
chip_core_inst/blockM_top_wrapper_0_inst/blockM_top_inst/p_qrst_uv0_i_wx12[13]
chip_core_inst/blockM_top_wrapper_0_inst/blockM_top_inst/p_qrst_uv0_
i_wx3_reg_10/q
chip_core_inst/blockM_top_wrapper_0_inst/blockM_top_inst/p_qrst_uv0_i_wx3[10]
```

For two blocks, `cell_top` is a simple file with no extra preambles. For the remaining two blocks, `blockM_wrapper_0_map` is a more complex file with preambles.

```
rtl_namemap_files = [
    {'file_name' : './cell_top',
     'rtl_preamble' : '',
     'gate_preamble' : '',
     'delimiter' : '/',
     'map_symbol' : ' ',
     'instances' : [
         'chip_core_inst/blockN_top_wrapper_3_inst',
         'chip_core_inst/blockN_top_wrapper_2_inst']},]

    {'file_name' : './blockM_wrapper_0_map',
     'rtl_preamble' : 'chip_core_inst/blockM_top_wrapper_0_inst/blockM_top_inst',
     'gate_preamble' : 'chip_core_inst/blockM_top_wrapper_0_inst/blockM_top_inst',
     'delimiter' : '/',
     'map_symbol' : ' ',
     'instances' : [
         'chip_core_inst/blockM_top_wrapper_0_inst',
         'chip_core_inst/blockM_top_wrapper_1_inst']}],]

vcv_settings = {
    'annotation_type' : 'All',
    'namemap' : {
        'namemap_files' : rtl_namemap_files }}}
vcv = db.create_value_change_view(
    design_view=dv,
    vcd_files=vcd_files,
    tag='vcv',
```

```
settings = vcv_settings,
options=options)
```

7.3.2.3. Name Mapping Rules

Name mapping rules are used when there are uncommon patterns in RTL VCD and gate/DEF names, which restrict auto mapping. Moreover, a name map file is also not available. In such cases, one can input a set of patterns after studying the names. As with the name map file, the format is exactly the same between VCV and SwitchingActivityView (SWA). It goes as another entry into 'namemap' key of settings.

For name mapping rules, there is one set of rules, which applies to the whole design. This is a list of dict, which is a list of rules. Each rule needs keys. 'type' is the type of rule. The types of rules are described in the following table:

Table 15: Name Mapping Rule Types

Value	Description
'rtl_to_gate'	The change specified is applied to RTL or VCD name.
'define_bus_rule'	The rule is to define how register buses in RTL are expected in gate netlist name.
'generate_label'	This is a special rule to generate blocks in RTL VCD.
'from'	The rule is for the pattern to be replaced.
'to'	The rule is for the pattern that comes in place to replaced one.

Examples of each rule type are explained below. It shows the RTL and gate name and then highlights the different or unexpected pattern. Then, the name map rule is mentioned.

Example1

VCD/RTL name: top1/u_hier2/sub3/**xtra_string_rtl**/pointer_addr[11] DEF/gate name:
top1/u_hier2/sub3/pointer_addr_11/_Q

Can be handled as : rtl_name_mapping_rules = [

```
{'type': 'rtl_to_gate', 'from' : "xtra_string_rtl", 'to' : ""},]
```

An extra string appears in the RTL name. This is mapped by using 'rtl_to_gate' rule.

Example2

VCD/RTL name: top_wrapper1/**LABEL_BLOCK[1]**.top_core/sub_core_switch/**GEN_LABEL[6]**.
hier_clk_gate_/block_reg/data[0]

DEF/gate name:

top_wrapper1/top_core_1/sub_core_switch/hier_clk_gate_6/block_reg/data_reg_0/_Q

Can be handled as: rtl_name_mapping_rules = [

```
{'type' : 'generate_label', 'from' : "LABEL_BLOCK" , 'to' : ""},  
'type' : 'generate_label', 'from' : "GEN_LABEL" , 'to' : ""},]
```

In this case, the number along with the label name in RTL name goes to the end of next hierarchy's name in the gate netlist name. This is where generate_label rule is used.

Example3

VCD/RTL name: hier_partition_4/sub2/count_q[0]
 DEF/gate name: hier_partition_4/sub2/count_q_**regex0x/Q** Can be handled as:
 rtl_name_mapping_rules = [
 {'type' : 'define_bus_rule', 'from' : "[%d]", 'to' : "**_regex%dx**"},]

The commonly seen bus transformation rule is that name[4] becomes name_reg_4. This is done automatically. For other kinds of bus transforms, use 'define_bus_rule' as shown below.

```
vcv_settings = {
    'annotation_type' : 'All',
    'namemap' : {
        'namemap_rules' : [
            {'type' : 'rtl_to_gate', 'from' : 'xtra_string_rtl', 'to' : ''},
            {'type' : 'generate_label', 'from' : 'LABEL_BLOCK', 'to' : ''},
            {'type' : 'define_bus_rule', 'from' : '[%d]', 'to' :
                '_regx%dx'},]]}
vcv = db.create_value_change_view(
    design_view = dv,
    vcd_files = vcd_files_vcv,
    settings = vcv_settings,
    options = options,
    tag = 'vcv')
```

For mapping any RTL name to gate name, each of the rule is applied to see whether a match is possible. You can also try default auto mapping with known patterns. So, even if the special patterns/rules applies to only a subset of names in the design, mapping takes place properly for the rest of the design.

7.3.2.4. MBFF Specific Settings

Multibit flip flops (MBFFs) are the type of registers that have more than one output pins. When synthesis is done, more than one registers from the RTL are added to this one register. Hence, mapping RTL VCD names to MBFFs is a difficult task. However, there are some common conventions used for mapping MBFFs and by understanding these conventions, most of the MBFFs can be mapped.

Typically, the names of individual registers that goes into each MBFF is concatenated, for example, if count[7], count[6], count[5], and count[4] are together made into one 4-bit MBFF, the expected name is as follows:

<>/ABC__count_reg_7_MB_count_reg_6_MB_count_reg_5_MB_count_reg_4_

Here, it is enough to identify the prefix and delimiter and input to VCV/SWA as shown in the following example:

```
vcv_settings = {
    'namemap': {
        'multi_bit_prefixes': [ABC__],
        'multi_bit_delimiters': [MB_]},}
```

Both the keys input a list of strings. More than one style can be given, and different types of conversion styles can be mapped. Also, you must notice the single underscore (_) after the string for delimiter (MB_). This has to match properly. For _MB_, the leading underscore belongs to the register name itself as seen from the last name (count_reg_4_). The trailing underscore belongs to the delimiter.

Another example is as follows:

```
reg_data[9] reg_data[10] ->
CDN_MBIT_reg_data_regex9x_MB_reg_data_regex10x
```

```

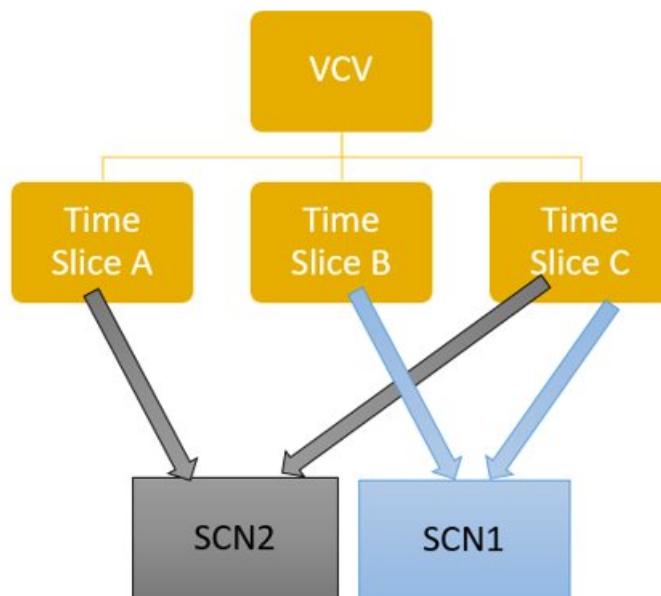
vcv_settings = {
    'namemap': {
        'multi_bit_prefixes': [CDN_MBIT_],
        'multi_bit_delimiters': [_MB_],
    },
    'namemap_rules' : [
        {'type' : 'define_bus_rule', 'from' : '[%d]', 'to' : '_regx%dx'}, ]}}

```

The prefix is `CDN_MBIT_` and the delimiter is `_MB_`. In this case, the register name does not end with underscore(' _ '), so both trailing and leading underscores (' _ ') of `_MB_` belongs to the delimiter.

7.3.3. Concept of Time Slices

Time slices are single or multiple VCD slices read from each VCD or block. VCV data is not directly input to the downstream views. VCV data is stored and referenced downstream by these time slices. Multiple VCD files can be read into a VCV. As shown in the following figure, multiple time slices from a single or multiple VCD files are used to create a single or multiple ScenarioViews. Also, you should use correct slice names so that it easy to refer to them in downstream views.



7.3.4. X State Handling

In the VCD file, there are X states, that is don't care logic states. These cannot be used as such for further logic processing and have to be converted to either low (0) or high (1) state. There are different options available for logic processing. The key `convert_x` inside settings is used to control the conversion of X state signals. The values are defined in the following table:

Table 16: X State Values

Value	Description
'low_state'	Converts all X to 0/low state.
'high_state'	Converts all X to 1/high state.

Value	Description
'toggle'	Both X to logic and logic to X are considered as toggle.
'ignore'	X logic is ignored. This is the default behavior.
'retain'	It considers X to logic as toggle and keeps logic to X as X itself. As such, this is not intended to be used for dynamic analysis.

Consider an example as the following VCD file waveform:

```
x -> 1 -> x -> 0 -> x -> 0
'convert_x': 'low_state' gives the value 0 -> 1 -> 0 -> 0 -> 0 -> 0
'convert_x': 'ignore' gives the value 1 -> 0 -> 0
```

Settings goes into VCV as follows:

```
vcv_settings = {
    'convert_x' : 'low_state'
}
vcv = db.create_value_change_view(..., settings = vcv_settings, ...)
```

The same setting is also applicable in the SwitchingActivityView.

Note: Only 'high_state', 'low_state', and 'retain' are accepted in SWA, with 'retain' being the default value.

7.3.5. Parsing Long Duration FSDB

RedHawk-SC has built-in automatic parallelism to efficiently read big FSDB files. These could be FSDBs with a lot of signals, FSDBs with extremely long duration, or a combination of both these factors. The flow used is called as automatic chunking in which the FSDB is split into multiple chunks signal wise and time wise. Parsing each chunk becomes a separate job which can be done by separate workers. This is done automatically by the tool.

Note: The parallel reading is done only for the FSDB files and not for the VCD files. FSDB APIs are used for splitting and parsing in parallel.

7.4. Vector-Based Dynamic Analysis

ScenarioView is required to perform dynamic voltage drop analysis. As already discussed, there are broadly two ways of creating scenarios, VCD and vectorless methods. This section describes VCD based dynamic analysis. For vectorless methods, see [Vectorless Dynamic Analysis](#) on page 143.

The advantages and disadvantages of VCD based dynamic analysis over vectorless dynamic analysis are as follows.

Advantages:

- For time-based VCD files, the exact arrival time comes from the VCD.

- Scenario generated is deterministic as it honors the activity from VCD files.

Disadvantages:

A design can have multiple VCD files. Several functional modes are associated with a design and VCD files can exist for a good number of these modes. The VCD files can also be of long duration. For dynamic voltage drop analysis, it is difficult to get the right VCD file and the right time slice in the VCD file. Procuring VCD files is also difficult. Gate VCDs are available late in the design cycle, and their generation consumes time and other resources. RTL VCDs can be made available early in the design cycle but coordinating between various design teams to get the correct VCD file and the correct time slice is a cumbersome task.

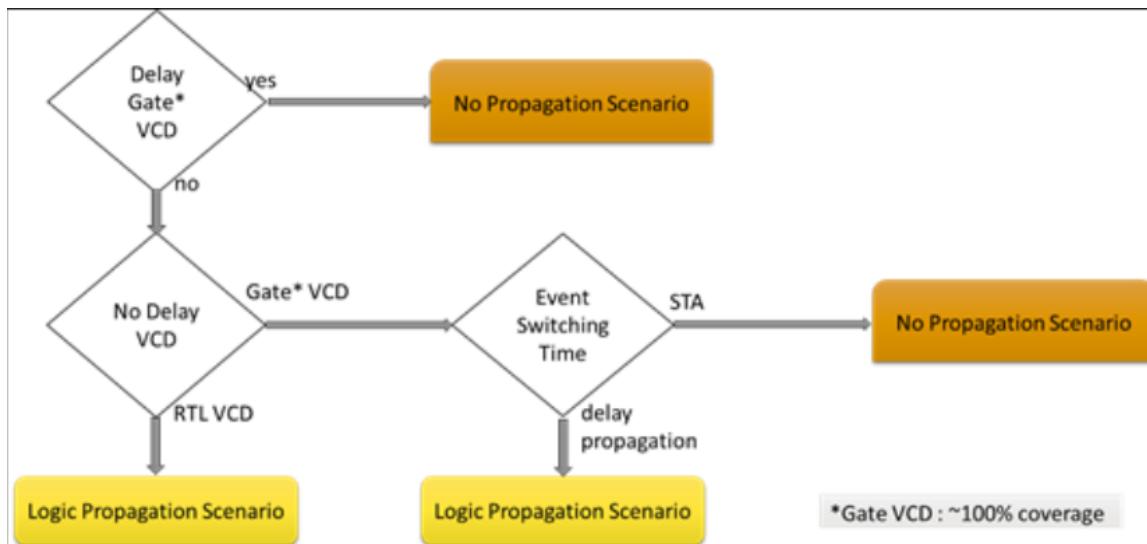
The following concepts are covered under this section:

- [Logic Propagation and No Propagation Scenarios](#) on page 250
 - [Types of VCD Files](#) on page 250
 - [VCD Settings in ScenarioView](#) on page 254
 - [Filtering Glitches from VCD in NPV ScenarioView](#) on page 258

7.4.1. Logic Propagation and No Propagation Scenarios

ScenarioView can be of two types, logic-propagation and no-propagation. ValueChangeView (VCV) data goes into dynamic ScenarioView. Both the logic-propagation and no-propagation ScenarioViews take in VCV. The following figure shows an overview of the ScenarioView flow.

Figure 53. Vector-Based ScenarioView Flows



For gate VCD, close to 100% annotation is expected, and hence no propagation is required. Therefore, no-propagation vectorless (NPV) ScenarioView is preferred for gate VCD. For specific cases where gate VCD is non-delay and STA data for arrival time is not good, logic-propagation ScenarioView is used for delay propagation.

7.4.2. Types of VCD Files

This section describes the handling of top-level and block-level VCD files input to the ValueChangeView.

You can have a single VCD file for the whole design, a set of block-level VCD files, or a mix of top and block level VCD files.

Further, one VCD file can be input to multiple blocks with a particular duration for one block and a different duration for another block. There can be multiple VCD files for the same block with low and high activity time slices.

Single VCD File for Entire Design

Single top-level VCD file is used for the whole design.

Note: Ensure that `value_change_data` points to the right VCV and slice name. This becomes important when the complexity of settings increases.

The following example describes the scenario:

```

vcd_files =
    {'file_name': '../design_data/vcd/top.fsdb',
     'instances': [''],
     'preamble' : 'label0/label1/top0/',
     'time_slices' : [
         {'slice_name' : 'fsdb1',
          'start_time' : 79983e-09,
          'stop_time' : 79993e-09, }],
    },
vcv_settings = {'annotation_type' : 'All'}
vcv = db.create_value_change_view(
    design_view=dv,
    vcd_files=vcd_files,
    settings = vcv_settings,
    tag='vcv',
    options=options)

value_change_data = [{'view' : vcv, 'slice_name' : 'fsdb1'},]

scn_npv_vcd = db.create_no_prop_scenario_view(...,
scenario_duration=10e-09, value_change_data = value_change_data, ...)
```

Multiple VCD Files

There are two VCD files in this case, one for each block. They are referred by the slice name. For details of time slicing, see [Concept of Time Slices](#) on page 248.

The following example describes the scenario:

```

vcd_files = [
    {'file_name': '../design_data/fsdb/block_name.0.fsdb',
     'instances': ['hier1/hier2/core0/block_name'],
     'preamble' : 'label1/label4/hier1/hier2/core0/block_name/',
     'time_slices' : [
         {'slice_name' : 'core0',
          'start_time' : 11086e-09,
          'stop_time' : 11089e-09, }],
    },
    {'file_name': '../design_data/fsdb/block_name.1.fsdb',
     'instances': ['hier1/hier2/core10/block_name'],
```

```

'preamble' : 'label1/label4/hier1/hier2/core1/block_name/',
'time_slices' : [
    'slice_name' : 'core1',
    'start_time' : 11082e-09,
    'stop_time' : 11085e-09,}],
],]

vcv_settings = {'annotation_type' : 'All'}
vcv = db.create_value_change_view(
    design_view=dv,
    vcd_files=vcd_files,
    settings = vcv_settings,
    tag='vcv',
    options=options)

value_change_data = [{'view' : vcv, 'slice_name' : 'core0'},
                     {'view' : vcv, 'slice_name' : 'core1'},]
scn = db.create_scenario_view(..., scenario_duration=3e-09,
                             value_change_data = value_change_data, ...)
```

The VCD data stored in VCV is referred in downstream views using time slice name. So, it is important to give proper and understandable name for these time slices.

Same VCD to Multiple Blocks

In this case, multiple blocks are instantiations of the same master block. The same VCD file and the same VCD slice is applicable for both the blocks.

The following example describes the scenario:

```

vcd_files = [
    {'file_name': '../design_data/fsdb/block_name.fsdb',
     'instances': ['hier1/hier2/core0/block_name_wrap0',
                   'hier1/hier2/core1/block_name_wrap1'],
     'preamble' : 'label1/label4/hier1/hier2/core0/block_name/',
     'time_slices' : [
         'slice_name' : 'cores_0_and_core_1',
         'start_time' : 11086e-09,
         'stop_time' : 11089e-09,}],
],]

vcv_settings = {'annotation_type' : 'All'}
vcv = db.create_value_change_view(
    design_view=dv,
    vcd_files=vcd_files,
    settings = vcv_settings,
    tag='vcv',
    options=options)

value_change_data = [
    {'view' : vcv, 'slice_name' : 'cores_0_and_core_1'},]
scn = db.create_scenario_view(..., scenario_duration=3e-09,
                             value_change_data = value_change_data, ...)
```

Single VCD File With Multiple Time Slices to Multiple Blocks

In this case, the same VCD is applied to multiple blocks but the VCD slices are different. Therefore, you must read them separately and give different slice names.

The following example describes the scenario:

```
vcd_files = [
    {'file_name': '../design_data/fsdb/block_name.fsdb',
     'instances': ['hier1/hier2/core0/block_name'],
     'preamble': 'label1/label4/hier1/hier2/core0/block_name/',
     'time_slices': [
         {'slice_name': 'core0',
          'start_time': 11086e-09,
          'stop_time': 11089e-09}, ],
    {'file_name': '../design_data/fsdb/block_name.fsdb',
     'instances': ['hier1/hier2/core1/block_name'],
     'preamble': 'label1/label4/hier1/hier2/core1/block_name/',
     'time_slices': [
         {'slice_name': 'core1',
          'start_time': 11025e-09,
          'stop_time': 11028e-09}, ],
    },
]
vcv_settings = {'annotation_type': 'All'}
vcv = db.create_value_change_view(
    design_view=dv,
    vcd_files=vcd_files,
    settings = vcv_settings,
    tag='vcv',
    options=options)

value_change_data = [{view : vcv, slice_name : 'core0'},
                     {view : vcv, slice_name : 'core1'}]
scn_npv = db.create_no_prop_scenario_view(..., scenario_duration=3e-09,
                                           value_change_data = value_change_data, ...)
```

Mixed Usage; Multiple ScenarioViews

In this case, the user interface is flexible and multiple scenarios are possible. Two time slices each are stored for two VCD files. Then, two ScenarioViews are created by mixing VCD windows.

Note: The hierarchy/instance information is also present in the time slices.

The following example describes the scenario:

```
vcd_files = [
    {'file_name': '../design_data/fsdb/core0.fsdb',
     'instances': ['hier1/hier2/core0/block_name'],
     'preamble': 'label1/label4/hier1/hier2/core_n/block_name/',
     'time_slices': [
         {'slice_name': 'core0_high_sw',
          'start_time': 11086e-09,
          'stop_time': 11089e-09},
         {'slice_name': 'core0_low_sw',
          'start_time': 11014e-09,
          'stop_time': 11017e-09}, ],
    },
    {'file_name': '../design_data/fsdb/core1.fsdb',
     'instances': ['hier1/hier2/core1/block_name'],
     'preamble': 'label1/label4/hier1/hier2/core_n/block_name/',
     'time_slices': [
```

```

        {'slice_name' : 'core1_high_sw',
         'start_time' : 11086e-09,
         'stop_time' : 11089e-09, },
        {'slice_name' : 'core1_low_sw',
         'start_time' : 11014e-09,
         'stop_time' : 11017e-09, }],
    ],
vcv_settings = {'annotation_type' : 'All'}
vcv = db.create_value_change_view(
    design_view=dv,
    vcd_files=vcd_files,
    settings = vcv_settings,
    tag='vcv',
    options=options)

value_change_data_1 = [
    {'view' : vcv, 'slice_name' : 'core0_high_sw'},
    {'view' : vcv, 'slice_name' : 'core1_low_sw'}]
value_change_data_2 = [
    {'view' : vcv, 'slice_name' : 'core0_low_sw'},
    {'view' : vcv, 'slice_name' : 'core1_high_sw'}]
scn1 = db.create_scenario_view(... scenario_duration=3e-09,
value_change_data = value_change_data_1, ...)
scn2 = db.create_scenario_view(... scenario_duration=3e-09, ,
value_change_data = value_change_data_2, ...)

```

The argument `scn1` has `core0` in high switching VCD slice and `core1` in low switching VCD slice. The argument `scn2` has `core0` in low switching VCD slice and `core1` in high switching VCD slice.

The recommended use model is to create as many time slices as needed with different VCD files, blocks, and VCD slices. Then, create ScenarioViews by accessing the right time slice name.

7.4.3. VCD Settings in ScenarioView

The next step is to specify the type of processing required for each VCD file. This is done in ScenarioView. VCD related settings are described in this section.

VCD files are of the following types:

- RTL VCD: It only has events at sequential output points and primary inputs. There is a need to propagate events from sequential outputs to the whole design.
- Gate VCD: It typically has events for all the nets/pins in the design. They can further be of two types, non-delay VCD and delay annotated VCD.
 - Non-delay VCD: It has non-delay or unit delay VCDs, where the exact VCD event times are unreliable. You need to find out event arrival times or timing window values from elsewhere.
 - True-time VCD or delay-annotated VCD: There is a need to take exact event switching times from the VCD file.

RTL VCD

In general, RTL VCD contains only the register names, which gets mapped to the pin names of instance. Only Logic Propagation ScenarioView can process and propagate events from RTL VCD.

The following example describes the scenario:

```

value_change_data = [ {'view' : vcv, 'slice_name' : 'that_rtl_vcd'}, ]
settings = {
    'pvt' : {voltage_levels : voltage_levels},
    'event' : [
        { 'default_clock_period' : 5.5e-09, 'block_name' : '*', 'toggle_rate' :
0.2 },
        { 'default_clock_period' : 5.5e-09, 'block_name' : 'hier1*', 'toggle_rate' :
0.15 },
        { 'infer_icg_enable_signals_from_fanin_prop' : True },
        { 'object_settings' : { 'design_values' : { 'vcd_mode' : 'non_true_time', ... }} }
    ]
}
scn = db.create_scenario_view(
    timing_view=tv,
    extract_view=ev,
    options=options,
    external_parasitics=evx,
    tag = 'scn',
    value_change_data = value_change_data,
    scenario_duration=3e-09,
    settings = settings
)

```

Mostly, the default settings are applicable in RTL VCD. Propagation is ON by default. With the `event` key, you can set toggle rate for registers that are not present in the VCD. Here, a different toggle rate is given to a block. The `infer_icg_enable_signals_from_fanin_prop` key helps in better handling of Integrated Clock Gates (ICGs). By default, all ICGs are turned ON. By setting the `infer_icg_enable_signals_from_fanin_prop` argument to `True`, you can enable the flow where ON/OFF status of each ICG per cycle is deduced by analyzing the propagated signal state at ICG enable pins. You should set `vcd_mode` to `non_true_time`. RTL VCDs are output early in design cycle and do not have correct delays or event times.

Non-Delay Gate VCD

Here, the settings are exactly the same as RTL VCD. When '`vcd_mode`' is set to '`non_true_time`', the event time is calculated either by the tool's delay propagation method or by using arrival times from the STA file. Another key, `event_time_precedence`, governs the precedence. The following example describes the scenario:

```

value_change_data = [
    {'view' : vcv, 'slice_name' : 'unit_delay_gate_vcd'}, ]
settings = {
    'pvt' : {voltage_levels : voltage_levels},
    'event' : [...],
    'object_settings' : {
        'design_values' : {
            'vcd_mode' : 'non_true_time',
            'event_time_precedence' : {
                'clock_instance': ['sta'],
                'data_instance': ['sta'],
                'sequential_launch': ['sta'],
                'non_sequential_start_point': ['sta'],
                'gate_vcd_instance': ['sta']
            }
        }
    }
}

```

```

scn = db.create_scenario_view(
    timing_view=tv,
    extract_view=ev,
    options=options,
    external_parasitics=evx,
    tag = 'scn',
    value_change_data = value_change_data,
    scenario_duration=3e-09,
    settings=settings
)

```

For keys defined in `event_time_precedence`, the default value is `['propagated']`. When it is set to `['sta']`, STA is given higher priority over delay propagation. This setting is used according to the situation, such as completeness of STA file, whether the STA and VCD match in terms of operational mode, and so on.

No-propagation vectorless (NPV) ScenarioView is also used here. The settings are mostly the same as Logic Propagation ScenarioView. NPV ScenarioView cannot do delay propagation. Therefore, `event_time_precedence` key is not needed. Event times are always taken from the STA file. The following example describes the scenario:

```

value_change_data = [
    {'view' : vcv, 'slice_name' : 'unit_delay_gate_vcd'},]
settings_npv = {
    'pvt' : {voltage_levels : voltage_levels},
    'event' : {...},
    'object_settings' : {'design_values' : {'vcd_mode' : 'non_true_time', ...}},
}
scn_npv = db.create_no_prop_scenario_view(
    timing_view=tv,
    extract_view=ev,
    options=options,
    external_parasitics=evx,
    tag = 'scn',
    value_change_data = value_change_data,
    scenario_duration=3e-09,
    settings = settings_npv,
)

```

Delay-Annotated Gate VCD

In this case, `vcd_mode` key in `'true_time'` key takes care of using exact events from the VCD.

The following example describes the scenario:

```

value_change_data = [
    {'view' : vcv, 'slice_name' : 'sdf_delay_gate_vcd'},]
scn_settings = {
    'pvt' : {voltage_levels : voltage_levels},
    'event' : {...},
    'object_settings' : {'design_values' : {'vcd_mode' : 'true_time', ...}}
}
scn = db.create_scenario_view(
    timing_view=tv,
    extract_view=ev,
    options=options,
    external_parasitics=evx,
    tag = 'scn',
    value_change_data = value_change_data,
)

```

```

        scenario_duration = 3e-09,
        settings = scn_settings
    )

value_change_data = [
    {'view' : vcv, 'slice_name' : 'sdf_delay_gate_vcd'},]
settings_npv = {
    'pvt' : {voltage_levels : voltage_levels},
    'event' : {...},
    'object_settings' : {'design_values' : {'vcd_mode' : 'true_time', ...}},
}
scn_npv = db.create_no_prop_scenario_view(
    timing_view=tv,
    extract_view=ev,
    options=options,
    external_parasitics=evx,
    tag = 'scn',
    value_change_data = value_change_data,
    scenario_duration=3e-09,
    settings = settings_npv,
)

```

Mixed Delay and Non-delay VCD

In this case, different settings are applicable for different blocks. That is where the key `object_settings` becomes useful. The `object_settings` key is available for many views in SeaScape, where you can specify parameters/settings per scope (top design, block, master cell, and instance).

The following example describes the scenario:

```

value_change_data = [
    {'view' : vcv, 'slice_name' : 'cpu1_unit_delay'},
    {'view' : vcv, 'slice_name' : 'cpu2_sdf_vcd'}]
settings_dyn = {
    'pvt' : {voltage_levels : voltage_levels},
    'event' : {...},
    'object_settings' : {
        'block_values' : [
            {'patterns' : 'level1/hier2/cpu1',
             'vcd_mode' : 'non_true_time'},
            {'patterns' : 'level1/hier2/cpu2',
             'vcd_mode' : 'true_time'},],}
}
scn = db.create_scenario_view(
    timing_view=tv,
    extract_view=ev,
    options=options,
    external_parasitics=evx,
    tag = 'scn',
    value_change_data = value_change_data,
    scenario_duration=3e-09,
    settings = settings_dyn
)

```

The '`vcd_mode`' values apply to the whole design. '`patterns`' can either be string patterns or regular expressions.

Controlling Precedence of VCD Pin and Net Annotations

VCD information can be annotated at a point from either pin or net. By default, the tool gives priority to events from net when both pin and net have events. To change the default setting, use the `vcd_annotation_precedence` key as shown in the following example:

```
scn_settings = {
    'object_settings' : {
        'design_values' : {
            'vcd_mode' : 'non_true_time',
            'vcd_annotation_precedence': ['pin', 'net']}}

scn = db.create_scenario_view(..., settings = scn_settings)
```

Specifying `vcd_annotation_precedence` as either `['pin']` or `['net']` ignores all annotation from the other source. This is used only for very specific situations or experimental runs.

7.4.4. Filtering Glitches from VCD in NPV ScenarioView

By default, the tool does not remove glitches present in the input VCD file, that is, no events from the VCD are filtered out.

To remove close occurring events read in from the VCD file, set the `remove_glitches` key to `True` in the `create_no_prop_scenario_view` command settings. Based on time between such close events, the tool filters out events as shown in the following example:

```
npv_settings = {'event' : { 'glitch' : {
                    'remove_glitches': True,
                    'glitch_suppress_multiplier' : 2.0}}}
npv = db.create_no_prop_scenario_view(..., settings = npv_settings, ...)
```

The tool uses cell delay as the criterion to filter events. By default, events that occur closer than the cell delay are removed. To change this, use the `'glitch_suppress_multiplier'` key. For example, when you set `'glitch_suppress_multiplier'` to 2.0, events that are closer than two times the cell delay are filtered out.

7.4.5. Enabling Macro Non-Clock Active Pins to Generate Events

A macro instance can have non-clock **active** pins, such as asynchronous RESET, for certain modes of current. In a VCD-based logic-propagation scenario, the tool can apply the corresponding APL current mode to these non-clock active pins.

To enable this feature, specify the value of the `macro_use_all_active_pins` key as `True`. The default is `False`.

```
scn_settings = {'event' : {'macro_use_all_active_pins' : True}}
scn = db.create_scenario_view(..., settings = scn_settings)
```

When you specify the `macro_use_all_active_pins` key as `True`, the tool considers all the active pins of macro instances, including non-clock pins, to generate events.

7.5. Checking VCD Annotation

VCD annotation or coverage data is required to understand how good are VCD and name mapping. This is more important in the case of RTL VCDs, wherein you need to understand whether a good percentage of sequential points are mapped. If it is not good, then you need to get the list of uncovered instances and then look for ways to improve mapping. The API to report coverage and the API to check the VCD status for specific instance/net are described in this section.

The following concepts are covered under this section:

- [VCV Annotation Reporting](#) on page 259
- [VCD Details of an Instance or Net](#) on page 260

7.5.1. VCV Annotation Reporting

`<ValueChangeView>.get_coverage()` returns the coverage statistics for VCD. A multi-level dict is returned as output.

The following output example describes the scenario:

```
'top': {'block_total': {'covered': 16737, 'inst_count': 58918},
        'comb': {'covered': 13597,
                 'from_auto_map': 13597,
                 'from_namemap_file': 0,
                 'inst_count': 53891,
                 'uncovered': 40294},
        'icg': {'covered': 98,
                 'from_auto_map': 98,
                 'from_namemap_file': 0,
                 'inst_count': 330,
                 'uncovered': 232},
        'macro': {'covered': 8,
                  'from_auto_map': 8,
                  'from_namemap_file': 0,
                  'inst_count': 8,
                  'uncovered': 0},
        'seq': {'covered': 3034,
                 'from_auto_map': 3034,
                 'from_namemap_file': 0,
                 'inst_count': 4689,
                 'uncovered': 1655}},
```

This is for the top level design, which has name '`'top'`'. Similarly, there are keys created for all DEF blocks in the design. The keys '`from_auto_map`' and '`from_namemap_file`' refers to the way mapping is done. The key '`rtl_covered`' refers to coverage from mapping of instance pins and '`gate_covered`' refers to coverage from mapping of nets.

By default, the `<valuechangeview>.get_coverage` command reports the overall coverage of all the cell types together from the VCD input file as shown in the following example:

```
vcv.get_coverage()

'block2a': {'block_total': {'covered': 0, 'inst_count': 21, 'port_count': 6,
                           'port_covered': 1},
            'overall': {'block_total': {'covered': 29, 'inst_count': 101, 'port_count': 24,
                                       'port_covered': 3}}
```

To report specific data types, use the `include_data_types` argument with one or more of `['comb', 'flop', 'icg', 'latch', 'macro', 'pg', 'ports', 'seq']` as shown in the following example. The data type, `seq`, includes both the cell types, `flop` and `latch`.

```
vcv.get_coverage(include_data_types=['comb', 'flop', 'latch', 'ports'])

'block2a': {'block_total': {'covered': 0, 'inst_count': 21, 'port_count': 6, 'port_covered': 1}, 'ports': {'covered': 1, 'from_auto_map': 0, 'from_namemap_file': 1, 'port_count': 6, 'uncovered': 5}, 'comb': {'covered': 2, 'from_auto_map': 2, 'from_namemap_file': 0, 'inst_count': 5, 'uncovered': 3}, 'flop': {'covered': 2, 'from_auto_map': 2, 'from_namemap_file': 0, 'inst_count': 5, 'uncovered': 3}, 'latch': {'covered': 2, 'from_auto_map': 2, 'from_namemap_file': 0, 'inst_count': 5, 'uncovered': 3}, 'overall': {'block_total': {'covered': 29, 'inst_count': 101, 'port_count': 24, 'port_covered': 3}, 'ports': {'covered': 3, 'from_auto_map': 0, 'from_namemap_file': 3, 'port_count': 24, 'uncovered': 21}}, 'comb': {'covered': 2, 'from_auto_map': 2, 'from_namemap_file': 0, 'inst_count': 5, 'uncovered': 3}, 'flop': {'covered': 2, 'from_auto_map': 2, 'from_namemap_file': 0, 'inst_count': 5, 'uncovered': 3}, 'latch': {'covered': 2, 'from_auto_map': 2, 'from_namemap_file': 0, 'inst_count': 5, 'uncovered': 3}, }
```

The command can also report the coverage based on nets. To enable the feature, use the `net_coverage` argument with the `get_coverage` command. The default is `False`, that is, the coverage is based on instances. An instance is considered covered if at least one pin or net connected to the pin is covered.

7.5.2. VCD Details of an Instance or Net

For a single instance or net name, you can check for the VCD details using the `get_attributes` API. Almost all views in SeaScape have `get_attributes` API, which returns basic data stored in the view.

ValueChangeView Attributes

Syntax for `vcv.get_attributes()` returns the coverage statistics for VCD. The arguments are as follows:

Table 17: VCV Attributes

Value	Description
<code>obj</code>	This is the instance or net name. This should be instance or net object.
<code>slice_name</code>	There can be many slices in the VCV. By default, information for all slices are returned. You can limit this by specifying the slice of interest.

Value	Description
include_data_types	The list of data stored inside VCV for each instance/net are <code>['fall_toggle_count', 'initial_value', 'logic_signal', 'map_type', 'one_probability', 'rise_toggle_count']</code> . Only <code>'rise_toggle_count'</code> and <code>'fall_toggle_count'</code> are shown by default. You need to add the rest by specifying this argument.
only_data_types: include_data_types	This is to add extra data to the already existing default data. <code>only_data_types</code> restricts data in the given list.

Following are the examples to explain the scenario:

Example1

```
>>> vcv.get_attributes(Instance('register2'))
{'fsdb1': {Pin('CK'): {'fall_toggle_count': 20, 'rise_toggle_count': 20},
            Pin('D'): {'fall_toggle_count': 13, 'rise_toggle_count': 14},
            Pin('Q'): {'fall_toggle_count': 5, 'rise_toggle_count': 5},
            Pin('R'): {'fall_toggle_count': 0, 'rise_toggle_count': 0},
            Pin('SE'): {'fall_toggle_count': 0, 'rise_toggle_count': 0},
            Pin('SI'): {'fall_toggle_count': 6, 'rise_toggle_count': 7},
            Pin('clk'): {'fall_toggle_count': 20, 'rise_toggle_count': 20}},
 'fsdb_rtl': {Pin('Q'): {'fall_toggle_count': 5, 'rise_toggle_count': 5}}}
```

Only the default data types are returned here. Also, VCV data are found for the instances in two slices, `'fsdb1'` where all pins are present and `'fsdb_rtl'` where only the output pin is mapped.

Example2

```
>>>
vcv.get_attributes(Instance('buffer_instance'),only_data_types=['logic_signal'])
{'fsdb1': {Pin('i'): {'logic_signal': {'initial_value': 0, 'events': [
{'time': 3.752e-09, 'dir': 'r'}, {'time': 9.904e-09, 'dir': 'f'}]}},
            Pin('o'): {'logic_signal': {'initial_value': 1, 'events': [
{'time': 3.769e-09, 'dir': 'f'}, {'time': 9.92e-09, 'dir': 'r'}]}}}}
```

In this example, data is restricted only to `'logic_signal'`. All events for the pins are returned.

Example3

```
>>> vcv.get_attributes(Net("<buffer_instance>"),
include_data_types=['fall_toggle_count', 'initial_value', 'logic_signal',
'map_type',
'one_probability', 'rise_toggle_count'])
{'slice_chan': {
    'fall_toggle_count': 5,
    'initial_value': '1',
    'logic_signal': {'initial_value': 1, 'events': [
        {'time': 2.71e-10, 'dir': 'f'},
        {'time': 1.997e-09, 'dir': 'r'}, {'time': 3.471e-09, 'dir': 'f'}, {'time': 5.197e-09, 'dir': 'r'},
        {'time': 6.671e-09, 'dir': 'f'}, {'time': 8.397e-09, 'dir': 'r'}, {'time': 9.871e-09, 'dir': 'f'},
        {'time': 1.1597e-08, 'dir': 'r'}, {'time': 1.3071e-08, 'dir': 'f'},
        {'time': 1.4797e-08, 'dir': 'r'}]},
    'map_type': 'AutoExact',
    'one_probability': 0.4508183002471924,
    'rise_toggle_count': 5}}
```

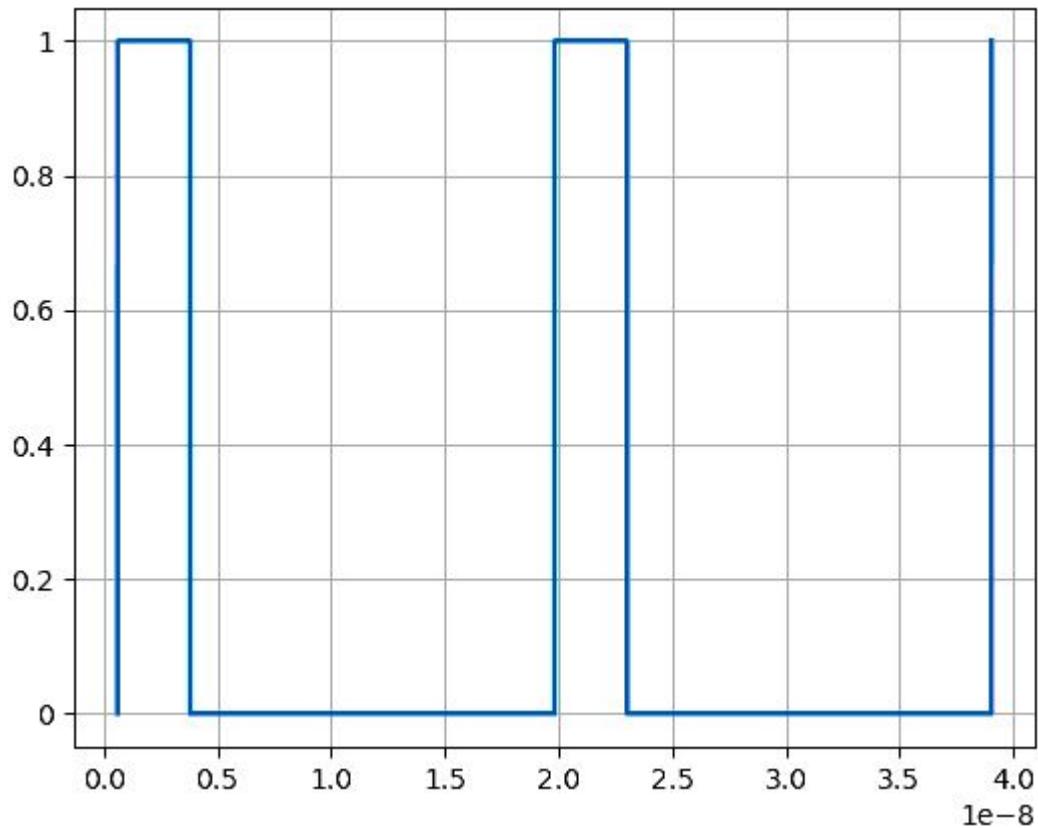
In this example, all data types are included for net objects.

- 'map_type': provides the type of mapping that is done.
- 'AutoExact': provides the exact name of net that is seen in the design.
- 'one_probability': gives the ratio of time the signal is in High/1 state.

The output logic signal can be converted to waveform type and plotted as shown in the following figure:

```
plot(vcv.get_attributes(Instance('adder'), include_data_types=['logic_signal'])  
['fsdb1'][Pin('sum')]['logic_signal'].convert_to_wf())
```

Figure 54. Plot of Logic Signal from VCV Attributes



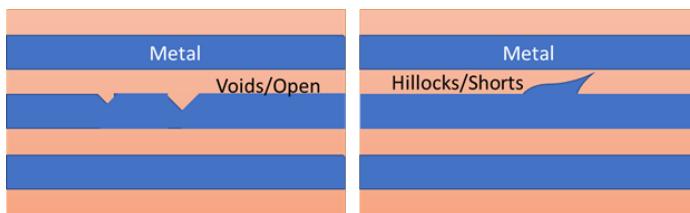
If a net-instance is not mapped to any VCD signal, `vcv.get_attributes` returns empty or null output.

8: Signal Electromigration Analysis

Electromigration (EM) is the movement of material due to the transfer of momentum between electrons and metal atoms under an applied electric field. This momentum transfer displaces metal atoms from their original positions. This effect increases with increasing current density in a wire, and at higher temperatures the momentum transfer becomes more severe.

Advanced technology node designs have high device currents, narrower wires, and increasing on-die temperatures. The reliability of interconnects and their possible degradation due to electromigration is a serious concern.

The transfer of metal ions over time can lead to either narrowing or hillocks (bumps) in wires. Narrowing of the wire can result in degradation of performance, or in extreme cases, in the complete opening of the conduction path. Widening and bumps in the wire can result in shorts to neighboring wires, especially if they are routed at the minimum pitch as shown in the following figure.



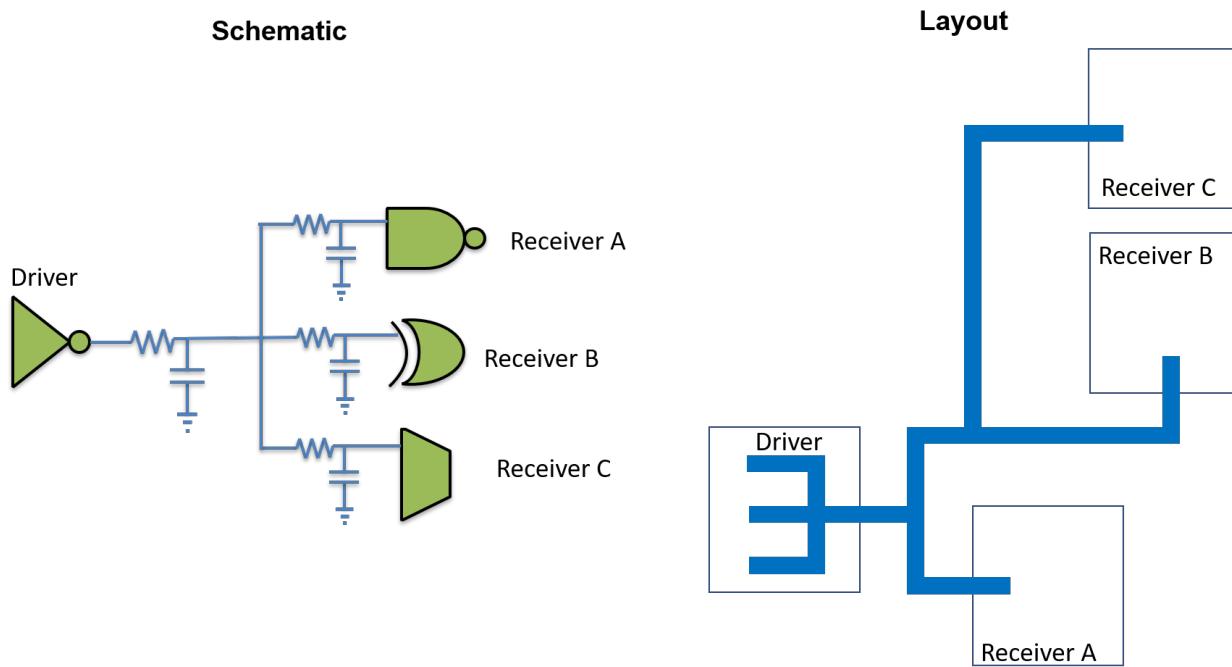
The following topics describe how to perform signal electromigration analysis by using the RedHawk-SC tool:

- [Signal Electromigration Analysis Methodology](#) on page 263
- [Signal Electromigration Flow](#) on page 264
- [Creating Timing View](#) on page 266
- [Creating SwitchingActivityView](#) on page 266
- [Creating ExtractView](#) on page 266
- [Creating Simulation View](#) on page 267
- [Creating SignalNetCurrentView](#) on page 267
- [Creating Electromigration View](#) on page 272
- [Viewing Signal Electromigration Results in GUI](#) on page 273
- [Reporting Signal Electromigration Results](#) on page 279

8.1. Signal Electromigration Analysis Methodology

The RedHawk-SC signal electromigration analysis calculates the currents flow through every signal net routing, including average current (I_{avg}), root mean square current (I_{rms}), and peak current (I_{peak}), and then compares them against their respective current limits present in the technology file.

The tool analyzes electromigration for all signal wires in a design, both inside cells (intra-cell) and between cells (inter-cell). Analysis is performed on every metal and via segment in the design starting from driver pin of the standard cell to all the receiver pins of the standard cells or primary outputs.

Figure 55. Scope of Intercell Signal EM Analysis

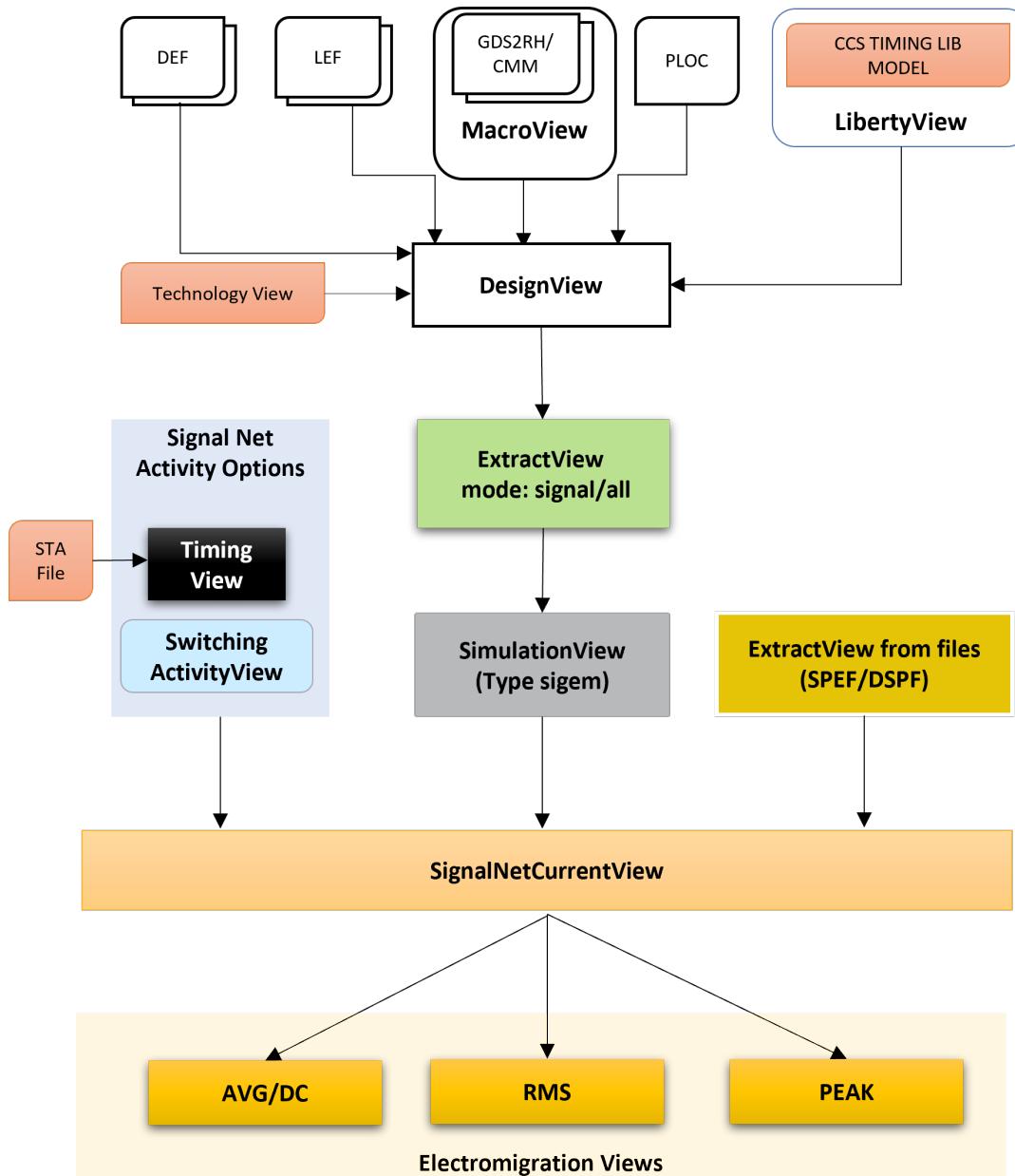
Input Data Requirements

Key data requirements for Signal electromigration analysis includes:

- Routed design netlist in DEF format
- Signal parasitic information in SPEF/DSPF format
- Timing information (frequency, slew, clock nets)
- Signal net activity information
- Technology file with specified electromigration limits
- Liberty files with composite current source (CCS) timing models. The tool supports dynamic signal electromigration flow with CCS timing only.

8.2. Signal Electromigration Flow

The following figure shows an overview of the signal electromigration flow:

Figure 56. Signal EM Flow

As shown in the flow diagram, ElectromigrationView depends on the SignalNetCurrentView where signal net currents are simulated. Before creating an ElectromigrationView, you must ensure that the SignalNetCurrentView is already created. You can specify the signal net activities either through ScenarioView or a combination of SwitchingActivityView and TimingView. SignalNetCurrentView depends on the following views:

1. TimingView
2. ExtractView (ExtractView from files, SPEF or DSPF)
3. SwitchingActivityView
4. SimulationView (Should use type='sigem')
5. The input ExtractView for SimulationView must have the `detail_extraction_net_type` argument set to `signal` or `all`.

8.3. Creating Timing View

The `create_timing_view` command creates a TimingView in the database. TimingView created for IR analysis can be reused for electromigration analysis and no specific changes are required. The created TimingView is passed to the SwitchingActivityView.

With the STA file, you can also input SDC files to the TimingView. The RedHawk-SC signal electromigration flow honors the following SDC constructs:

1. set_load

For example, the primary output load specification of the dout[0] pin:

```
set_load -pin_load 0.010 [get_ports {dout[0]}]
```

2. set_input_transition

For example, the primary input slew specification of the dly_cfg[0] pin:

```
set_input_transition -min -rise 0.300 [list [get_ports {dly_cfg[0]}]]
```

3. set_driving_cell

For example, the bufx_hvt cell that drives the primary input pin, dly_cfg[1] :

```
set_driving_cell -min -rise -clock clk -lib_cell bufx_hvt -library
my_lib -pin z -from_pin a [get_ports {dly_cfg[1]}]
```

The following example shows how to use the `sdc_files` argument to input an SDC file to the TimingView:

```
sdc_files = [{file_name' : 'design.sdc', 'instance_name' : '',
'time_unit' : '1e-12', 'capacitance_unit' : 1e-12, }]
db.create_timing_view (... , sdc_files=sdc_files, ...)
```

8.4. Creating SwitchingActivityView

The `create_switching_activity_view` command creates SwitchingActivityView in the data base. For signal electromigration Analysis, the recommended setting is default activity. See [Specifying Default Activity](#) on page 107. SwitchingActivityView is passed to SignalNetCurrentView.

8.5. Creating ExtractView

The `create_extract_view` command creates ExtractView and extracts parasitics of the design stored as DesignView.

By default, the tool performs detailed extraction (resulting in distributed RC network) of only power and ground (PG) nets. For other net types, the tool extracts only the lumped capacitance values. To perform detailed extraction of signal nets for signal electromigration analysis, set the `detail_extraction_net_type` key to `signal` under the settings dict of the `create_extract_view` command. The default is `pg`. When you set `detail_extraction_net_type` to `all`, the tool performs detailed extraction of both PG and signal nets.

8.6. Creating Simulation View

Use the `db.create_simulation_view` command to create a `SimulationView`.

By default, the tool simulates only PG nets for electromigration analysis. To simulate signal nets for signal electromigration analysis, set the `sim_type` key under the `settings` dict to `sigem`. The default is `pg`.

8.7. Creating SignalNetCurrentView

Similar to an `AnalysisView` for power and ground net simulation, the `SignalNetCurrentView` is specialized for signal net simulation, and provides current information to perform electromigration checks.

The `db.create_signal_net_current_view` command creates a `SignalNetCurrentView` and simulates signal net currents. You can specify the signal net activities through the `ScenarioView`, or a combination of `TimingView` and `SwitchingActivityView`.

The following sections how to prepare different nets for signal electromigration analysis in the `SignalNetCurrentView`:

- [Including and Excluding Nets](#) on page 268
- [Analyzing Constant Signal Nets for Electromigration](#) on page 269
- [Analyzing Clock Mesh Drivers](#) on page 269
- [Customizing Signal Net Parameters](#) on page 270
- [Specifying Current Distribution Fractions for Signal Pin](#) on page 272

8.7.1. Handling Ports and Pins

The following topics describe handling pins and ports for signal electromigration analysis:

- [Inputting Parasitics File for Primary Output Port](#) on page 267
- [Defining Driver Pin](#) on page 268

Inputting Parasitics File for Primary Output Port

You can specify a complex netlist for a primary output port by using the optional `primary_output_parasitics_file` argument.

The following example shows the format of such an input parasitic text file, `primary_output_port_parasitics`:

```
{
  "external_parasitics": [
    {
      "port_patterns": [
        "pio_plt_mem_wr_ack"
      ],
      "network": "c_load",
      "node": 1
    }
  ],
  "c_load": {
    "nodes": [
      {
        "id": 1,
        "cap": 2.5e-14
      }
    ]
  }
}
```

```

        }
    ]
}
}
```

The following example shows how to input the **primary_output_port_parasitics** file to SignalNetCurrentView:

```
av_sigem= db.create_signal_net_current_view(**arg,
primary_output_parasitics_file = primary_output_port_parasitics)
```

Defining Driver Pin

You can specify an instance pin as the driver during signal net electromigration analysis.

To specify the driver pin, use the `driver_pin` under `object_settings` as shown in the following example:

```
settings= {'object_settings': {
    'net_values': [{ 'nets' : [Net('N1')],  

    'driver_pin' : (Instance('a'),Pin('a1')) }]} }
```

You can also use this feature if net is connected to multiple INOUT pins, an output instance pin or input port, and you want to use one of the INOUT pins as driver pin instead of the output instance pin or input port.

8.7.2. Including and Excluding Nets

By default, the tool analyzes all the signal nets of the full chip. To exclude particular nets or net types from signal electromigration analysis, set the Boolean `include` key to `False`. The `include` key is available in each scope of the `object_settings` dict under the `settings` argument of the `create_signal_net_current_view` command.

The following example shows how to exclude all data signal nets of the design from electromigration analysis. You can specify net types of `clock`, `data`, or `all`.

```
object_settings = {'design_values' : {'net_type' : 'data', 'include' : False}}
```

The following example shows how to exclude all clock signal nets of the blocks that match the specified patterns from electromigration analysis.

```
object_settings = {'block_values' :  
[{'patterns' : 'u1/u2/u17', 'net_type' : 'clock', 'include' : False}]]
```

The following example shows how to exclude the clock nets of instances that match the specified patterns from electromigration analysis.

To specify nets that match a given pattern, use the wildcard (*) character. For example, to exclude all clock nets from analysis with names starting from `CPU`, use `CMRegex('CPU.*')`. All nets of the type `clock` that are instances of the cell, `ALU44` are also not analyzed.

```
object_settings = {'cell_values' :  
[{'patterns' : [CMRegex('ALU44'), CMRegex('CPU.*')],  
'net_type' : 'clock', 'include' : False}]}]
```

The following example shows how to exclude specific nets from electromigration analysis.

You can also use the `net_values` scope to override parameters of specific nets. This should be used only for small number of nets. Use STA file to override parameters for a large number of nets.

```
object_settings = {'net_values' : [{ 'nets' : [Net('signal_net_name')],  
                                    'driver_transition_time' : 100e-12,  
                                    'toggle_rate' : 0.4,  
                                    'frequency' : 1e9,  
                                    'receiver_capacitance' : 5e-15,  
                                    'voltage' : 1.2},  
                                    {'nets' : [Net('signal_net_name_2')]}], 'include' :  
                                    False}  
}  
settings = {'object_settings' : object_settings}
```

8.7.3. Analyzing Constant Signal Nets for Electromigration

Constant nets are those nets of the design that are not part of any clock domain. Pins connected to such nets are marked as CONST in the input static timing analysis (STA) file.

By default, the tool assigns zero toggle rate to such signal nets in the SwitchingActivityView. To analyze constant signal nets for electromigration, use the `constant_net_overrides` key under the `settings` argument of the `create_signal_net_current_view` command.

`constant_net_overrides` is a dict with the following keys:

- `toggle`
- `slew`
- `frequency`
- `voltage`

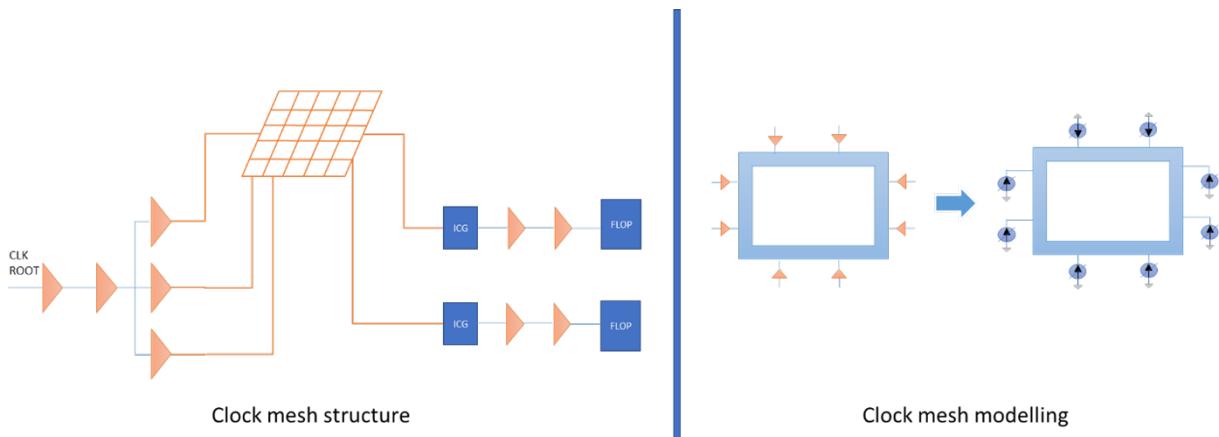
Note: To override constant nets, `toggle` and `slew` are required inputs. If you do not specify `frequency` and `voltage`, the tool assigns a default frequency of 1GHz, and a default voltage of the instance from the connected domain.

The following example assigns a toggle rate of two transitions with each transition of 50 ps, frequency of 1 GHz, and a voltage of 0.88 V to the driver instance that is marked CONST in the input STA file. With this setting, the tool analyzes the associated constant signal net for electromigration.

```
settings = {  
    'constant_net_overrides' : {'toggle': 2.0, 'slew': 5e-11, 'frequency': 1e9,  
    'voltage': 0.88}  
}  
db.create_signal_net_current_view(..., settings = settings, ...)
```

8.7.4. Analyzing Clock Mesh Drivers

Clock meshes are shorted grids of metal lines driven by multiple clock drivers. In a design, clock meshes are used to reduce clock skew and variations, such as on-chip and chip-to-chip variations.

Figure 57. Clock Mesh Structure and Mesh Modeling

By default, signal nets driven by multiple drivers are dropped from analysis. To enable signal electromigration analysis of clock meshes, perform the following steps:

1. Input the current source models of clock mesh drivers in the FSDB file format while creating the ValueChangeView.

To do so, Specify the FSDB file with the `file_name` key under the `vcv_files` argument of the `create_value_change_view` command. When the FSDB files are specified, the clock mesh signal electromigration flow is automatically enabled.

```
vcv_files = [ {'file_name' : 'path_to_clock_mesh_insts.fsdb',
  'preamble' : 'x1', 'time_slices' :
  [ {'slice_name' : 'slice0', 'start_time' : 500 e-12, 'stop_time' : 1400e-12
} ] } ]
```

```
vcv = db.create_value_change_view (design_view=dv, vcd_files=vcv_files,
rtl_namemap_files=rtl_namemap_files, tag='vcv', options=options)
```

2. Input the ValueChangeView to the `create_signal_net_current_view` command by using the `value_change_data` argument.

```
value_change_data = [ { 'view' : vcv, 'slice_name' : 'slice0' } ]
db.create_signal_net_current_view(..., value_change_data=value_change_data,
...)
```

8.7.5. Customizing Signal Net Parameters

You can override the values of the following input signal net parameters by using the `net_values` key in `object_settings` dict under the `settings` argument of the `create_signal_net_current_view` command:

- Transition time
- Toggle rate
- Frequency
- Receiver capacitance
- Voltage

`net_values` is a list of dict of values that apply to specified nets. You can define the following keys under the `net_values` key.

Key	Description
nets	A net or a list of nets to customize parameters for
driver_transition_time	Driver transition time (in seconds) override
toggle_rate	Toggle rate override value
frequency	Frequency override value (in Hertz)
receiver_capacitance	Receiver load pin capacitance override value (in Farads) The specified value is equally split among all the receivers.
voltage	Supply voltage override (in Volts) for drivers of the nets

Note: Use `net_values` only for a small number of nets. For a large number of nets, use the STA file to override parameter values.

The following example overrides the existing signal net parameters of `net1` and `net3` with the specified values.

```
settings = {
    object_setting = {'net_values' : [ {'nets' : [Net('net1'), Net('net3')],,
        'driver_transition_time' : 90e-12,
        'toggle_rate' : 0.5,
        'frequency' : 2e9,
        'receiver_capacitance' : 1e-14,
        'voltage' : 1},]}}
sigem_scv=db.create_signal_net_current_view(..., settings=settings)
```

The following example overrides the existing signal net parameters of `net1` and `net2` with one set of values, and of `net3` and `net4` with another set of values.

Because `toggle_rate` and `frequency` are not specified for `net3` and `net4`, the tool uses existing input data values for these parameters.

```
settings = {
object_settings =
{'net_values' : [ {'nets' : [Net("net1"), Net("net2")],,
        'driver_transition_time' : 100e-12,
        'toggle_rate' : 0.6,
        'frequency' : 2e9,
        'receiver_capacitance' : 5e-15,
        'voltage' : 0.9,},
    {'nets' : [Net("net3"), Net("net4")],,
        'driver_transition_time' : 300e-12,
        'receiver_capacitance' : 20e-15,
        'voltage' : 2.2,}]]}
sigem_scv=db.create_signal_net_current_view(..., settings=settings)
```

8.7.6. Specifying Current Distribution Fractions for Signal Pin

By default, the tool evenly distributes the current among the different shapes of a signal pin. This can cause unnecessary signal electromigration violations in certain cases.

For improved current distribution during signal net electromigration analysis, you should define custom phantoms while creating DesignView and specify the current fractions for the different shapes of a signal pin under the `custom_cell_phantoms` key.

In the following example, the `current_fraction` distribution is 0.2 and 0.8 for the two pin shapes of the Z pin of INV1 cell.

```
dv_settings['custom_cell_phantoms'] = { ...,
    'cells' : {
        'INV1' : {
            'coordinates' :
                [{"id":9999, 'layer':'METAL1', 'pin':'Z', 'xx':0.9, 'yy':0.7},
                 {'id':9998, 'layer':'METAL1', 'pin':'Z', 'xx':0.85, 'yy':0.45}],
            'conditions' :
                [{"signal_pin' : 'Z',
                  'signal_current_fractions' :
                      [{"coordinate_id' : 9999, 'current_fraction': 0.2},
                       {'coordinate_id' : 9998, 'current_fraction': 0.8}],}],
        },
    }
}
dv0 = db.create_design_view(..., tag= 'dv0', settings=dv_settings)
```

For the tool to honor the specified current fractions during signal net electromigration analysis, set the `short_driver_pin_phantoms` key to False while creating SignalNetCurrentView:

```
scv = db.create_signal_net_current_view
(..., tag = 'scv', settings={'short_driver_pin_phantoms': False})
```

8.8. Creating Electromigration View

The `db.create_electromigration_view` command performs electromigration checking. It can also perform failure-in-time (FIT) calculation. Results are saved in the ElectromigrationView. One of the differences between power and signal electromigration analysis is the input view used. Power electromigration analysis uses the AnalysisView whereas signal electromigration analysis requires the SignalNetCurrentView.

To specify the type of electromigration analysis, specify the `mode` key under the `settings` dict. Valid value is DC, RMS, or PEAK.

You can also specify additional controls for electromigration checks under the `settings` dict. For Example,

```
em_settings =
{'metal_line_number':9, 'target_life_time' : 'DEFAULT'}
db.create_electromigration_view(analysis_view, ..., ..., settings=em_settings,
...)
```

For more information, specify the `help` command at the tool command prompt:

```
help(SeaScapeDB.create_electromigration_view)
```

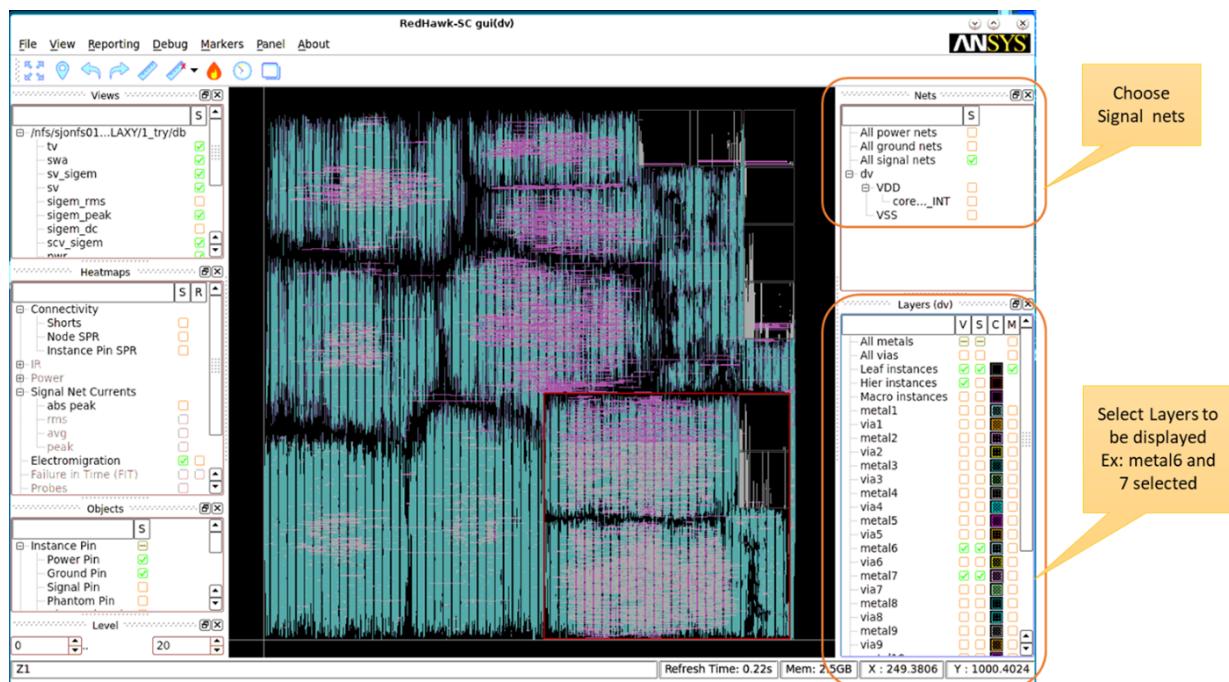
8.9. Viewing Signal Electromigration Results in GUI

When the ElectromigrationView is completed, you can view and analyze signal electromigration heatmaps in the Layout window of the RedHawk-SC console.

To enable viewing signal nets in the Layout window, do the following:

1. Select **All signal nets** in the **Nets** selector panel on the top-right of the Layout window. By default, the Layout window displays only PG nets.
2. Select the metal layers to view from the **Layers** panel at the bottom-right corner of the Layout window.

For example, to view Metal6 and Metal7 signal nets, check the box under **V** for **Metal6** and **Metal7** rows:

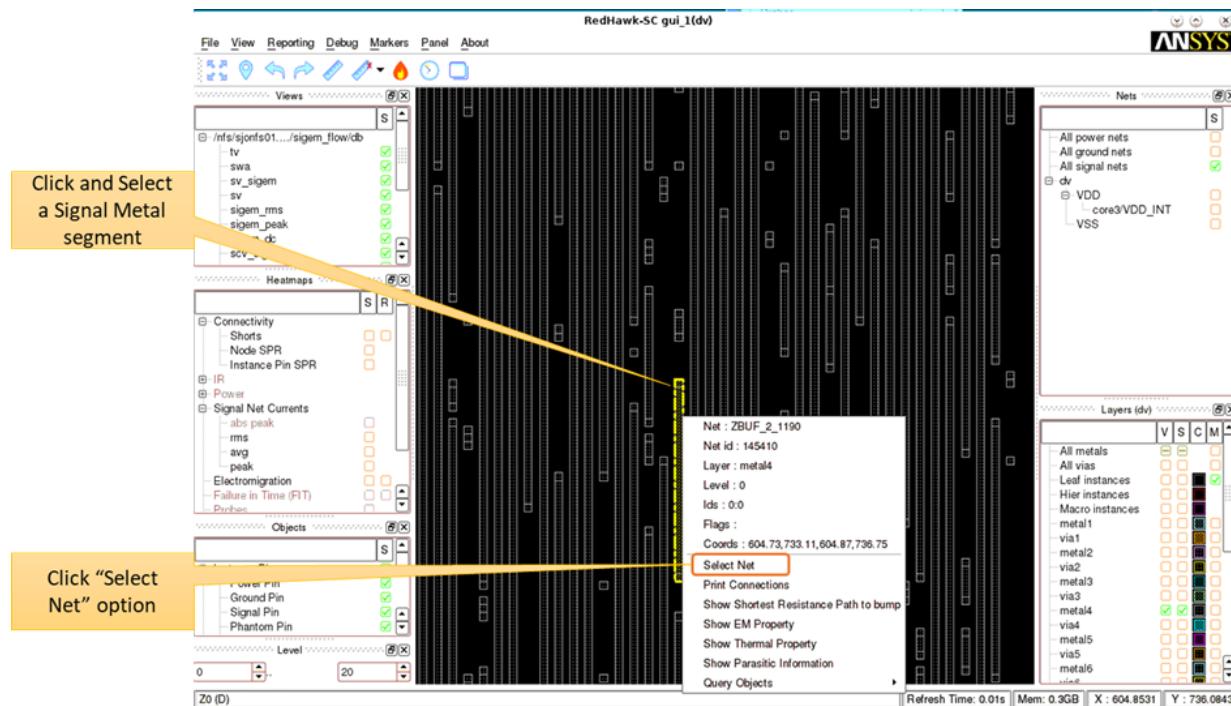


Viewing Particular Signal Nets

While debugging signal electromigration analysis results, it is useful to view only the violating nets. You can select such nets in the layout area and then enable only their display.

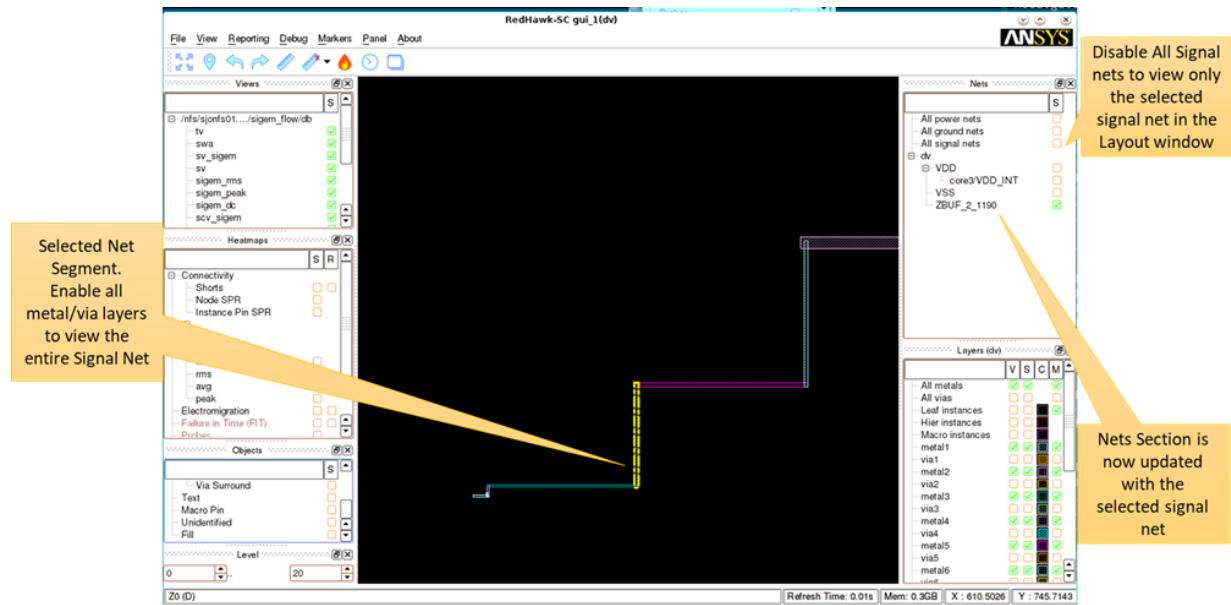
To view only a given net:

1. Click the signal net.
2. Right-click and then choose **Select Net**.

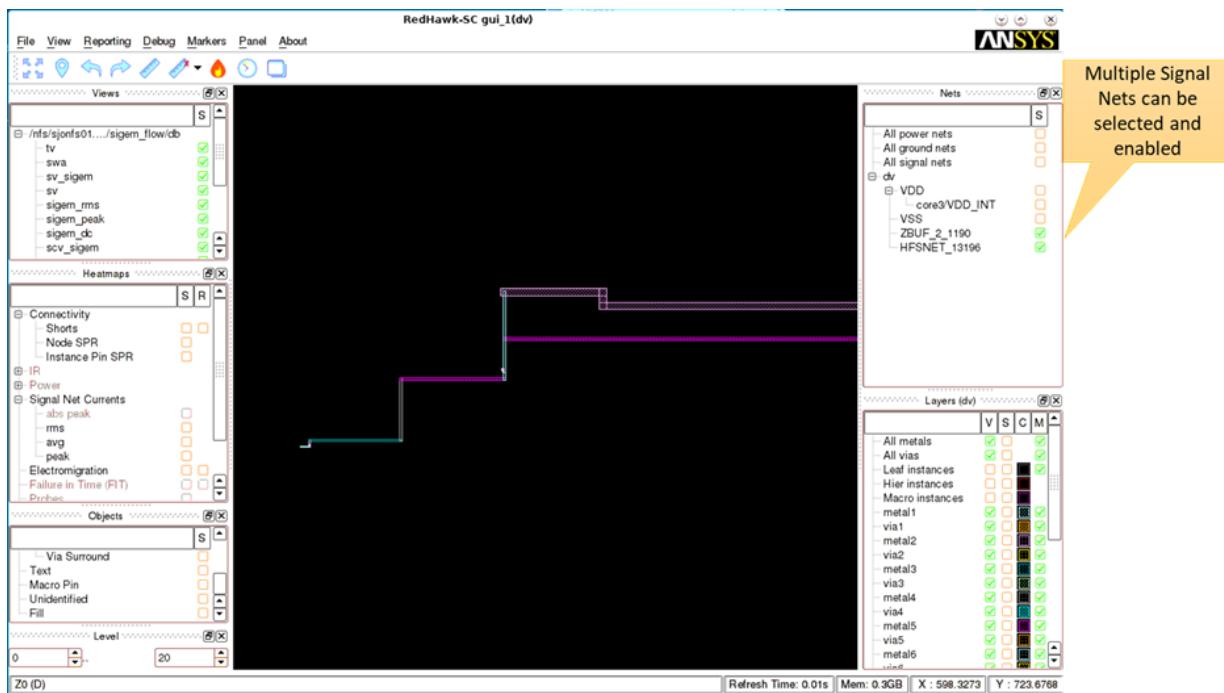


This displays the selected net name in the **Nets** panel.

- To view only the selected net, unselect all other nets in the **Nets** panel.



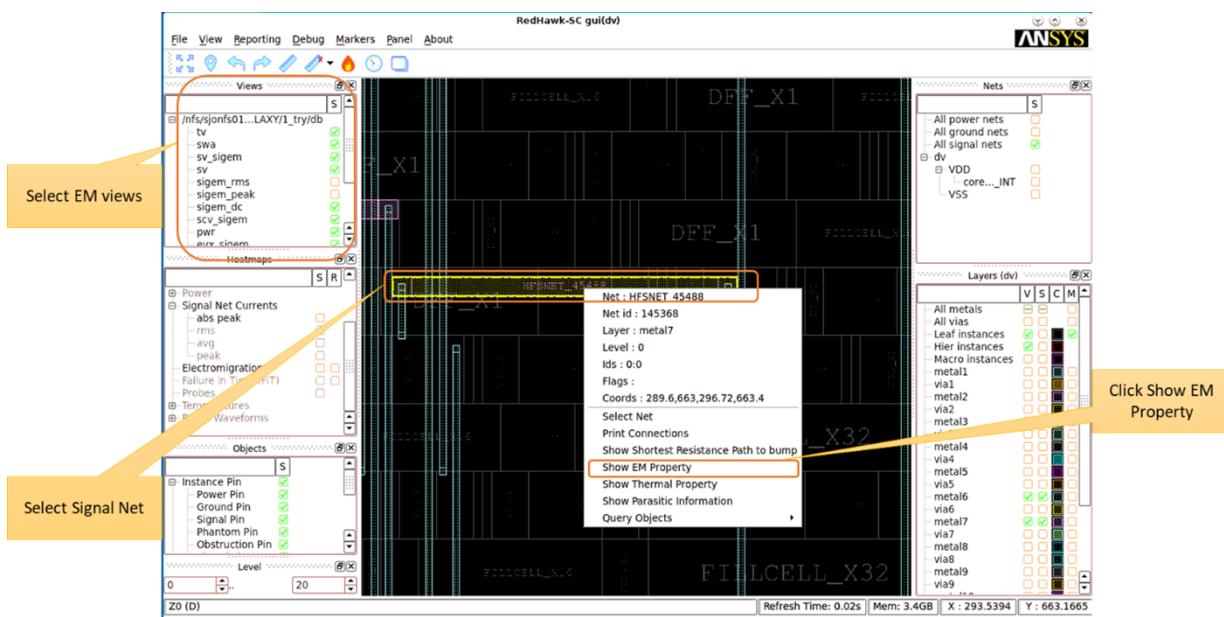
You can also enable viewing multiple signal nets, as shown in the following figure:



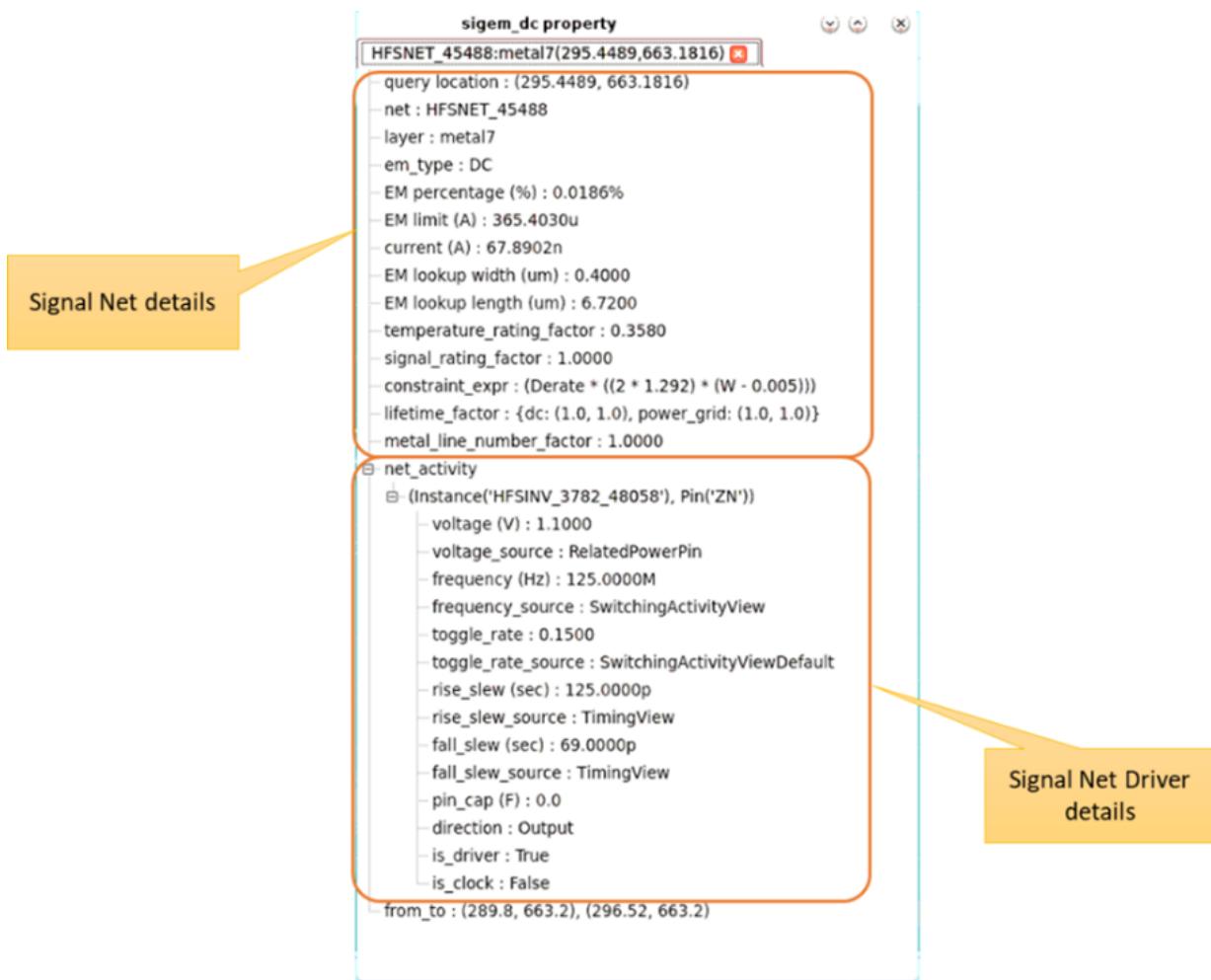
Viewing Electromigration Properties

To view the electromigration properties of a signal net, perform the following steps:

1. Select the signal net as described in [Viewing Particular Signal Nets](#) on page 273. Ensure that for the displayed net, the box under the **S** (Select) column head is checked in the **Layers** panel.
2. Right-click the net wire segment and select **Show EM Property**.



This opens the property window, as shown in the following example:



The property window displays query location details, net name, layer, and type of EM. It also displays the EM parameters, such as, current, limit, and violation percentage. All the factors used for EM limit computation are also displayed.

The **net_activity** section displays the driver information of the selected net including the driver pin toggle rate, voltage, frequency, slews, and direction.

For a database with multiple EM views or types, the following **Select View** window opens.



You can now choose from the available set of EM views. The signal EM views in the drop-down list are already enabled in the **Views** selector panel in the layout window.

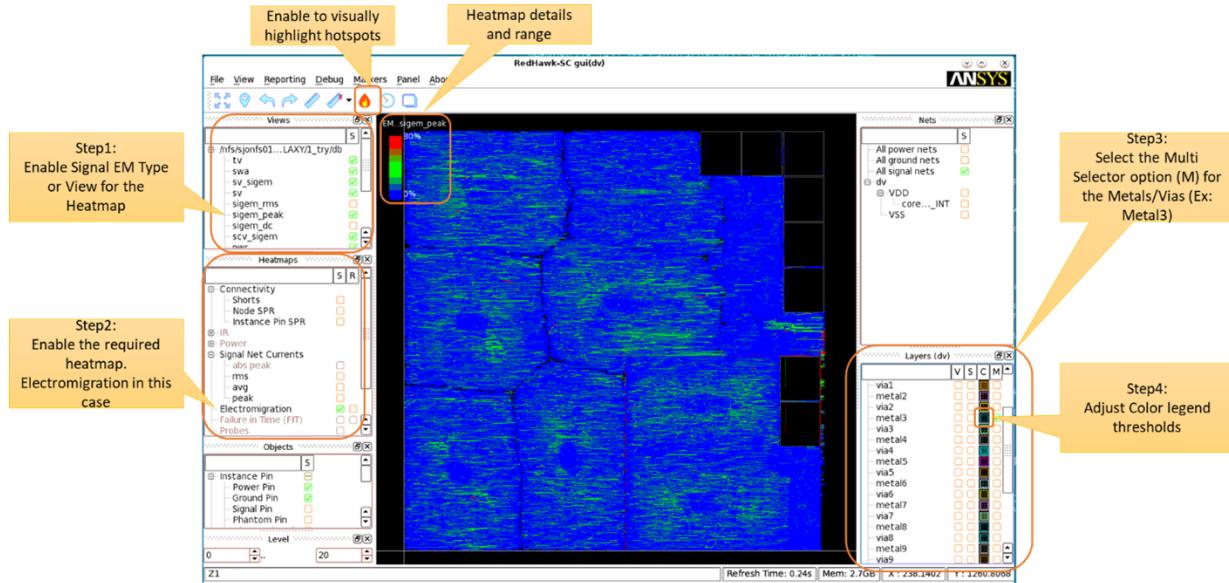
Viewing Electromigration Heatmaps

RedHawk-SC signal electromigration heatmaps are available for metals and vias for both currents and violations.

To view the signal electromigration heatmap in the Layout window, follow these steps:

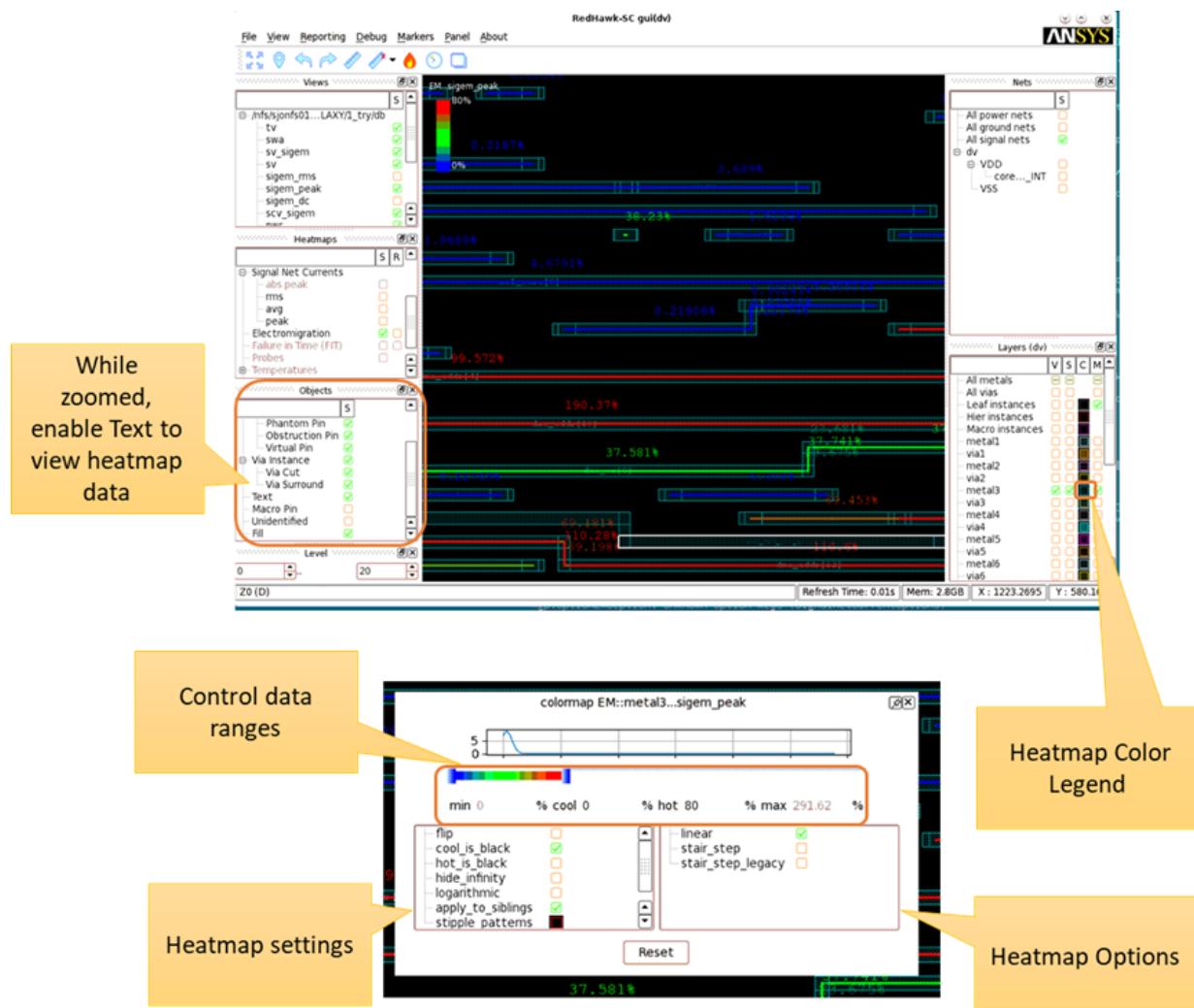
1. Select the ElectromigrationView name under **Views**. This step enables the electromigration heatmap.
2. Select the heatmap name under the **S** check box in **Heatmaps**.
3. Select the metal or the via layer under the multiselector (**M**) check box in **Layers**.

For example, the following figure shows how to enable signal electromigration **PEAK** violation heatmap for Metal 3.



4. (Optional) By default, the heatmap color scheme is selected in range from 0-100% across all layers for the electromigration heatmap.

To customize the range for each layer individually,



Note: By default, only the absolute peak current per signal net (**abs peak**) is available under **Signal Net Currents** in **Heatmaps**. The absolute peak current has the highest peak among the average, RMS, and peak currents.

To enable viewing the average, RMS, and peak current heatmaps, use the option, `options['signal_net_current_options'].current_heatmap_modes = ['rms', 'avg', 'peak']`, with the `create_signal_net_current_view` command. For example,

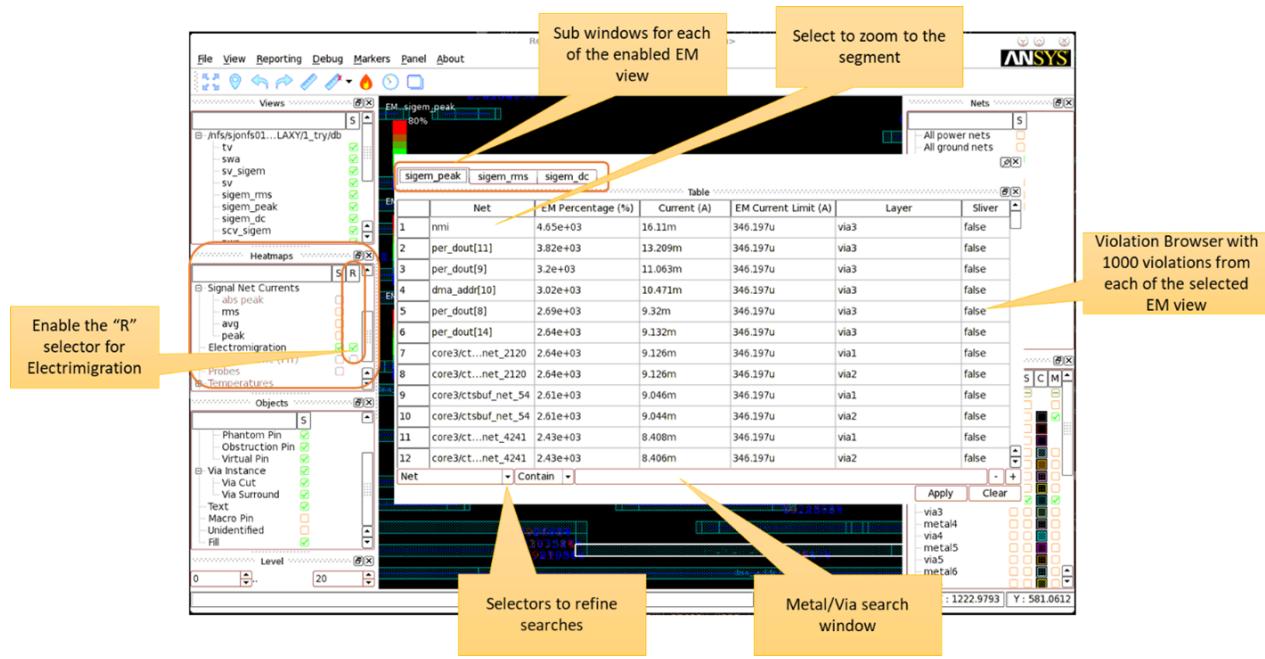
```
options = get_default_options()
options['signal_net_current_options'].current_heatmap_modes = ['rms', 'avg', 'peak']
db.create_signal_net_current_view( ..., options=options, ... )
```

This displays the **rms**, **avg**, and **peak** rows under **Signal Net Currents** in **Heatmaps**.

EM Violation Browser

To access the EM violation browser in the Layout window, select the check box under the **R** column for the electromigration heatmap from the **Heatmaps** panel.

The EM violation browser lists the top 1000 violations from each of the enabled ElectromigrationViews. You can select the nets or wire-segments and zoom in to view the violation.



8.10. Reporting Signal Electromigration Results

Several commands are available to report signal electromigration analysis results. The following sections describe the different methods to report electromigration analysis results:

- [Reporting Electromigration Violations in Metal Layers](#) on page 279
- [Reporting Electromigration Violations in Vias](#) on page 281
- [Reporting Width Guidance of Signal Nets](#) on page 283
- [Reporting Dropped Nets](#) on page 285
- [Reporting Dirty Nets](#) on page 286
- [Reporting Signal Net Attributes](#) on page 287
- [Querying Signal Electromigration Data](#) on page 287

Reporting Electromigration Violations in Metal Layers

To output a formatted electromigration report for metal layers, use the `emir_reports.write_em_metal_report` command:

```
emir_reports.write_em_metal_report
(electromigration_view, output_file='./metal.em.rpt',
 ignore_nets_file='ignore_nets.list', columns=None, sort=True,
 sort_order='descending', sort_columns=['violation'], formats=None,
 header=None, footer=None, max_lines=5000, nets=[], nets_regex=None,
 layers=None, layers_regex=None, em_range=None, ignore_sliver=True)
```

The following table provides details of the arguments:

Argument	Description
electromigration_view	The electromigration analysis view from which to write the report. (type=ElectromigrationView, required=True)
output_file	The path to the output file for the report. (type=str, default_value='./metal.em.rpt')
ignore_nets_file	The path to file with net names that must be excluded from the generate report. (type=str, default_value='ignore_nets.list')
columns	A list of columns to be included in the report. May be one or more of 'metal_line_number_factor', 'layer', 'applied_duty_ratio', 'lifetime_factor', 'violation', 'vomin', 'power_grid', 'em_type', 'constraint_expr', 'net', 'from', 'constraint', 'seg_length', 'current', 'to', 'length', 'si_width', 'applied_pulse_width', 'pulse_width', 'duty_ratio', 'width', 'status', 'current_direction'. The output file will include only the columns specified in the order specified. (type=list, default_value=None)
sort	Set to True to sort the output file or False for no sort. (type=bool, default_value=True)
sort_order	Set to either 'descending' or 'ascending' to select the desired sort order. (type=str, default_value='descending')
sort_columns	A list of column(s) that are used to sort the report (primary keys first). (type=list, default_value=['violation'])
formats	A format() string that is used to format each line of the report. Set to None for default output. (type=str, default_value=None)
header	A string that is used as the header in the output file. If None is specified, the columns are used to create a default header. (type=str, default_value=None)
footer	A string that is used as the footer in the output file. If None is specified, the footer is omitted. (type=str, default_value=None)
max_lines	The maximum number of lines to include in the report (defaults to write all lines). (type=int, default_value=5000)
nets	A list of net names to include in the report. If None is specified, defaults to all nets. (type=list, default_value=[])
nets_regex	A regular expression (CMRegex) that specifies nets to be selected for the report (type=CMRegex, default_value=None)
layers	A list of metal layer names to include in the report. If None is specified, defaults to all metal layers. (type=list, default_value=None)
layers_regex	A regular expression (CMRegex) that specifies metals layers to be selected for the report (type=CMRegex, default_value=None)
em_range	A tuple containing the (low, high) EM range (in percent) to include in the report or a single float that specifies the lower EM threshold (also in percent) . (type=object, default_value=None)
ignore_sliver	Ignore sliver EM violations. (type=bool, default_value=True)

You can specify report formatting options with the `formats` argument. The default format string is as follows:

```
'{:^10} {:^22} {:^22} {:^10.4g} {:^10.4g} {:^10.4g} {:^10.4g}
{:^10.4g} {:^6} {:<}'
```

By default, the `write_em_metal_report` command reports the following columns: `layer`, `from`, `to`, `length`, `width`, `current`, `constraint`, `violation`, `status`, and `net`. To customize the columns to report, use the `columns` argument.

For example, to generate a custom report without the `length` column, you must specify the `header` and the `formats` arguments:

```
columns = ['layer', 'from', 'to', 'width', 'current', 'constraint',
           'violation', 'status', 'net']
formats = '{:^10} {:^22} {:^22} {:^10.4g} {:^10.4g} {:^10.4g}
           {:^10.4g} {:^6} {:<}'
```

```
header = '{:^10} {:^22} {:^22} {:^10} {:^10} {:^10}
           {:^10} {:^6}\n'.format(*columns)

emir_reports.write_em_metal_report(em, output_file='./custom_output',
                                   columns=columns, header=header, formats=formats)
```

The following is an example of signal electromigration metal report for peak currents. For more information about electromigration metal reports, see [DC Electromigration Metal Report](#) on page 140.

#	<code>em_type = PEAK</code> , <code>ignore_sliver = True</code> , 'length' column is blank	<code>layer</code>	<code>from</code>	<code>to</code>	<code>length</code>	<code>width</code>	<code>current</code>	<code>constraint</code>	<code>violation</code>	<code>status</code>	<code>net</code>
#			(u)	(u)	(u)	(u)	(A)	(A)	(%)		
meta13	(1224.44, 628.32)		(1226.795, 628.32)		2.355	0.07	0.01613	0.005531	291.7	FAIL	nmi
meta14	(1224.44, 627.76)		(1224.44, 628.32)		0.56	0.14	0.01611	0.005656	284.8	FAIL	nmi
meta15	(1221.36, 627.76)		(1224.44, 627.76)		3.08	0.14	0.01609	0.006562	245.2	FAIL	nmi
meta16	(1221.36, 628.0)		(1221.36, 627.76)		0.24	0.14	0.01607	0.006562	244.9	FAIL	nmi
meta13	(1224.16, 626.92)		(1226.795, 626.92)		2.635	0.07	0.01322	0.005531	239.1	FAIL	per_dout[11]
meta14	(1224.16, 625.52)		(1224.16, 626.92)		1.4	0.14	0.0132	0.005656	233.3	FAIL	per_dout[11]
meta13	(1226.76, 641.2)		(1226.795, 641.2)		0.035	0.07	0.01108	0.005531	200.4	FAIL	per_dout[9]
meta13	(1220.8, 579.88)		(1226.795, 579.88)		5.995	0.07	0.0105	0.005531	189.9	FAIL	dma_addr[10]
meta13	(1222.76, 628.705)		(1222.76, 628.74)		7.86	0.105	0.007521	0.004322	174	FAIL	lfixt_clk
meta15	(1225.84, 641.2)		(1226.76, 641.2)		0.92	0.14	0.01108	0.006562	168.9	FAIL	per_dout[9]
meta13	(1224.44, 626.64)		(1226.795, 626.64)		2.355	0.07	0.009325	0.005531	168.6	FAIL	per_dout[8]
meta13	(1223.6, 639.1)		(1223.6, 639.135)		9.18	0.105	0.007224	0.004322	167.1	FAIL	irq[5]
meta13	(1223.32, 627.76)		(1226.795, 627.76)		3.475	0.07	0.009145	0.005531	165.4	FAIL	per_dout[14]

Reporting Electromigration Violations in Vias

To output a formatted electromigration report for via layers, use the `emir_reports.write_em_via_report` command:

```
emir_reports.write_em_via_report(electromigration_view,
                                 output_file='./via.em.rpt', ignore_nets_file='ignore_nets.list',
                                 columns=None, sort=True, sort_order='descending', sort_columns=['violation'],
                                 formats=None, header=None, footer=None, max_lines=5000, nets=[],
                                 nets_regex=None, layers=None, layers_regex=None, em_range=None,
                                 ignore_sliver=True)
```

Value	Description
<code>electromigration_view</code>	The electromigration analysis view from which to write the report. (type=ElectromigrationView, required=True)
<code>output_file</code>	The path to the output file for the report. (type=str, default_value='./via.em.rpt')

Value	Description
ignore_nets_file	The path to file with net names that must be excluded from the generate report. (type=str, default_value='ignore_nets.list')[MH1] [SP2] [SP3]
columns	A list of columns to included in the report. May be one or more of 'loc_x', 'loc_y', 'num_cuts', 'metal_line_number_factor', 'layer', 'from', 'Wb', 'constraint', 'violation', 'Lb', 'current', 'to', 'Wu', 'Lu', 'power_grid', 'via_length', 'em_type', 'via_width', 'net', 'is_upstream', 'constraint_expr', 'lifetime_factor' or 'status', 'current_direction'. The output file will include only the columns specified in the order specified. (type=list, default_value=None)
sort	Set to True to sort the output file or False for no sort. (type=bool, default_value=True)
sort_order	Set to either 'descending' or 'ascending' to select the desired sort order. (type=str, default_value='descending')
sort_columns	A list of column(s) that are used to sort the report (primary keys first). (type=list, default_value=['violation'])
formats	A format() string that is used to format each line of the report. Set to None for default output. (type=str, default_value=None)
header	A string that is used as the header in the output file. If None is specified, the columns are used to create a default header. (type=str, default_value=None)
footer	A string that is used as the footer in the output file. If None is specified, the footer is omitted. (type=str, default_value=None)
max_lines	The maximum number of lines to include in the report (defaults to write all lines). (type=int, default_value=5000)
nets	A list of net names to include in the report. If None is specified, defaults to all nets. (type=list, default_value=[])
nets_regex	A regular expression (CMRegex) that specifies nets to be selected for the report (type=CMRegex, default_value=None)
layers	A list of via layer names to include in the report. If None is specified, defaults to all via layers. (type=list, default_value=None)
layers_regex	A regular expression (CMRegex) that specifies via layers to be selected for the report (type=CMRegex, default_value=None)

Value	Description
em_range	A tuple containing the (low, high) EM range (in percent) to include in the report or a single float that specifies the lower EM threshold (also in percent). (type=object, default_value=None)
ignore_sliver	Ignore sliver EM violations. (type=bool, default_value=True)

The following is an example of signal electromigration via report. For more information about electromigration via reports, see [DC Electromigration Metal Report](#) on page 140.

```
# em_type = DC, ignore_sliver = True
# layer      loc_x      loc_y      via_length      via_width      current      constraint      violation      status      net
#   (u)        (u)        (u)        (u)           (A)          (u)           (%) 
via1    443.27    659.785     0.07       0.07    2.1e-05    0.002082    1.008     PASS    ctsbuf_net_88
via2    443.27    659.82      0.07       0.07   2.099e-05    0.002082    1.008     PASS    ctsbuf_net_88
via1    909.365    31.465     0.07       0.07   2.016e-05    0.002082    0.9684    PASS    core3/ctdbuf_net_3635
via2    909.365    30.905     0.07       0.07   2.011e-05    0.002082    0.9657    PASS    core3/ctdbuf_net_3635
via3    904.585    30.905     0.07       0.07   1.984e-05    0.002082    0.9528    PASS    core3/ctdbuf_net_3635
via3    904.585    31.465     0.07       0.07   1.968e-05    0.002082    0.9449    PASS    core3/ctdbuf_net_3635
via1    511.48     713.44      0.07       0.07   1.794e-05    0.002082    0.8616    PASS    ctsbuf_net_3030
via1    535.61     1001.385    0.07       0.07   1.791e-05    0.002082    0.86       PASS    ctsbuf_net_4141
via2    535.61     1001.7      0.07       0.07   1.791e-05    0.002082    0.8599    PASS    ctsbuf_net_4141
via2    512.05     714.84      0.07       0.07   1.788e-05    0.002082    0.8586    PASS    ctsbuf_net_3030
via3    511.0      714.84      0.07       0.07   1.785e-05    0.002082    0.8571    PASS    ctsbuf_net_3030
via1    98.8       264.985     0.07       0.07   1.771e-05    0.002082    0.8505    PASS    ctsbuf_net_247247
via2    98.8       265.3       0.07       0.07   1.771e-05    0.002082    0.8504    PASS    ctsbuf_net_247247
```

Reporting Width Guidance of Signal Nets

For width guidance after signal electromigration analysis, use the following commands from the `emir_reports` module.

To generate the data, use the following command:

```
emir_reports.get_em_width_adjustment_guidance(electromigration_view,
                                              layers, nets=None, em_threshold=100.0, relative_tolerance=5.0)
```

Argument	Description
electromigration_view	Electromigration view of the Design (type=ElectromigrationView, required=True)
layers	Only consider edges for the specified metal layer(s) (type=list, required=True)
nets	nets : Only consider edges for the specified net(s) (type=list, default_value=None)
em_threshold	Only consider edges with higher EM violation value than this (type=float, default_value=100.0)
relative_tolerance	A specified percentage value to find potential EM results closer to em_threshold, based on which the relative change should be better less than this, but not a must (type=float, default_value=5.0)

To write out the report, use the following command:

```
emir_reports.report_em_width_adjustment_guidance
(electromigration_view, input_chunked_data,
```

```
output_file='./em_width_adjustment_guidance.rpt', sort_order='descending',
sort_columns=['originalViolation'], max_lines=5000)
```

Argument	Description
electromigration_view	Electromigration view of the Design (type=ElectromigrationView, required=True)
input_chunked_data	A ChunkedData dict holding EM width adjustment guidance details info for all processed edges (type=dict, required=True)
output_file	The path to the output file for the report (type=str, default_value='./em_width_adjustment_guidance.rpt')
sort_order	Set to either 'descending' or 'ascending' to select the desired sort order. (type=str, default_value='descending')
sort_columns	A list of columns(s) that are used to sort the report (primary keys first). (type=list, default_value=['originalViolation'])
max_lines	The maximum number of lines to include in the report (default to write all lines). (type=int, default_value=5000)

The report includes width suggestions to fix electromigration violations on the metals segment and also potential violation percentage with the suggested width. The metal width recommendation is not DRC aware and is not guaranteed to fix the violation by considering secondary effects.

Note: Changing the metal-wire width might cause changes to the extracted parasitic netlist, including resistance, capacitance, and inductance. As a result, simulation could be impacted and the current flowing through the related wire segment might also change. Those are considered as secondary effects and are not taken into account when making resizing recommendations.

The following example obtains the width guidance of layers M1 to M10.

```
data = emir_reports.get_em_width_adjustment_guidance
(em_rms, layers=['M0', 'M1', 'M2', 'M3', 'M4', 'M5',
'M6', 'M7', 'M8', 'M9', 'M10'], em_threshold=100)
emir_reports.report_em_width_adjustment_guidance(em_rms,
data, output_file='./em_width_adjustment_guidance.rpt')
```

The following example shows the generated report.

from_coord (u)	to_coord (u)	original_width (u)	suggested_width (u)	original_violation (%)	potential_violation (%)	layer	em_type	net
METAL	DC	VSS						
(4.8, 6.1)	(5.1, 9.1)	0.02	0.03	125.5	99.87	M2	RMS	n1
(6.9, 6.7)	(7.2, 8.7)	0.02	0.03	118	93.87	M2	RMS	n2
(6.4, 9.7)	(6.7, 9.9)	0.02	0.03	115.6	91.97	M2	RMS	n3
(5.6, 7.1)	(5.9, 9.1)	0.02	0.03	110.2	97.56	M2	RMS	n4

Reporting Dropped Nets

Nets that cannot be analyzed due to missing inputs or broken netlists are called dropped nets.

The `create_electromigration_view` command drops these nets from signal electromigration analysis, and categorizes them in the output report according to the missing data with the following tags:

Value	Description
ConstNet	Nets tagged as CONST in STA file
DriverReceiverShort	Net has driver and receiver shorted with a single node
MultiDriver	There are multiple drivers for this net
NoCapacitor	There is no valid cap on this net
NoDriver	Missing DEF driver
NoNode	There is no valid node on this net
NoReceiver	There is no valid receiver connected to this net
ReceiverDisconnected	All the receivers are disconnected from driver pin
UnknownDrop	Net is dropped with unknown reason

To report dropped nets, you can use one of the following commands:

- The `get_sigem_dropped_net_report` and `write_sigem_dropped_net_report` commands from the `emir_reports` module, such as:

```
d_dropped_nets = emir_reports.get_sigem_dropped_net_report(av_sigem)
emir_reports.write_sigem_dropped_net_report(av_sigem, "./dropped_nets.rpt",
d_dropped_nets)
```

The following is a dropped net report example.

```
#NN  (NoNode): There is no valid node on this net
#NC  (NoCapacitor): There is no valid cap on this net
#DRS (DriverReceiverShort): Net has driver and receiver shorted with a single node
#MD  (MultiDriver): There are multiple driver for this net
#RD  (ReceiverDisconnected): All of the receivers are disconnected from driver pin
#UD  (UnknownDrop): Net is dropped with unknown reason
#CN  (ConstNet): Tagged as CONST in STA
#NR  (NoReceiver): There is no valid receiver connected to this net
#ND  (NoDriver): Missing DEF driver

#net_name      tag
aclk_en_mcout ['CN']
core0.openMSP430_inst0.multiplier_0.sumext[15]  ['NR', 'RD']
core0.openMSP430_inst1.multiplier_0.sumext[15]  ['NR', 'RD']
core0.regfile_data_memory._26709_  ['NR', 'RD']
core0.regfile_data_memory._26710_  ['NR', 'RD']
core0.regfile_data_memory._26711_  ['NR', 'RD']
```

- The `get_dropped_nets` command that returns the list of dropped nets with their respective tags in the dict format from a `SignalNetCurrentView`, such as:

```
sigem_scv.get_dropped_nets()
```

Reporting Dirty Nets

Nets with incomplete data for signal electromigration analysis are called dirty nets. The analysis results for these nets might not be reliable.

The `create_electromigration_view` command analyzes dirty nets by using default values for the missing data. For example, the tool uses default values for missing frequency and transition times of nets. The tool reports such nets as dirty nets so that you can review the dirty nets.

The following tags are available in the dirty net report:

Table 18: Dirty Net Report

Value	Description
DriverNoValidVoltage	There is no valid voltage on driver pin (using default parameters)
HasLogicalConnNoPhyConnPin	There are pins have logical connections but no physical connections on this net
HasNoDirectionPin	There are pins have no direction defined on this net
HasPhyConnNoLogicalConnPin	There are pins have physical connections but no logical connections on this net
MissingFreq	No frequency coverage (using default parameters)
MissingSlew	No slew coverage (using default parameters)
MissingSpfCap	No capacitance information from SPEF
MissingSpfDriver	No driver from SPEF which is consistent with DEF
UnknowDirty	Net is set as dirty with unknown reason
WireDisconnected	There are wires/vias disconnected with driver on this net, will ignore these broken areas when simulation
MissingDriverModel	Indicates a missing composite current source (CCS) timing Liberty model. A signal net is marked as a dirty net with this label when all CCS timing arcs are missing in the specified Liberty files for the cell of the net's driver instance. For example, <code>SignalNetCurrentView.is_net_dirty(<net_name>) ['MissingDriverModel']</code>

The tool uses the following default values for dirty nets (nets missing input data):

Table 19: Default Values for Dirty Nets

Value	Description
Frequency	1 GHz
Voltage	1 V
Toggle Rate	0.2
Slew	50 ps

You can overwrite these default values while creating the SignalNetCurrentView. See [Analyzing Constant Signal Nets for Electromigration](#) on page 269. For more information, contact Ansys support.

To report dirty nets, you can use one of the following commands:

- The `get_sigem_dirty_net_report` and `write_sigem_dirty_net_report` commands from the `emir_reports` module, such as:

```
d_dirty_net = emir_reports.get_sigem_dirty_net_report(av_sigem)
emir_reports.write_sigem_dirty_net_report(av_sigem, "./dirty_nets.rpt",
d_dirty_net)
```

The following is a dirty net report example.

```
#NPC (HasLogicalNoPhysicalConnectionPins): There are pins on this net that have no physical connection but logic connections only
#UDP (HasUndefinedDirectionPins): There are pins on this net that have no direction defined
#NLC (HasPhysicalNoLogicalConnectionPins): There are pins on this net that have no logic connection but physical connections only
#MS (MissingSlew): No slew coverage (Using default parameters)
#NVV (DriverNoValidVoltage): There is no valid voltage on driver pin (Using default parameters)
#MSC (MissingSpfCap): No capacitance information from SPEF
#MF (MissingFreq): No frequency coverage (Using default parameters)
#UD (UnknownDirty): Net is set as dirty with unknown reason
#WD (WireDisconnected): There are wires/vias disconnected with driver on this net, will ignore these broken areas when simulation
#MSD (MissingSpfDriver): No driver from SPEF which is consistent with DEF

#net_name tag
HFSNET_45156 ['MSC']
HFSNET_5276 ['NF', 'MS']
HFSNET_5876 ['NF', 'MS']
ZBUF_1003_4208 ['MSC', 'NF', 'MS']
ZBUF_1011_2883 ['MSC']
HFSNET_44851 ['MSC']
ZBUF_1018_4122 ['MSC', 'NF', 'MS']
```

- The `get_dirty_nets` command that returns the list of dirty nets with their respective tags in the dict format from a SignalNetCurrentView, such as:

```
sigem_scv.get_dirty_nets()
```

Reporting Signal Net Attributes

After the completion of signal electromigration analysis, you can generate a signal net information file containing all the relevant signal net data for electromigration analysis. To report the attributes of signal nets that are stored in the SignalNetCurrentView, use the `write_sigem_net_attributes` command from the `emir_reports` module as shown:

```
d_net_attrs = emir_reports.get_sigem_net_attributes(av_sigem)
emir_reports.write_sigem_net_attributes(av_sigem, "./sigem.info", d_net_attrs)
```

Reported attributes include net name and type, driver voltage, frequency, instance and cell name, connected pin, toggle rate, transition times, capacitance, driver details, type, clock domain names, and PG domain names.

Querying Signal Electromigration Data

The RedHawk-SC syntax includes various commands to query signal electromigration information from available databases. To see the list of commands available to query a particular view, use the following command:

```
help(<view_name>)
```

For example, `help(scv_sigem)` where `scv_sigem` is the SignalNetCurrentView name, returns the following output:

```
get_current_heatmap(mode='default')
```

Returns a heatmap of current for the signal mode.

```
get_current_stats(net, layer, coord)
```

Returns current statistics at given location on specific layer for specific net.

```
get_dirty_nets(limit=100)
```

Return the list of dirty nets which will still be analyzed.

```
get_dirty_net_identifier()
```

Return the detail description of dirty nets' tags.

```
get_disconnected_nets(limit=100)
```

Get the list of nets which have disconnected wire.

```
get_drop_net_identifier()
```

Return the detail description of drop nets' tags.

```
get_dropped_nets(limit=100)
```

Return the list of nets which been dropped for analysis.

```
get_edge_current_stats(circuit_edge)
```

Returns current statistics for a circuit edge.

```
get_info()
```

Get summary information about this view.

```
get_instance_pin_activity(instance, pin=None)
```

Return activity information for given instance, (and pin).

```
get_net_activity(net, limit=None)
```

Return activity information calculated for given net.

```
get_net_arcs(net)
```

Get parameters of a given net.

```
get_net_disconnected_nodes(net)
```

Get the disconnected nodes' id and coordinate of given net.

```
get_net_driver_current(net, instance=None, pin=None, mode=None)
```

Get the signal net driver current.

```
get_net_driver_current_from_pwl(net)
```

Get the signal net driver current waveform from pwl.

```
get_net_driver_current_from_vcv(net)
```

Get the signal net driver current waveform from ValueChangeView.

```
get_related_views(view_type=None)
```

Returns a list of related views for this view.

```
get_used_net_parameters_map_function(convert_to_name=False)
```

Get map function for reporting net parameters used for simulation in map reduce job.

```
is_dirty_net_dropped(net)
```

Check to see if the given net is dropped because of dirty.

```
is_net_dirty(net)
```

Check to see if the given net is set as dirty.

```
is_net_dropped(net)
```

For details of a particular query command, use `help` in RedHawk-SC interactive shell as shown in the following example.

```
>>> help(scv_sigem.get_dropped_nets)
```

```
get_dropped_nets(limit=100)
```

Returns the list of nets which been dropped for analysis.

ARGUMENTS

`limit` : max number of nets will be printed. Return all nets if `limit=0`.
 (Default is 100) (type=int, default value=100)

RETURN VALUE

List of nets which been dropped and why they are dropped

The following is a list of commonly used commands to query SignalNetCurrentView.

- <view_name>.get_net_driver_current()
- <view_name>.get_dropped_nets()
- <view_name>.get_dirty_nets()
- <view_name>.get_drop_tags()
- <view_name>.get_dirty_tags()
- <view_name>.get_nets_arcs()
- <view_name>.get_activity()
- <view_name>.get_stats()
- <view_name>.is_net_dropped()
- <view_name>.is_net_dirty()

The following is a list of commonly used commands to query ElectromigrationView.

- <em_view_name>.get_current_mode()
- <em_view_name>.get_current_stats()
- <em_view_name>.get_em_constraint()
- <em_view_name>.get_emViolations()
- <em_view_name>.get_metal_line_number_rating_factor()
- <em_view_name>.get_violations_histograms()
- <em_view_name>.get_temperature_rating_factor()

9: Rampup Analysis for Power-Gated Designs

In complex SoCs containing multiple power-gated blocks, it is important to measure the impact of rampup (rush) current, while switching on the power gated domains. This rush current acts as a tradeoff against the leakage power reduction when the block is in the OFF state. An effect of this rush current is noise coupling – the large amount of current flowing during rampup may lead to additional stress on the PG grid and cause additional IR violations on instances connected to the always-on domain.

RedHawk-SC can efficiently analyze power gated designs to find various parameters such as rampup current, switch turn-on times, and IR drop due to noise coupling. You only need one-time characterization of each rampup block and the characterized results are imported to any regular switching scenario. The tool also allows you to perform differential voltage check and rampup time check.

The following topics describe how to perform rampup analysis by using RedHawk-SC:

- [Power Gating](#) on page 291
 - [Input Requirements](#) on page 294
 - [Rampup Analysis Flow in RedHawk-SC](#) on page 297
 - [Analyzing Rampup Impact Using GUI Heatmaps](#) on page 302
 - [Scripts for Rampup and Absolute Turn-On Time Heatmaps](#) on page 302
 - [Scripts for Rampup Reporting](#) on page 304
 - [Evaluating Results](#) on page 305
-

9.1. Power Gating

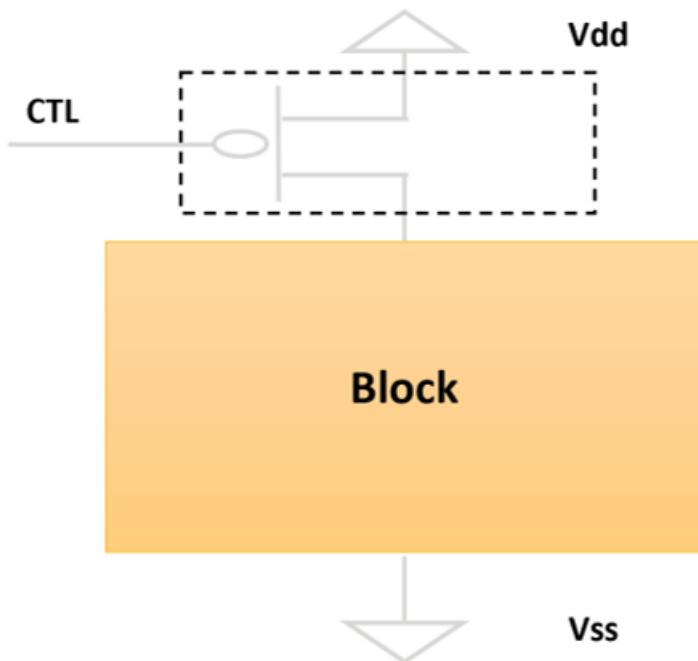
Power gating is one of the most effective techniques to reduce the static power dissipation or leakage power. For the SoCs containing multiple power-gated blocks, it is important to measure the impact of rush current, while switching on the power gated domains. This section describes the switching architecture, implementation methods, and various challenges of the power gated designs.

Switch Architecture

In the power gating structure, PMOS and NMOS transistors are used as switches to turn off power supply to the blocks in standby mode. Header switch uses PMOS transistor to switch VDD supply and footer switch uses NMOS transistor to control VSS supply. Header switches are used for power gating due to less leakage.

Note: For rampup analysis, RedHawk-SC does not support footer switches in the design.

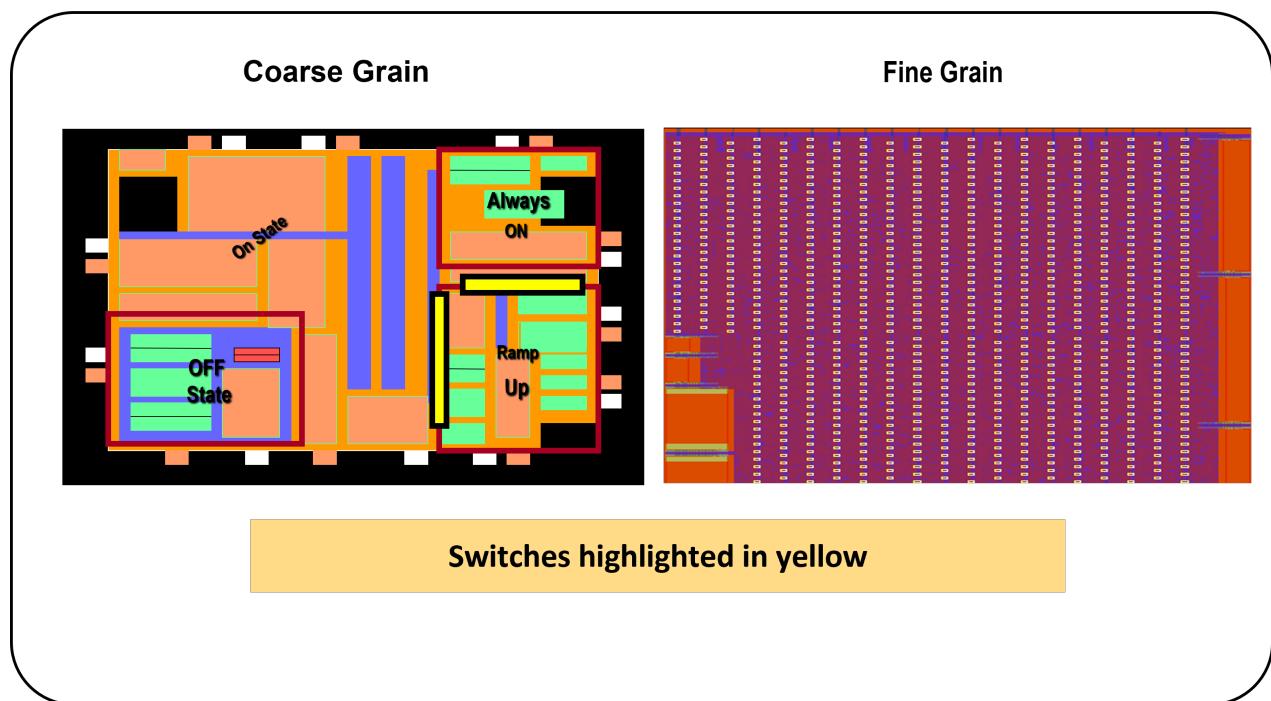
Header Switches (PMOS)



Switch Implementation

Power gating can be implemented using the coarse grain and fine grain techniques. Coarse grain is a commonly used technique where the switches control entire block of the standard cell and reduce the leakage power significantly. Fine grain technique has an in-built power switch.

Full-chip Implementation

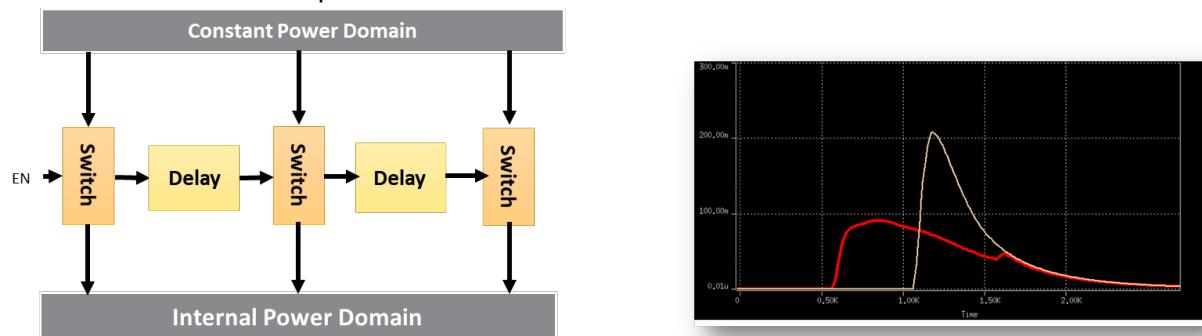


The following table lists the switch operating states.

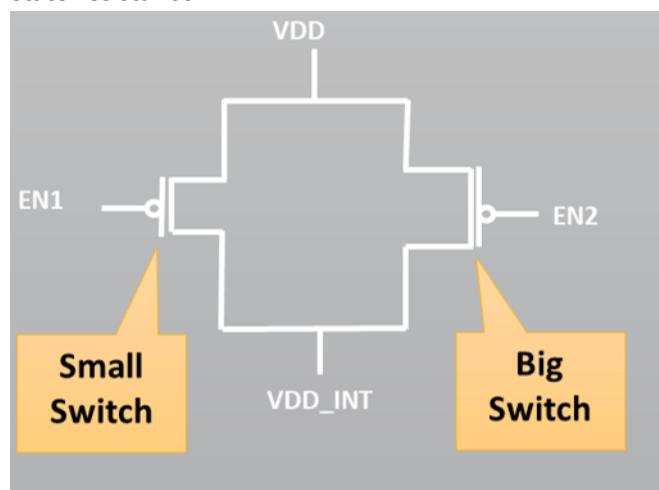
Mode	Description
OFF	When block is inactive.
ON	When block is functional.
Power-up	When block is transitioning from inactive to functional state.
Power-down	When block is transitioning from functional state to inactive state.

Challenges in Power Gated Designs

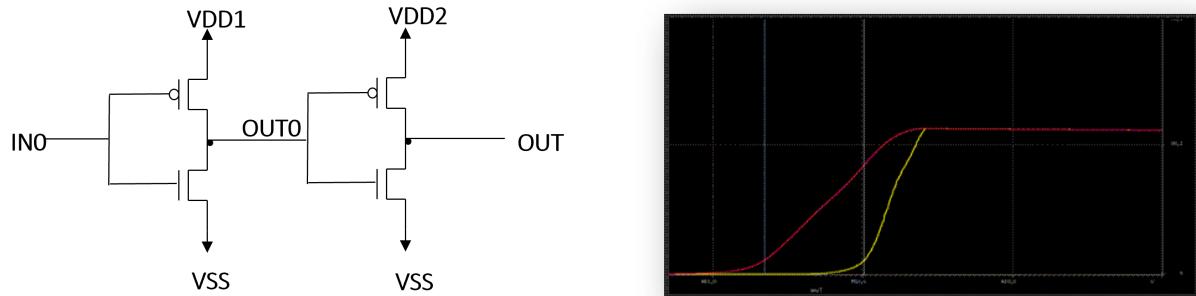
Power gated designs offer various challenges such as high rampup current and voltage drop, and differential voltage. There is high surge of current to charge and discharge internal capacitance when you ramp-up the block, which causes huge voltage drop on the always-on blocks. To reduce rampup current, daisy chain structure is used, wherein delay is introduced between each switch. However, increasing the delay also increases the block wake-up time.



You can reduce the rush current by decreasing the number of switches, but this will increase the current flow through individual switches, power grid resistance (ON state), and wake-up time. High rush current can cause oscillations in the PG network due to fast turn on ($L \frac{di}{dt}$). In general, power switches add more resistance in the power grid and increases the static/dynamic voltage drop during ON state analysis. There is an alternate switch architecture wherein you can use a small switch and a big switch to reduce the rush current and ON state voltage drop. Small switch is used to charge the internal nodes and big switch is used to reduce ON state resistance.



RedHawk-SC performs differential voltage analysis on the driver-receiver switch pair for each internal domain. In the following example, the receiver (VDD2) ramps up faster than the driver (VDD1), therefore, OUT0 remains above the threshold voltage for more time. This leads to high crowbar current in the receiver cell. The crowbar current is more significant when the wakeup time is more.



9.2. Input Requirements

The following inputs are to be specified while invoking RedHawk-SC for rampup analysis:

- LibertyView
 - APL switch model file
 - APL PWCAP file
- ScenarioView
 - LSO for switch control pins
 - External currents dumped from rampup AnalysisView

9.2.1. LibertyView

Rampup current requires a detailed electrical model for each power gate cell in the design, as well as models that capture the leakage current characteristics of standard cells or macros in the virtual (internal) domain. These models can be generated while characterizing the APLs for a design. They may also be available in CCS Liberty files.

Switch Model File

A 3D PWL switch model file is required for setting up the flow in RedHawk-SC. The following example of 3D switch model shows switch type, control pin definitions, on state characteristics, PWL indices, and PWL table. The PWL table displays the instantaneous switch current for each combination of V_{ds} (difference between internal pin and external pin voltage) and V_{gs} (difference between control pin and external pin voltage).

```

SWITCH_CELL SWITCH_CELL {
    VDD_number 1
    VDD 0.855 {
        SWITCH_TYPE: HEADER      Switch Type and Supply Pins
        EXT_PIN: VDD
        INT_PIN: VDD_INT
        CTRL_PIN: CNTL_IN1 R F  Control Pin Definitions
        CTRL_PIN: CNTL_IN2 R F
        ON:
            R 9.77091
            I 2.60504e-08  On State Characteristics
            C 5.03316e-14
            IDSAT 0.0487684
        OFF:
            C 4.81777e-14
        PWL_CURRENT: 3          Indices for PWL
    V VDD VDD_INT 12
    0 0.0777273 0.155455 0.233182 0.310909 0.388636 0.466364 0.544091 0.621818 0.699545 0.777273 0.855
    V VDD CNTL_IN1 12
    0 0.0777273 0.155455 0.233182 0.310909 0.388636 0.466364 0.544091 0.621818 0.699545 0.777273 0.855
    V VDD CNTL_IN2 12
    0 0.0777273 0.155455 0.233182 0.310909 0.388636 0.466364 0.544091 0.621818 0.699545 0.777273 0.855
    I VDD VDD_INT 1728
    1.903833e-05 1.905088e-05 1.914668e-05 1.981026e-05
    2.296024e-05 2.945799e-05 1.294351e-07 3.482081e-08
    7.343266e-09 2.770282e-09 2.169771e-09 2.097137e-09  PWL Table
    1.903966e-05 1.905222e-05 1.914801e-05 1.981157e-05
    2.296150e-05 2.945912e-05 1.294549e-07 3.484252e-08
    7.365503e-09 2.792716e-09 2.182219e-09 2.110588e-09

```

These files are usually provided by the library vendor and may be generated using the APLSW tool by setting 'MD_PWL 1' in the configuration file. The switch properties may also be captured in CCS Liberty files if switch model files are not available.

APL PWCAP File

For standard cells and macros in the virtual domain, RedHawk-SC requires you to provide the APL PWCAP (PieceWiseCap) file. The PWCAP tables consist of the instantaneous capacitance, resistance and leakage current for a given cell at multiple values of supply voltage. RedHawk-SC uses this information for OFF state voltage calculation.

```

Info: Reading pwlcap file- APL PWCAP file path

Info: cell= cell1
Info:     arc= vdd vss, vdd= 0.092 V, cap= 0.00726232 pf, res= 9.99394 ohm, leak= 3.22242e-05 uA
Info:     arc= vdd vss, vdd= 0.552 V, cap= 0.0106247 pf, res= 7543.39 ohm, leak= 2.33147e-05 uA
Info:     arc= vdd vss, vdd= 1.012 V, cap= 0.0157762 pf, res= 100.911 ohm, leak= 0.00064071 uA

Info: cell= cell2
Info:     arc= vdd vss, vdd= 0.092 V, cap= 0.00836681 pf, res= 9.91009 ohm, leak= 1e-06 uA
Info:     arc= vdd vss, vdd= 0.552 V, cap= 0.0151231 pf, res= 1045.85 ohm, leak= 0.000172584 uA
Info:     arc= vdd vss, vdd= 1.012 V, cap= 0.0159112 pf, res= 97.2904 ohm, leak= 0.00259348 uA

Info: cell= cell3
Info:     arc= vdd vss, vdd= 0.092 V, cap= 0.00730174 pf, res= 9.96193 ohm, leak= 3.20161e-05 uA
Info:     arc= vdd vss, vdd= 0.552 V, cap= 0.0103328 pf, res= 7657.04 ohm, leak= 2.38143e-05 uA
Info:     arc= vdd vss, vdd= 1.012 V, cap= 0.0162181 pf, res= 107.194 ohm, leak= 0.000703104 uA

```

Use the following template to specify the switch model file and APL PWCAP file pointers:

```

apl_files = [ <.. add any APL current files etc ..>,
    {'file_name' : <path_to_pwcap_file_1> },
    {'file_name' : <path_to_pwcap_file_2> },]

switch_files = [

```

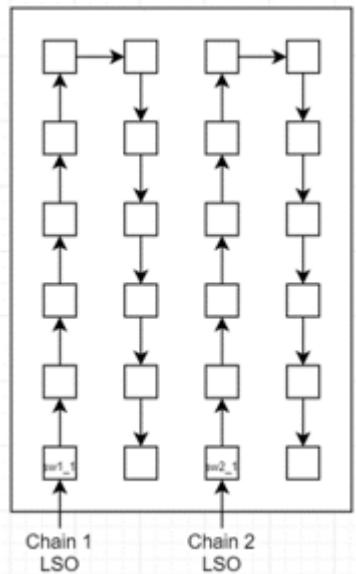
```
{ 'file_name' : <path_to_switch_model_file_1>, 'temperature' : <T> }
{ 'file_name' : <path_to_switch_model_file_2>, 'temperature' : <T> } ]
```

These files are passed to the LibertyView using the following command:

```
lv = db.create_liberty_view( ...,
    apl_files = apl_files,
    apl_switch_file_names = switch_files)
```

9.2.2. ScenarioView

Another important input for rampup analysis is the logic signal override (LSO) file to provide input signals at the power gate control inputs. For typical designs with switch chains, the LSO input must be provided at the head of each chain. RedHawk-SC propagates the control pin logic based on the cell delay available from the Liberty file.



LSO syntax:

```
# version statement is mandatory
$version = 1.0
# Set the time unit
$time_unit = 1.0e-09
# Each entry is in the following format
# @ip <inst_name>/<pin_name> {
#     (t1, logic_at_t1),
#     (t2, logic_at_t12),
#     (t1, logic_at_t1),
#     (tn, logic_at_tln),
# }
# Specify a logic rise at the head of switch chain 1 ,
# Starting at 10 ns with a slew of 40 ps
@ip head_sw_1/ctrl_in {
    (0.0, 0),
    (10.0, 0),
    (10.04, 1)
}
```

```
# Specify a logic fall at the head of switch chain 2,
# Starting at 15 ns with a slew of 100 ps
@ip head_sw_2/ctrl_in {
    (0.0, 1)
    (15.0, 1),
    (15.1, 0)
}
```

Use the following functions to create an LSO:

- Returns a dict of { Net : List of Instances }

```
>>> power_gate_instances = dv.get_power_gate_instances()
```

- Returns a dict of { Net : List of Instance, Pin tuples }

```
>>> power_gate_instance_pins = dv.get_power_gate_instance_pins()
```

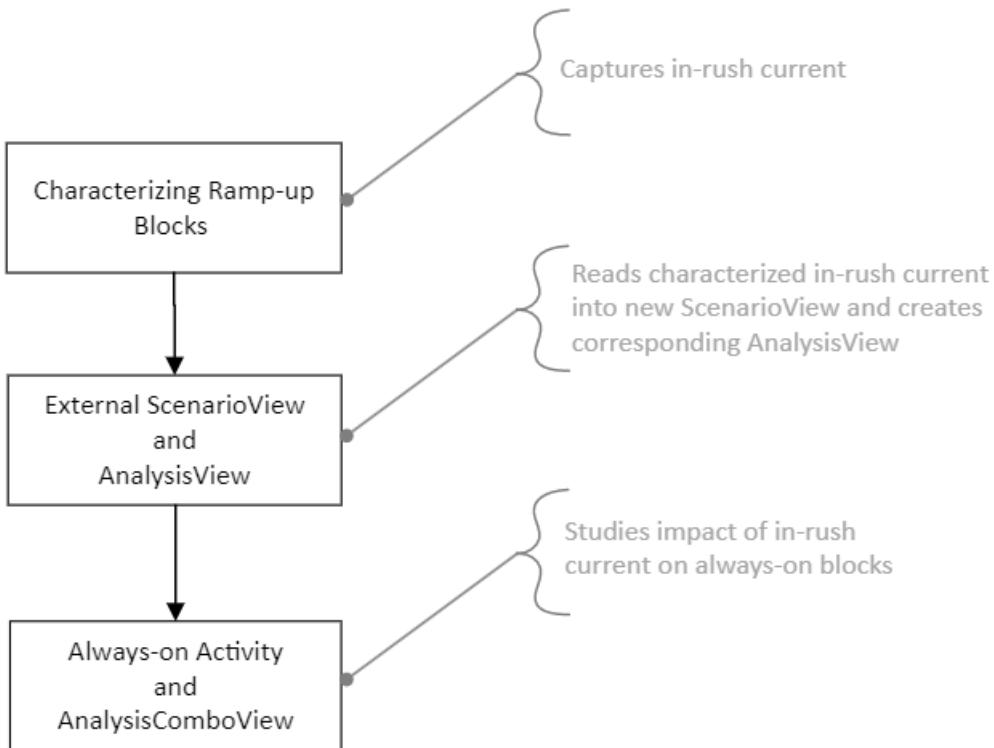
- Check power gate annotation in TimingView

```
>>> tv.get_tw_attributes( switch_instance )
```

To generate an LSO from the TimingView, import `get_lso_from_tv` from `thpkgs.ae_utils`.

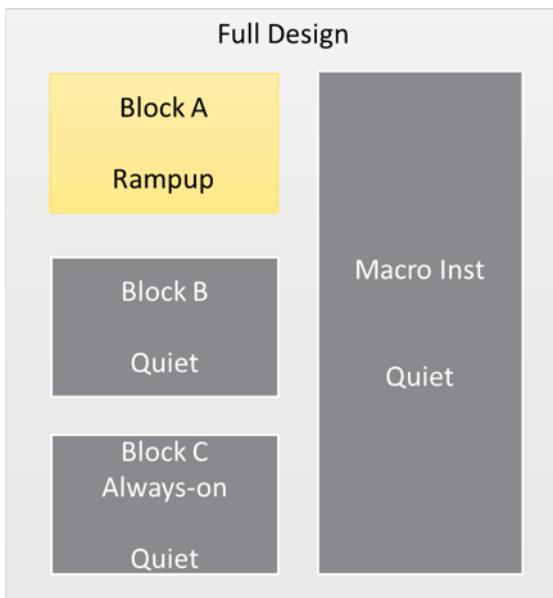
9.3. Rampup Analysis Flow in RedHawk-SC

RedHawk-SC uses superposition principle to perform multi-block rampup analysis. This principle helps in much faster and longer simulations than the traditional methods. The rampup analysis flow involves the following steps:



Characterizing Rampup Blocks

The aim of characterization step is to write out a file (*.power_gate_currents) capturing the currents flowing through each power gate cell. This file can then be subsequently read back while creating the rampup AnalysisView in the next step. In this stage, RedHawk-SC performs rampup analysis on a single rampup block and keep rest of the blocks in the quiet state or with zero activity, which helps in studying the isolated effects of rampup of the single block. The `ramp_up_nets` argument is the main input, which is passed to the AnalysisView. The AnalysisView uses the LSO file given to the TimingView to determine the switch turn-on times. Characterization should be done for each rampup block separately. Only one time characterization is required, you need not execute these commands again in the future runs. The `RampUp` scenario type is used to create a scenario where you do not provide the scenario duration, instead the tool calculates it based on the LSO file. These views allow you to generate rampup time, total power gate currents, and switch turn-on heatmaps.



Syntax for characterizing rampup blocks:

- Create a quiet ScenarioView using LSO files.

```
scn_quiet_settings = {
    'event': {
        '...', 'block_name' : '*', 'toggle_rate' : 0.0},
        'clock_source_toggle_rates': {'*': 0.0}}
    'scenario_type': 'RampUp'
}

scn_quiet = db.create_scenario_view(..., tv = tv, lso_files =
<pointer_to_lso_file>), settings=scn_quiet_settings)
```

- Create AnalysisView using the quiet ScenarioView and write out the power gate currents.

- **keep_stats_level:** a setting that controls the data that is saved during simulation. The valid strings to this setting are ['Bqm', 'Full', 'High', 'Low', 'Medium', 'SystemResponse'].
- **ramp_up_nets:** a list of PG nets that are in rampup state. If this key is not set, there are no ramp-up nets.
- **off_state_nets:** a list of PG nets that are in OFF state. If this key is not set, there are no off-state nets.

```
keep_stats_level = KeepStats('Low', 'pin_voltages' = True,
    'power_gate_currents' = True)

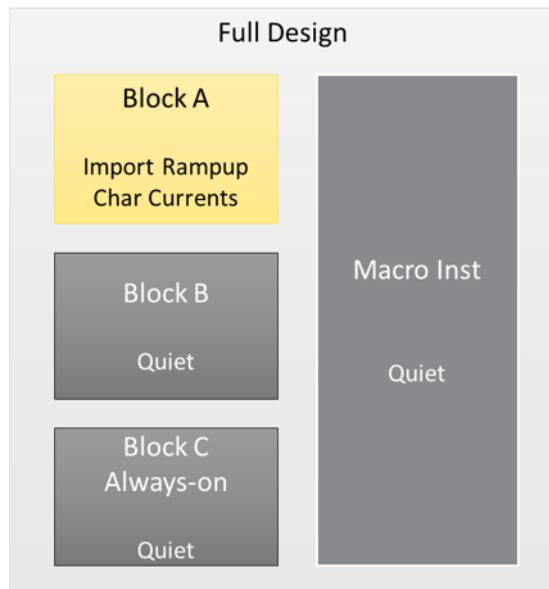
av_ru_quiet_settings = {
    'keep_stats_level' : keep_stats_level,
    'ramp_up_nets' : [Net('<block_internal_net>')]
}

av_ru_quiet = db.create_analysis_view(
    options = options,
    scenario_views = scn_quiet,
    settings = av_ru_quiet_settings,
    tag = 'av_ru_quiet'
)
```

```
write_power_gate_currents(av_char, '<block_name>.power_gate_currents',
[Net('<block_internal_net>')])
```

External ScenarioView and AnalysisView

In this step, the characterized rampup currents are read in to a new ScenarioView, and the corresponding AnalysisView is created. This step and the next step ([AnalysisComboView](#) on page 301) are required only if you want to study the impact of rampup on the full-chip. For each rampup block, create the ScenarioView and AnalysisView as described. These views allow you to query IR heatmaps, instance IR and node IR. To perform what-if analysis in future, directly read the current file from [Characterizing Rampup Blocks](#) on page 298.



Syntax for reading the characterized currents and creating external ScenarioView and AnalysisView:

- Create a pointer to the characterized currents file.

```
current_source_files = [<block_name> + '.power_gate_currents']
```

- Import these currents file into an External ScenarioView.

```
scn_ru_settings = {
'pvt' : {'voltage_levels' : voltage_levels},
'scenario_type' : 'External'
}
scn_ru = db.create_scenario_view(design_view = dv, current_source_files =
current_source_files, tag = 'scn_ru', settings = scn_ru_settings, options =
options)
```

- Create AnalysisView using an External ScenarioView.

Note: Any AnalysisView passed to the `SeaScapeDB.create_analysis_combo_view` command should be created with `KeepStats(enable_for_combo_view=True)`. Otherwise, the run errors out.

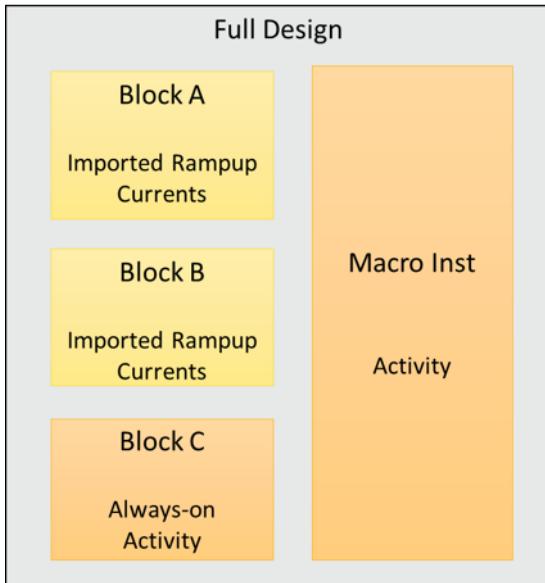
```
# Solve matrix.
combo_keep_stats_level = KeepStats('Full', enable_for_combo_view=True)

av_ru_settings = {
    'duration': 160e-9,
    'step_size': 150e-12,
    'keep_stats_level': combo_keep_stats_level,
    'initial_condition': 't0',
    'off_state_nets' : ramp_up_nets
}

av_ru = db.create_analysis_view(
    options = options,
    scenario_views = scn_ru,
    #simulation_view = simulation_view,
    settings = av_ru_settings,
    tag = 'av_ru'
)
```

AnalysisComboView

This is the final step in the flow that is used to study the impact of rampup current when combined with other sources of activity in an SoC, such as always-on blocks. The AnalysisComboView is based on the principle of superposition. For each rampup AnalysisView, specify the time during simulation at which current values should be passed into the base AnalysisView. These views allow you to generate IR drop, heatmaps, and voltages statistics, when rampup is considered.



Syntax for creating AnalysisComboView for always-on blocks:

- Create ScenarioView and AnalysisView to use as the base view. The `scn_always_on` is a scenario of any type with vectorless and vector-based activity on always-on instances, and macros.

```
av_always_on_settings = {
    'off_state_nets' : [Net('<block_internal_net>')],
    'keep_stats_level' : 'Full',
    'step_size' : <t_step_ru>,
    'duration' : <t_sim_ru>
}

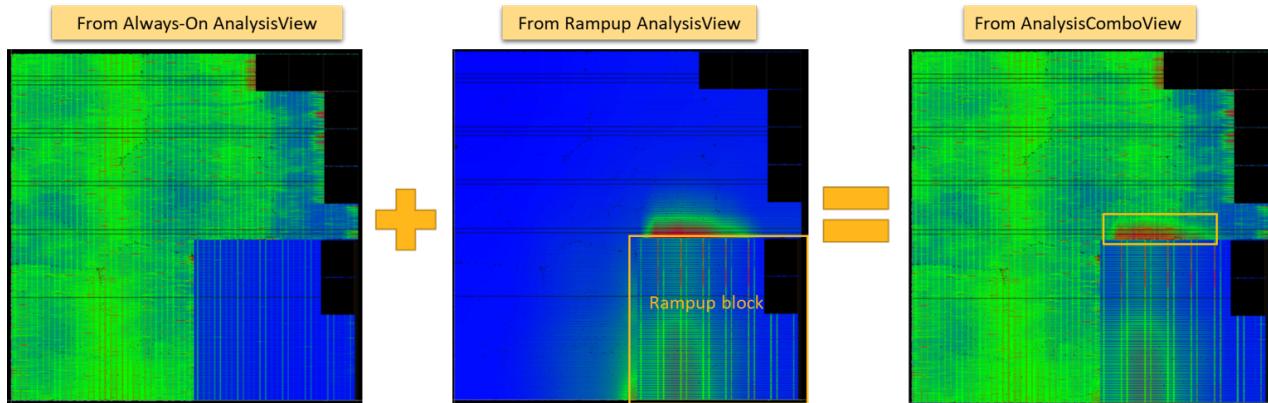
av_always_on = db.create_analysis_view(..., scenario_views = scn_always_on,
settings =av_always_on_settings)
```

- Create AnalysisComboView using the `create_analysis_combo_view` command and the following arguments:
 - Base AnalysisView object (`av_always_on`).
 - A list of rampup AnalysisView objects (`[av_ru]`) that are superposed on the base AnalysisView.
 - A list of offsets for each rampup AnalysisView (`[10.0e-9]`).
 - A tag representing the view name.

```
av_combo = db.create_analysis_combo_view(av_always_on, [av_ru], [10.0e-09],
tag = 'av_combo_1')
```

9.4. Analyzing Rampup Impact Using GUI Heatmaps

For heatmap-based analytics, the RedHawk-SC GUI allows you to view the instance or node voltage heatmaps from the always-on AnalysisView, rampup AnalysisView, and AnalysisComboView. After running the scripts for generating heatmaps, open the heatmaps in the GUI and use the principle of superposition to overlay rampup AnalysisView on the always-on AnalysisView to generate AnalysisComboView. In AnalysisComboView, you can view the impact of rampup on the always-on block. On comparing the following heatmaps, it is possible to identify zones in the design where rampup currents lead to new IR violations.

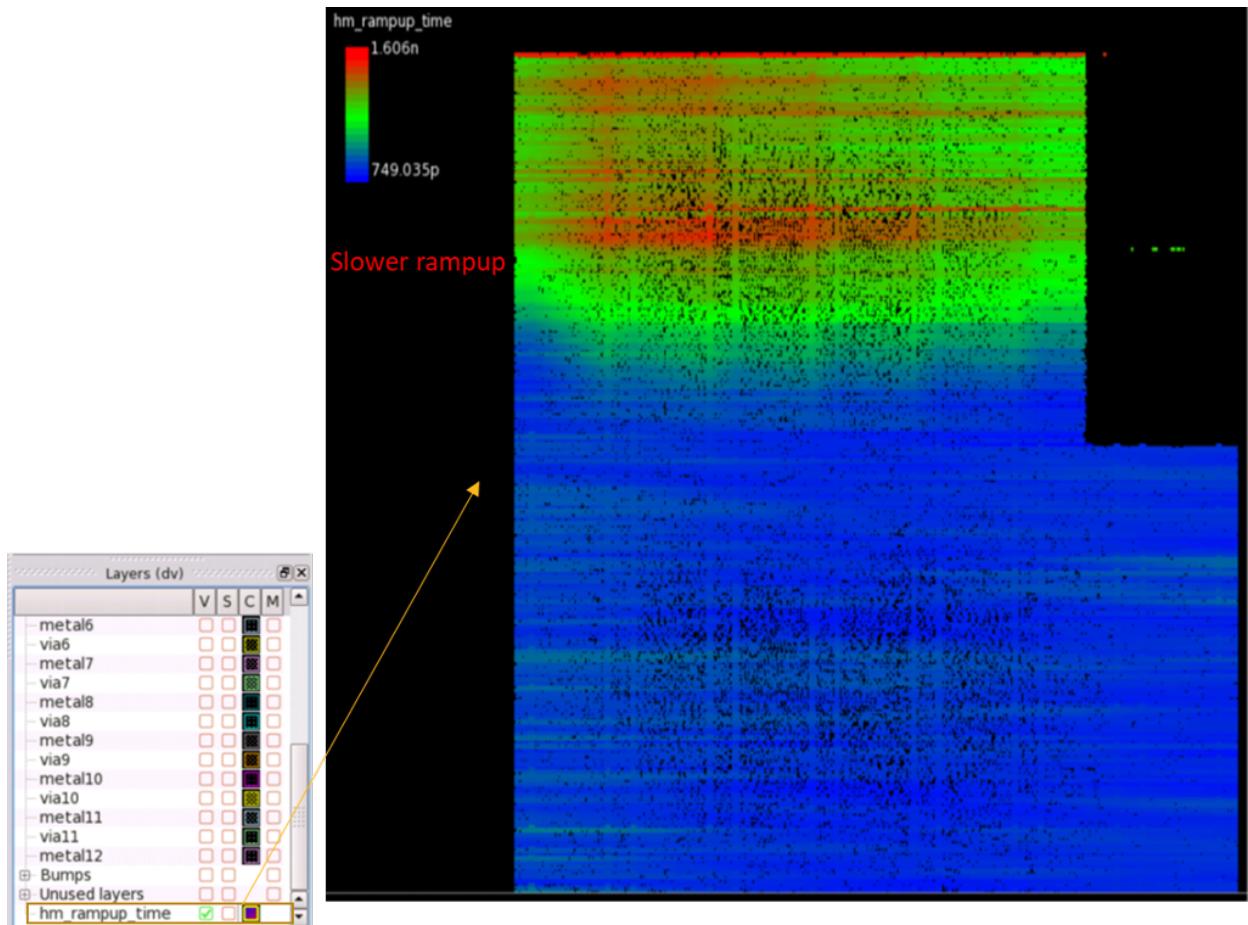


9.5. Scripts for Rampup and Absolute Turn-On Time Heatmaps

The following scripts are used for generating rampup and absolute switch turn-on time heatmaps:

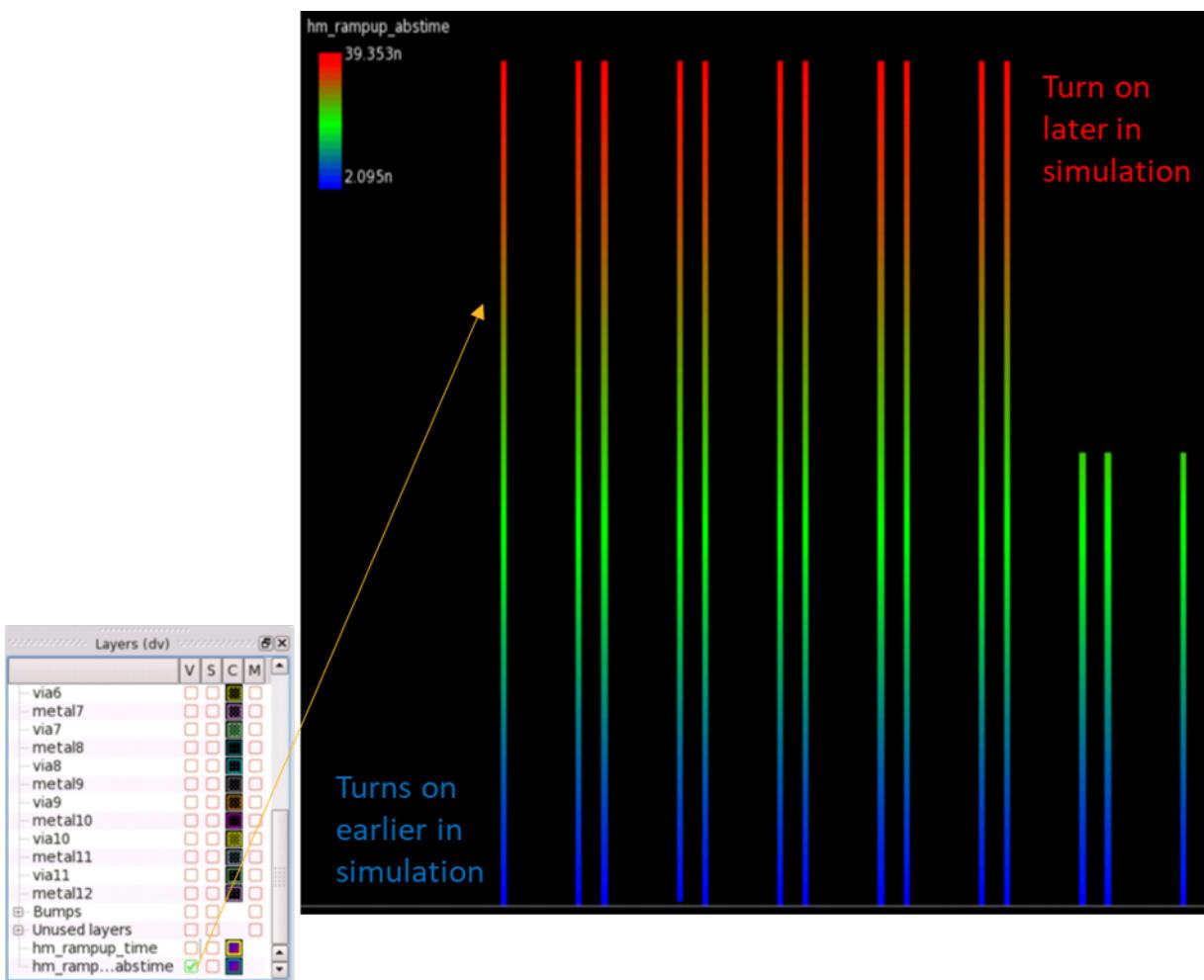
- Rampup time is the time for internal pin voltage to change from 10% to 90%. Use the following function to generate rampup time heatmaps.

```
>>> from thpkgs.ae_utils import rampup_get_worst_node
>>> heatmaps_ru = rampup_get_worst_node.get_rampup_node_heatmap(
    av           = av_ru_quiet,
    domains      = [Net('core3/VDD_INT')])
>>> gui.add_layer(heatmaps_ru[Net('core3/VDD_INT')])
```



- Absolute switch turn-on time is the time at which internal pin voltage reaches 90% of the nominal voltage. Use the following function to generate absolute switch turn-on time heatmaps.

```
>>> from thpkgs.ae_utils import rampup_get_worst_node
>>> heatmaps_abs = rampup_get_worst_node.get_rampup_node_heatmap(
    av           = av_ru_quiet,
    domains      = [Net('core3/VDD_INT')], 
    from_zero    = True,
    check_only_power_gates = True)
>>> gui.add_layer(heatmaps_abs[Net('core3/VDD_INT')])
```



9.6. Scripts for Rampup Reporting

Scripts are available as part of the RedHawk-SC build to perform differential voltage check. The script dumps a report with a list of switch pairs having the worst differential voltage, along with a list of driver-receiver pairs in the design that are affected by the worst differential voltage.

- Generating the differential voltage report

```
>>> from thpkgs.ae_utils import rampup_get_differential_report

>>> rampup_get_differential_report.dump_differential_voltage_report(
    filename      = 'VDD_INT_differential_voltage.rpt',
    av           = av_ru_quiet,
    internal_net = Net('core3/VDD_INT'))
```

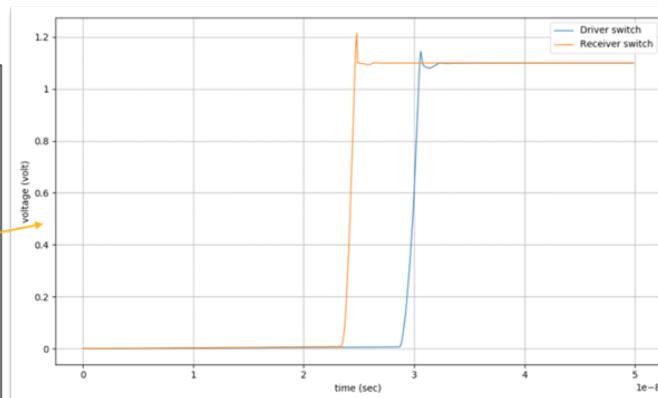
- Use `help(rampup_get_differential_report.dump_differential_voltage_report)` to obtain a list of all the arguments.

```
>>> rampup_get_differential_report.dump_differential_voltage_report(
    filename      = '<net_name>_differential_voltage.rpt',
    av           = av_char,
    internal_net = Net('<internal_net>'))
```

```
# Differential Voltage Report for Net('core3/VDD_INT')
#
# <DriverSwitch> <ReceiverSwitch> <DifferentialVoltage>
# - <DriverInst1> <ReceiverInst1>
# - <DriverInst2> <ReceiverInst2>

core3/inst_sw_S2_929 core3/inst_sw_S2_754 1.208
- core3/HFSINV_792_1928
core3/regfile_program_memory.MUX2_X1_3678
- core3/HFSINV_792_1928
core3/regfile_program_memory.MUX2_X1_3680
- core3/HFSINV_846_2579
core3/regfile_program_memory.MUX2_X1_2693

core3/inst_sw_S1_977 core3/inst_sw_S2_754 1.207
- core3/ZBUF_2_inst_25980 core3/ZBUF_17_inst_19280
```



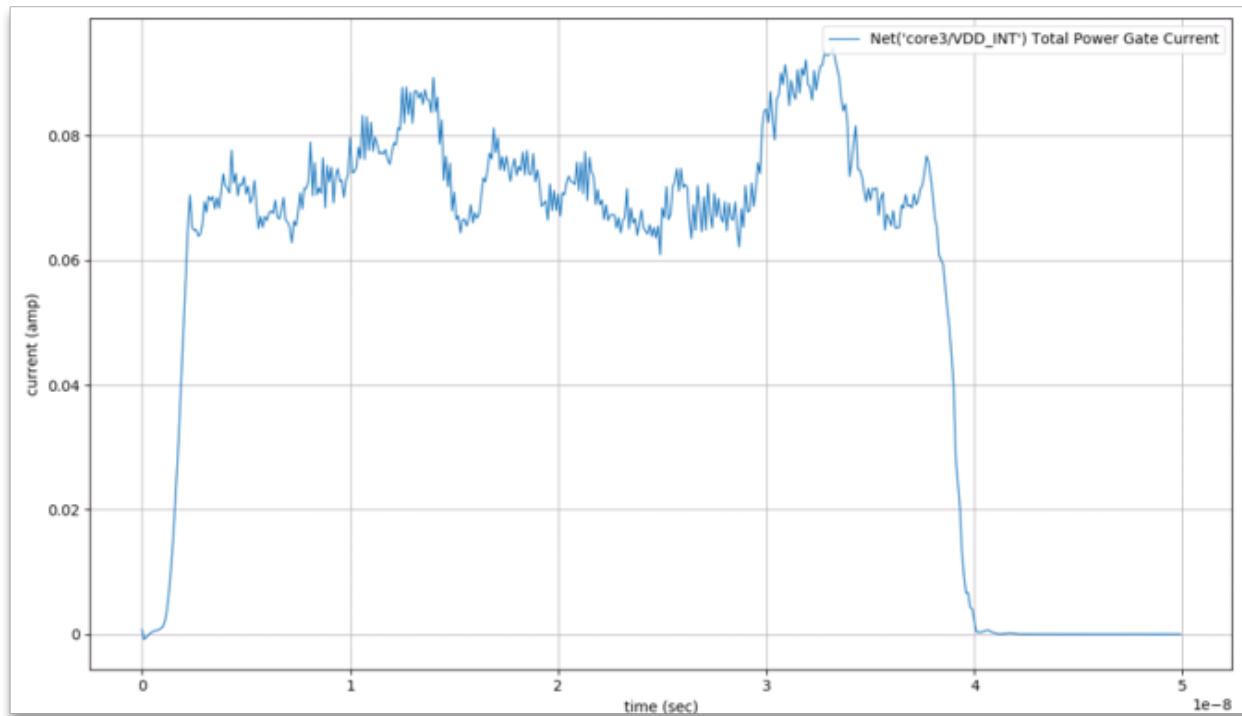
9.7. Evaluating Results

This chapter describes the functions that can be used to evaluate the results of RedHawk-SC rampup analysis.

Virtual Domain Currents and Voltages

Virtual domain currents and voltages can be queried from the AnalysisView created during block characterization. The virtual domain currents are calculated as the sum of currents through each power gate in the domain. These currents are dependent on the number and distribution of power gates, switch turn-on times and the total capacitive load. Typically, the parameters extracted from this plot are the peak virtual domain current and the area under the curve represents the total charge.

```
>>> av_ru_quiet.get_total_power_gate_currents().keys()
[Net('core3/VDD_INT')]
>>> plot(av_ru_quiet.get_total_power_gate_currents()[Net('core3/VDD_INT')])
```



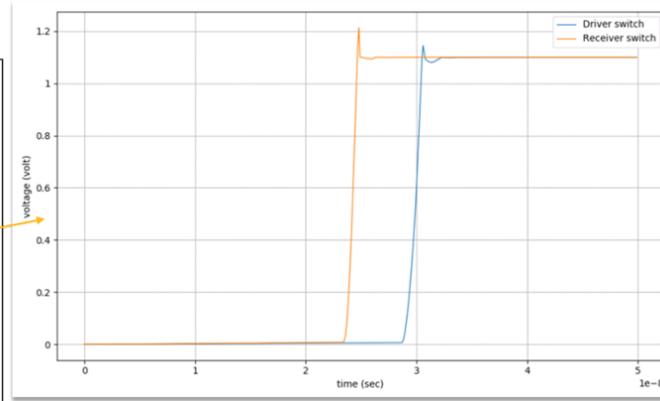
The rampup voltages can be queried at the internal power pin of each power gate instance.

```
av_ru_quiet.get_voltage(sw_inst, Pin('<internal_pin>'))
>>>first_sw_wfm = av_ru_quiet.get_voltage(Instance('core3/inst_sw_S1_0') ,
Pin('VDD_INT'))
>>>last_sw_wfm =
av_ru_quiet.get_voltage(Instance('core3/inst_sw_S1_1319'),Pin('VDD_INT'))
>>>plot([first_sw_wfm , last_sw_wfm])
```

```
# Differential Voltage Report for Net('core3/VDD_INT')
#
# <DriverSwitch> <ReceiverSwitch> <DifferentialVoltage>
# - <DriverInst1> <ReceiverInst1>
# - <DriverInst2> <ReceiverInst2>

core3/inst_sw_S2_929 core3/inst_sw_S2_754 1.208
- core3/HFSINV_792_1928
core3/regfile_program_memory.MUX2_X1_3678
- core3/HFSINV_792_1928
core3/regfile_program_memory.MUX2_X1_3680
- core3/HFSINV_846_2579
core3/regfile_program_memory.MUX2_X1_2693

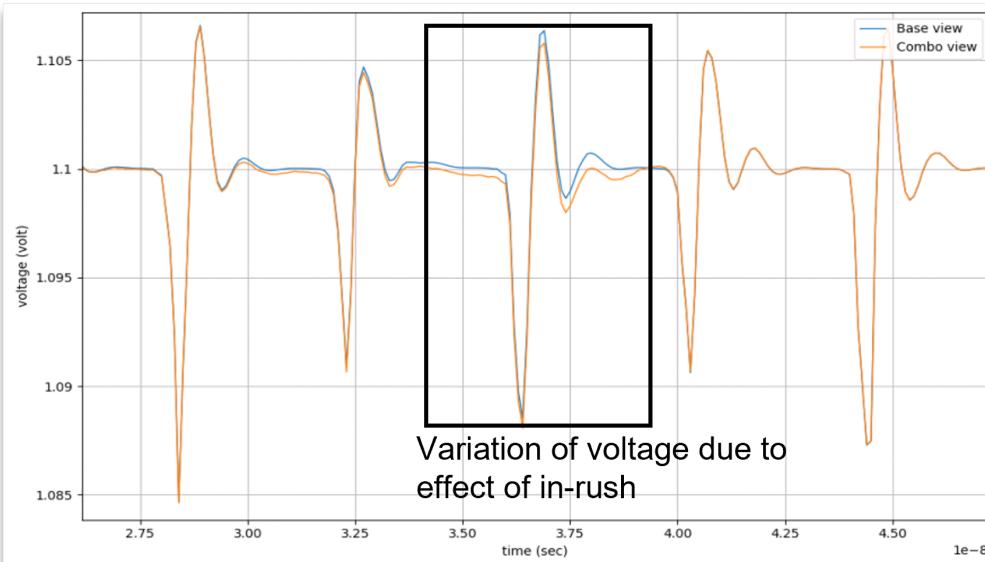
core3/inst_sw_S1_977 core3/inst_sw_S2_754 1.207
- core3/ZBUF_2_inst_25980 core3/ZBUF_17_inst_19280
```



Rampup Impact on Adjacent Blocks

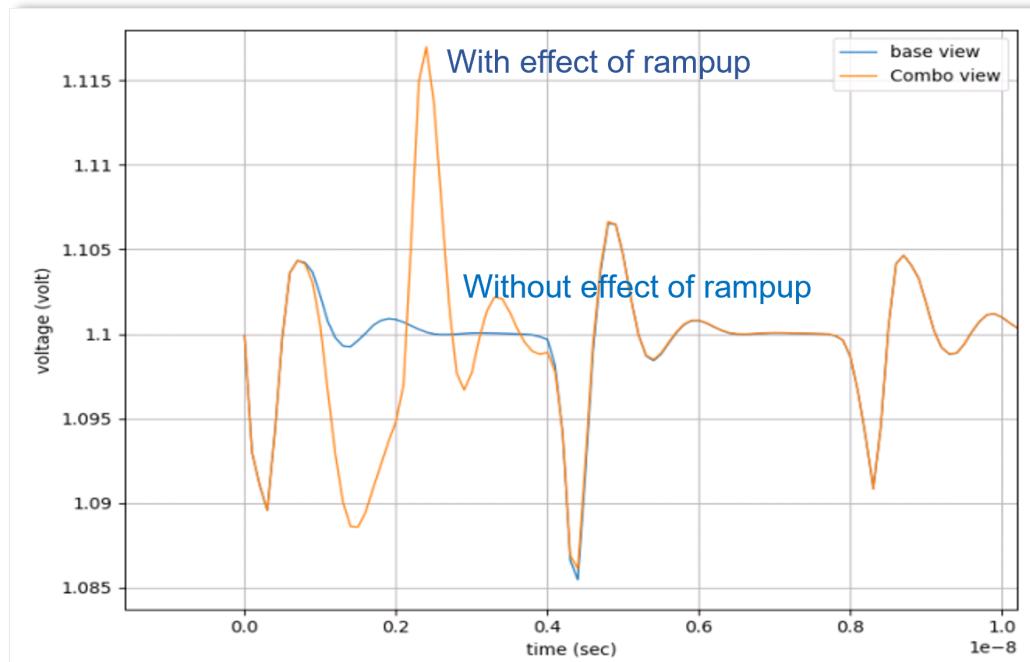
To study the impact of rampup currents on adjacent regions, use the following AnalysisComboView function to query instance voltages, heatmaps, and voltage stats. Any variation in the voltages between the always-on AnalysisView and the AnalysisComboView can be attributed to the rampup currents.

```
>>> base_wfm =
av_ru_always_on.get_voltage(Instance("core0.regfile_data_memory.DFF_X1_740"),Pin('VDD'))
>>> combo_wfm =
av_combo_1.get_voltage(Instance("core0.regfile_data_memory.DFF_X1_740"),Pin('VDD'))
>>> plot([base_wfm,combo_wfm] , labels = ['Base view' , 'Combo view'])
```



Impact of Changing LSO

You can modify the LSO settings to perform what-if analysis and create different rampup scenarios. In this example of worst case scenario, all the switches are switching simultaneously at a particular time and you can observe high voltage drop as the result of rampup analysis on the always-on blocks. Similarly, you can check for multiple scenarios where switches do not turn on simultaneously, switch delay is modified, or all the switches turn on at the same time.



10: Chip Power Modeling

Chip package and printed circuit board (PCB) designers need an accurate and simple IC power model to design and optimize effective packages and boards, including key parameters such as impedance and resonant frequencies of the global power delivery network (PDN). An equivalent circuit for the chip PDN must provide not only an accurate multiple-terminal impedance model, but also accurate current waveforms to represent a realistic worst-case switching scenario of the chip.

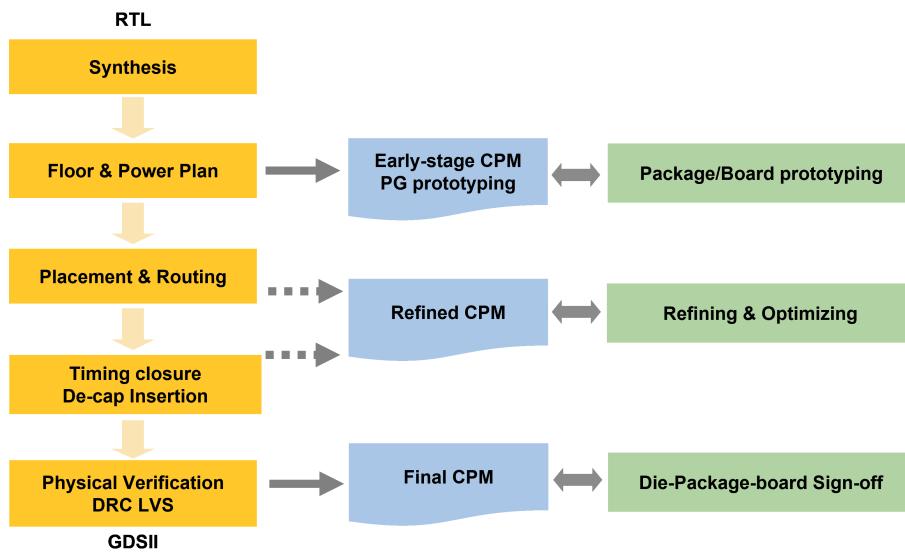
This chapter describes the following topics to generate and use a Chip Power Model (CPM) by using the RedHawk-SC tool:

- [Overview](#) on page 309
- [Creating CPM Using Single Worker](#) on page 311
- [Creating CPM Using Multiple Workers](#) on page 321
- [Storing CPM Output](#) on page 322
- [Validating CPM](#) on page 327

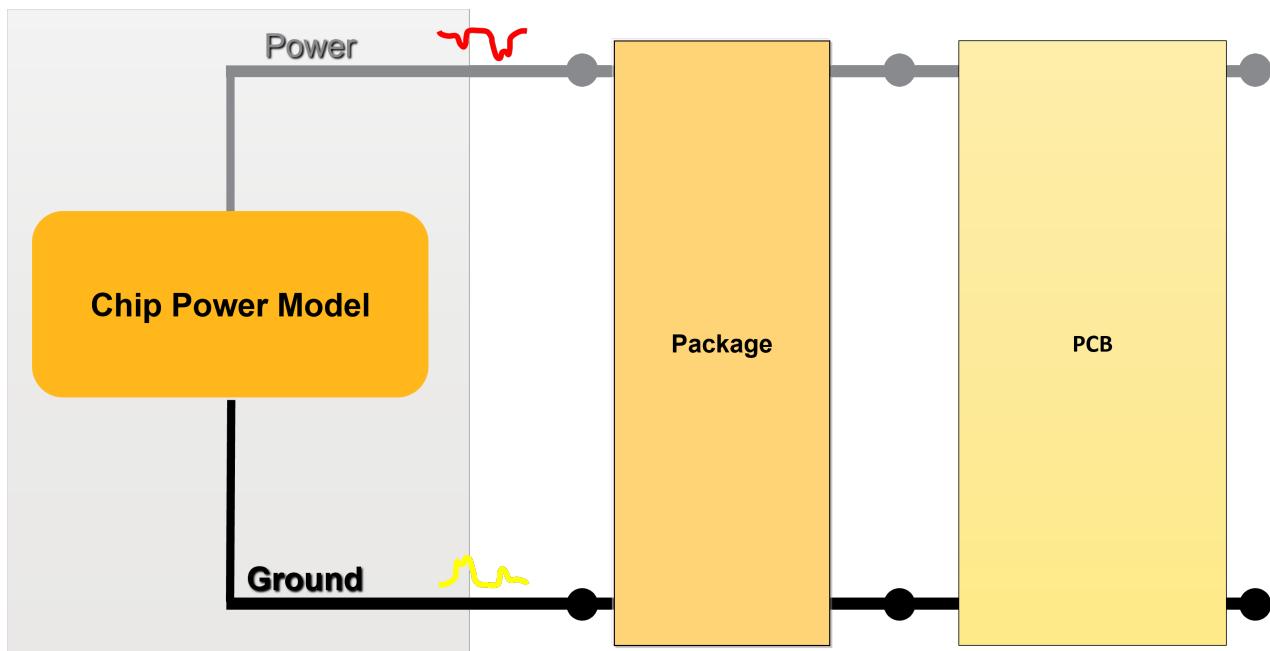
10.1. Overview

The RedHawk-SC Chip Power Model (CPM) enables die-package-board co-design and co-verification for dynamic power integrity. Built on the RedHawk-SC full-chip dynamic power integrity platform, the CPM engine generates a compact and accurate model of the full-chip power delivery network (PDN) at various key stages of chip and package design.

Figure 58. CPM based IC-System Co-Design



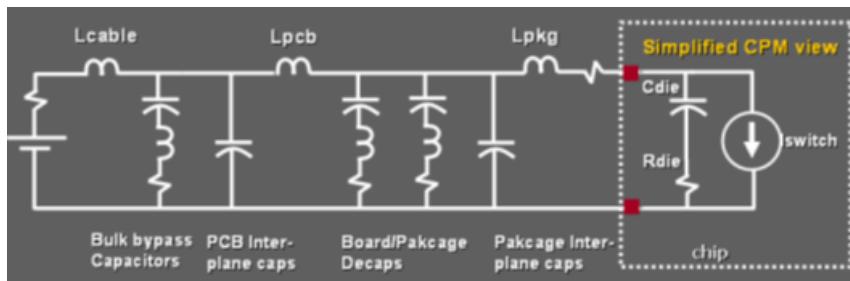
The following is a simple diagram showing a CPM of the chip that provides the power and ground noise information for optimum package and printed circuit board (PCB) designs.



CPM bridges the PDN design between the IC and the associated package and PCB. System designers can use CPM to guide and verify the off-chip PDN design by evaluating the impact of on-die parasitics over the global PDN impedance, diagnose potential chip-package LC resonance, validate the package and board dynamic voltage noise margins, and optimize the off-chip decoupling capacitor placements.

In a CPM, both current signatures and parasitic network are distributed across an equivalent multiple-terminal circuit to reflect their temporal and spatial dependencies. The simplest form of the CPM can be considered as a serially-connected R_{die} and a C_{die} , in parallel with a current switch, as shown in the following figure:

Figure 59. Power Delivery Network and Simple Chip Power Model



C_{die} is the frequency-dependent effective capacitance of the die that is determined by measuring the input impedance of the chip PDN. At low frequencies, C_{die} value closely matches the algebraic sum of all on-die capacitances. R_{die} is the effective die resistance.

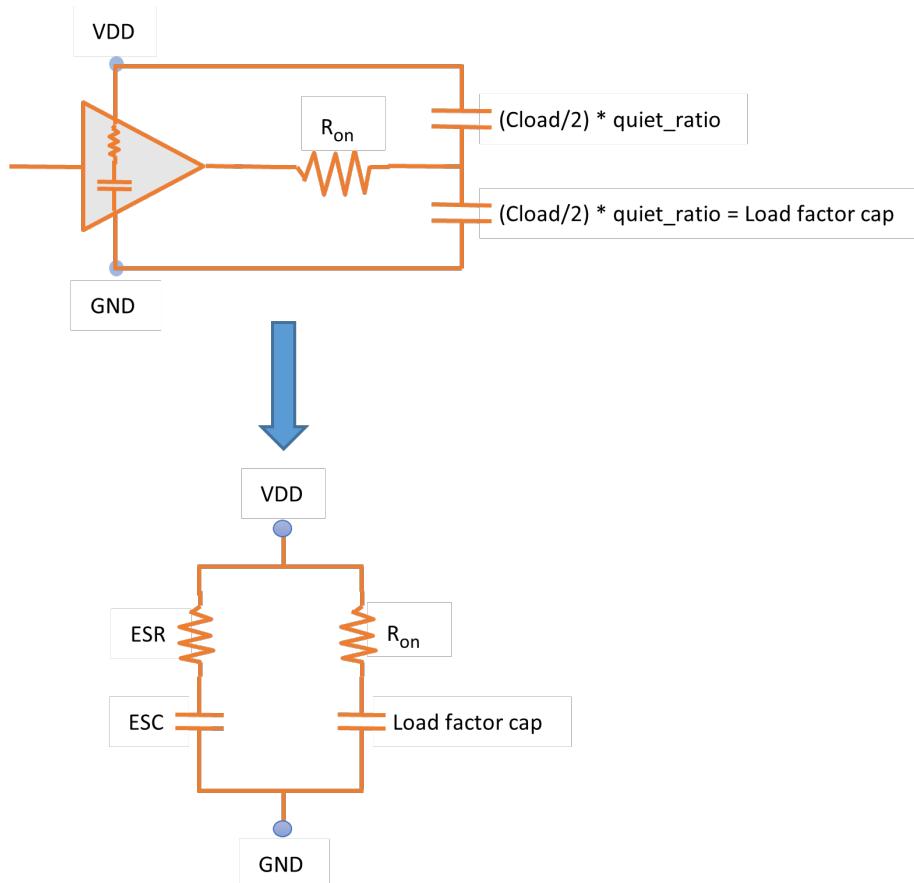
SPICE-compatible CPM includes parasitics of non-linear switching and non-switching devices, parasitics of power and ground wires, decoupling capacitors, and effective RC values of load capacitances from signal interconnects. In addition, CPM contains full-chip switching current signatures based on transistor-level SPICE simulations. For details about the contents of a CPM, see [Storing CPM Output on page 322](#).

Significant chip capacitance sources including parasitics and decoupling capacitors are included in the RedHawk-SC CPM, as shown in the following circuit:

- Intentional decoupling capacitors from RedHawk-SC characterization (APL)
- Intrinsic device decoupling capacitors from RedHawk-SC characterization (APL)
- Signal loading capacitors from SPEF

- Coupling capacitors between power and ground wires from RedHawk-SC extraction

Figure 60. Simple Representation of Parasitics and Decoupling Capacitors in RedHawk-SC CPM

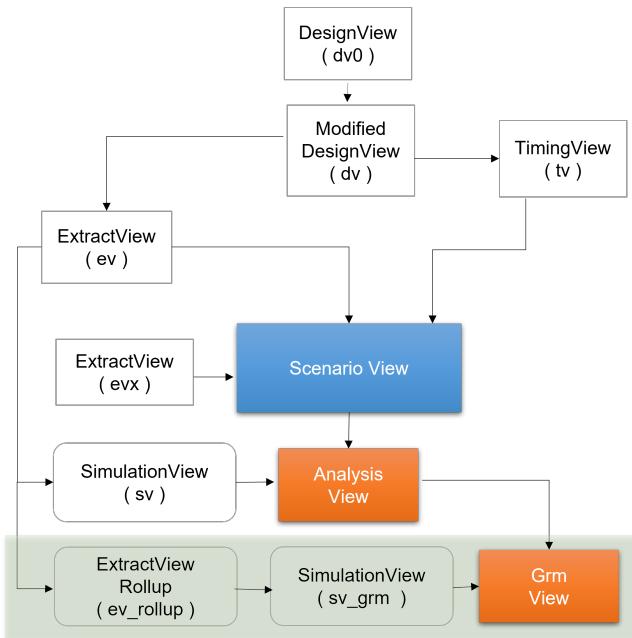


10.2. Creating CPM Using Single Worker

A Chip Power Model (CPM) is a reduced order model that represents the chip power delivery network (PDN) for package and printed circuit board (PCB) simulation. The RedHawk-SC tool provides two commands to create GrmView (generic reduced model view) that stores the CPM information. You query the GrmView to obtain CPM outputs and save them as SPICE and other format files.

- To create GrmView with a single worker, use the `create_chip_power_model_view` command.
- To create GrmView with multiple workers (distributed execution), use the `perform_chip_power_model_generation` command.

The following figure shows the CPM generation flow when you use a single worker to create the GrmView.

Figure 61. Typical CPM Generation Flow With Single Worker

The following topics describe CPM creation with a single worker based on this flow.

- [Prerequisites to Generate GrmView With Single Worker](#) on page 312
- [Generating GrmView With Single Worker](#) on page 313
- [CPM Script Example](#) on page 320

10.2.1. Prerequisites to Generate GrmView With Single Worker

To generate a CPM with a single worker, you might need to use metal rollup techniques during extraction as shown in [Figure 61: Typical CPM Generation Flow With Single Worker](#) on page 312. This reduces the node count for faster turn around time (TAT) without significant loss of accuracy in large designs. Further, you might require a worker with large memory.

The following sections describe the prerequisites to generate a GrmView with a single worker:

- [Setting up the Worker](#) on page 312
- [Setting Rollup](#) on page 313
- [Setting the SimulationView](#) on page 313

Setting up the Worker

For large designs, generating a CPM using a single worker might consume large memory. You must launch a worker with sufficient memory to create CPM. The following example shows how to create a launcher with 250 GB memory for a large design.

```

big_launcher = create_grid_launcher('big_launcher', 'qsub -V -b y -cwd-j y -q
                                    ae_perf -l mfree = 250G -o uge.log2)
big_launcher.set_jobs(['cpm.write_spice_deck*', 'cpm.run_asim_power_model*'])
big_launcher.launch(1)

```

Setting Rollup

To create CPMs for large designs using a single worker, you might need to use rollup techniques to reduce node count and memory usage without significant loss of accuracy. To use metal rollup, use the `rollup` key with the `create_extract_view` command as shown:

```
import ev_utils
ev_rollup=db.create_extract_view(..., settings =
{'reduction' : {'rollup':ev_utils.create_rollup_settings(dv,keep_metals=2) }})
```

`keep_metals = 2` keeps only the top two layers of the design and rolls up all other layers. To retain the metal-insulator-metal (MiM) structure in MiM capacitor designs, use `keep_metals = 5`.

Setting the SimulationView

You can use an existing SimulationView or create a new one with '`sim_type`' : '`cpm`':

```
sv_grm = db.create_simulation_view(ev_rollup, options=options,
settings={'sim_type' : 'cpm'}, tag='sv_grm',...)
```

10.2.2. Generating GrmView With Single Worker

To create a generic reduced model view (GrmView) by using a single worker, use the `create_chip_power_model_view` command:

```
gv = db.create_chip_power_model_view
(sv_grm,analysis_view = av_dynamic,options=options,tag='gv',cpm_nx = 1, cpm_ny = 2)
```

The command computes the reduced order model of the chip PG grid and saves the results in the GrmView. For more details, see `help(SeaScapeDB.create_chip_power_model_view)`.

Note: It is not possible to reuse an AnalysisView with package in the related SimulationView. You must ensure that a run with package is not used to generate a CPM. AnalysisView is required to get bump currents and decoupling capacitance values.

The following sections describe the methods to enable different configurations in GrmView:

- [Enabling Coupled Simulation](#) on page 313
- [Using Adaptive Frequency Sweep in AC Mode](#) on page 314
- [Generating CPM Without Current Conservation](#) on page 315
- [Specifying Parasitic Model Type](#) on page 315
- [Probing Internal Nodes](#) on page 316
- [Customizing Frequency Range for AC Analysis](#) on page 317
- [Creating CPM Ports](#) on page 317

Enabling Coupled Simulation

By default, the RedHawk-SC CPM solver considers voltage domain circuits that behave in a decoupled fashion. The following example shows a circuit with two different voltage domains, VCCO_11 and VCCO_22.

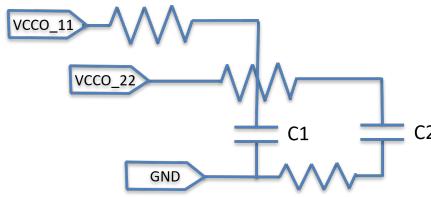
Figure 62. A Circuit With Two Voltage Domains

Figure 63: Decoupled Circuit Example on page 314 shows the decoupled voltage domains of Figure 62: A Circuit With Two Voltage Domains on page 314. The equivalent circuit to compute the input impedance between VCCO_11 and GND has high capacitance at the GND terminal.

In general, a ground terminal of a decoupled circuit includes capacitors corresponding to other voltage domains. This approximation is good for single PG domain chips with symmetric packages.

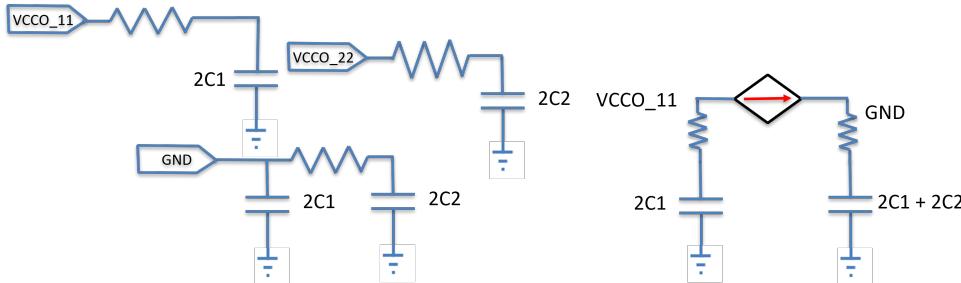
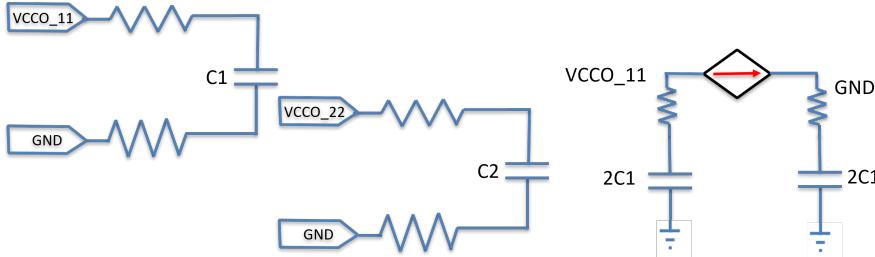
Figure 63. Decoupled Circuit Example

Figure 64: Coupled Circuit Example on page 314 shows the coupled circuit of Figure 62: A Circuit With Two Voltage Domains on page 314 where each voltage domain is coupled to its corresponding ground domain. The equivalent circuit to compute the input impedance between VCCO_11 and GND results in the same capacitance value as in the original circuit.

When you enable coupled simulation, the CPM solver considers the coupling capacitors between respective domains. This approximation is good for multiple PG domains and asymmetric packages. Coupled simulation typically takes more run time.

Figure 64. Coupled Circuit Example

To enable coupled simulation, set the `coupled_simulation` argument of the `create_chip_power_model_view` command to `True`. The default is `False`.

Using Adaptive Frequency Sweep in AC Mode

By default, the tool does not use the adaptive frequency sweep (AFS) for AC mode CPM generation.

To use the AFS function in CPM generation, set the `cpm_use_afs` argument of the `create_chip_power_model_view` command to `True`. The tool intelligently selects only seven to nine

frequencies (sufficient to reach convergence) for sampling rather than using a total of 26 frequency samples in the default AC mode.

This setting is recommended when the design size is less than 50 ports in the output CPM. You can estimate the port count by computing $m \times n_x \times n_y$, where m is the number of power nets or ground nets of the chip, n_x is the number of x-partitions, and n_y is the number of y-partitions. For more details about creating partitions in CPM, see [Creating CPM Ports Per Partition](#) on page 319.

Generating CPM Without Current Conservation

By default, current is conserved (power and ground current values are balanced) during CPM generation. However, there can be cases where power and ground currents are unbalanced and have slightly different values, for example, in RedHawk-SC dynamic DVD analysis.

To generate the CPM model without current conservation to achieve better correlation with simulation results, set the `cpm_pincurrent` argument of the `create_chip_power_model_view` command to `False`. The default is `True`.

The following table shows the effect of `cpm_pincurrent` settings on the dynamic current ($I(t)$) and the passive portion of the output CPM, and that for dynamic current, the `cpm_pincurrent` settings override the `coupled_simulation` settings in the output CPM.

cpm_pincurrent	coupled_simulation	$I(t)$	Passive Portion
False	False (default)	Current from N-1 ports flow to one common VSS port. Note: N is the total number of CPM ports.	Capacitance connected to Spice node 0. Not affected by these options.
False	True	Current from N-1 ports flow to one common VSS port.	Capacitance connected to Spice node 0. Not affected by these options.
True (default)	False (default)	Current for all N ports flow to Spice node 0.	Capacitance connected to Spice node 0. Not affected by these options.
True (default)	True	Current for all N ports flow to Spice node 0.	Capacitance connected to Spice node 0. Not affected by these options.

Specifying Parasitic Model Type

By default, the RLC parasitics from power and ground terminals are connected to SPICE global ground (node 0). To specify the type of parasitic model in CPM without SPICE node 0, set the `cpm_noglobal_gnd` argument of the `create_chip_power_model_view` to `True`.

When set to `True`, there is a direct connection between the power and ground ports without going through SPICE node 0.

The following table shows the effect of `cpm_noglobal_gnd` settings on the dynamic current ($I(t)$) and the passive portion of the output CPM, and that the `cpm_noglobal_gnd=True` settings override the `coupled_simulation` settings in the output CPM.

cpm_noglobal_gnd	coupled_simulation	$I(t)$	Passive Portion
False (default)	False (default)	Not affected by these options. Controlled by <code>cpm_pcurrent</code> .	Capacitance connected to Spice node 0 (global ground).
False (default)	True	Not affected by these options. Controlled by <code>cpm_pcurrent</code> .	Capacitance connected to Spice node 0 (global ground).
True	False (default)	Not affected by these options. Controlled by <code>cpm_pcurrent</code> .	Coupled between Vdd and Vss.
True	True	Not affected by these options. Controlled by <code>cpm_pcurrent</code> .	Coupled between Vdd and Vss.

Note: When you generate the CPM for use with S-parameter package models, setting `coupled_simulation=True` is recommended and `cpm_noglobal_gnd=True` is required.

Computing C_{die} and R_{die} for Chip

To obtain the equivalent C_{die} and R_{die} value for the chip, set the `cpm_cdie` argument of the `create_chip_power_model_view` command to `True`. This setting connects all ports of same net together to compute a single C_{die} and a single R_{die} for the entire chip. Default is `False`.

To query this information, use the `get_chip_power_model_cdie_info` function. See [Reporting Die Parasitics](#) on page 326.

Probing Internal Nodes

The RedHawk-SC tool can generate CPMs with terminals at user-defined device locations in addition to the traditional C4 bump and pad locations.

This capability exposes user-defined probes as external ports and enables you to probe critical device locations inside the chip, such as near PLL or FGU blocks. This is useful to evaluate the drop through the on-die PDN in fast system-level simulations.

Create the probes in ExtractView and then set the `cpm_probes` argument to `True` with the `create_chip_power_model_view` command:

```
def create_user_probes(dv):
    user_probes = [{"net":Net('VDD'), 'layer':Layer('M1'),
                    'coord':RealCoord(49.,3), 'name':'Prb1'},
                   {'net':Net('VSS'), 'layer':Layer('M3'),
                    'coord':RealCoord(49,6.5), 'name':'Prb2'}]
    return user_probes
ev = db.create_extract_view
```

```
(dv, tech_view = nv, options=options, probes=create_user_probes(dv))
gv = db.create_chip_power_model_view
(sv_grm, analysis_view=av, options=options, cpm_probes=True)
```

Customizing Frequency Range for AC Analysis

By default, the range of frequency domain analysis using adaptive frequency sweep (AFS) is from 10 MHz to 2.5e+09 Hz.

To use a different range of frequencies, define the range in a file and load the file by specifying the file name with the `cpm_options_file` argument of the `create_chip_power_model_view` command.

The following example shows how to define a custom frequency range in a text file.

```
# specifies the lowest frequency when using log grid default is 10MHz
fmin=1e3
# specifies the highest frequency for AC simulation default is 2.5GHz
fmax=5.0e9
```

10.2.3. Creating CPM Ports

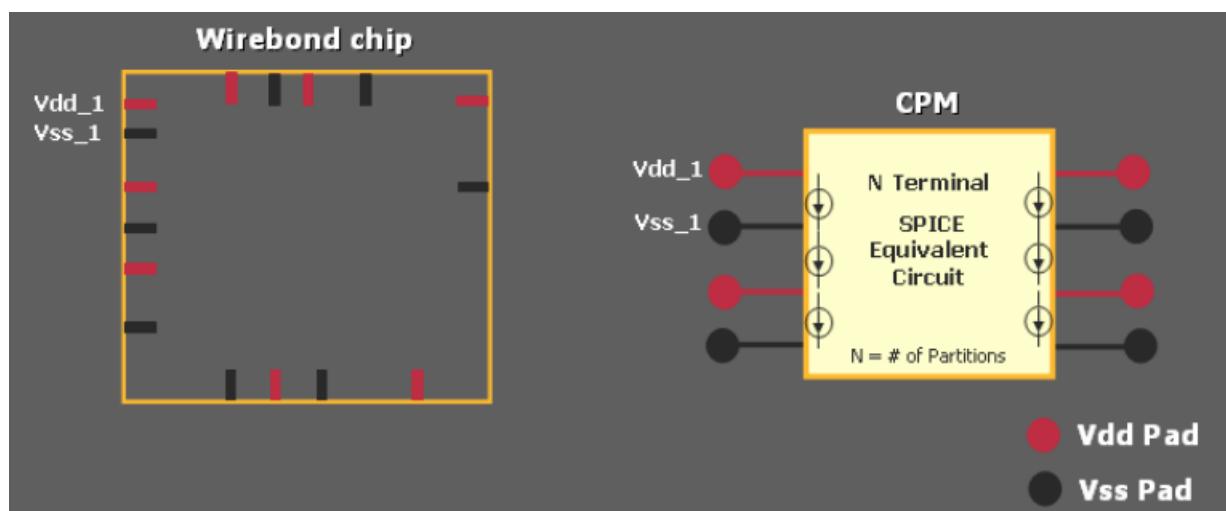
The RedHawk-SC CPM creation engine enables you to use multiple methods to map chip pad locations to create and group ports in the output CPM. This section describes the following methods to create CPM ports:

- [Creating CPM Ports Per Pad Location](#) on page 317
- [Creating CPM Ports Per Group Specified in PLOC File](#) on page 318
- [Mapping Bumps to SPICE Ports](#) on page 318
- [Creating CPM Ports Per Partition](#) on page 319

Creating CPM Ports Per Pad Location

By default, the tool creates a port in the output CPM file for each pad location of the design when you specify the `cpm_plocname` argument with the `create_chip_power_model_view` command.

This is suitable for wirebond designs with a small number of power and ground pads. The tool creates an N-terminal CPM with each terminal of the model corresponding to a wire-bond pad, as shown in the following figure:



Creating CPM Ports Per Group Specified in PLOC File

For wirebond designs with a large number of pads, the tool groups the pads based on the group names specified in the PLOC file input to DesignView. Typically, a PLOC file has five columns. You can assign a group name to multiple PLOCs in the sixth column, as shown in the following snippet example.

```
#source_name #loc_x #loc_y #layer_name #POWER/GROUND #group_name
DVDD1      10      5     METAL6    POWER      GROUP_POWER1
DVDD2      15      5     METAL6    POWER      GROUP_POWER1
...
DVSS1      10     15     METAL6   GROUND     GROUP_GROUND1
DVSS2      15     15     METAL6   GROUND     GROUP_GROUND1
...
```

Note: These group names are also called SPICE package node names, as this mechanism is used to connect a package by using the PACKAGE_SPICE_SUBCKT keyword.

To create a set of CPM ports that correspond to the PLOC groups, use the `cpm_plocname` argument with the `create_chip_power_model_view` command.

This enables you to create individual CPM models for different blocks or sub-systems of the chip. Package and board designers can divide and customize a CPM for multiple sub-system simulations, each targeting a specific operating mode or area of the chip.

The tool outputs the current signature of each PLOC group in the CPM. This is useful to examine the impact of different blocks of the chip and their combined activity for DVD and EMI debug. The following example shows an output CPM file snippet with current signatures of different groups defined in the PLOC file.

```
I_group1_cursig p1 p2 pwl(+0.000000ps 0.181292
...
+)
I_group2_cursig p1 p2 pwl(+0.000000ps 0.181292
...
+)
I_others_cursig p1 p2 pwl(+0.000000ps 0.155827
...
+)
```

Note: If the `cpm_plocname` argument is used with `cpm_nx` and `cpm_ny` arguments, the node names are generated in the following form: PAR_0_0_VDD1, PAR_0_0_VSS2, and so on.

Mapping Bumps to SPICE Ports

If the chip pad to CPM port map is not provided in the sixth column of the PLOC file, you can use the `cpm_port_map` argument with the `create_chip_power_model_view` command to call the `parse_pad_file_func` function from the RedHawk-SC package module. This function parses a user-defined map file that contains the bump to CPM port map data, as shown in the following example:

```
import package
gv = create_chip_power_model_view(cpm_port_map=package.parse_pad_file_func
                                  (pad_file_name = 'Die2PackageMapping.txt'),
...)
```

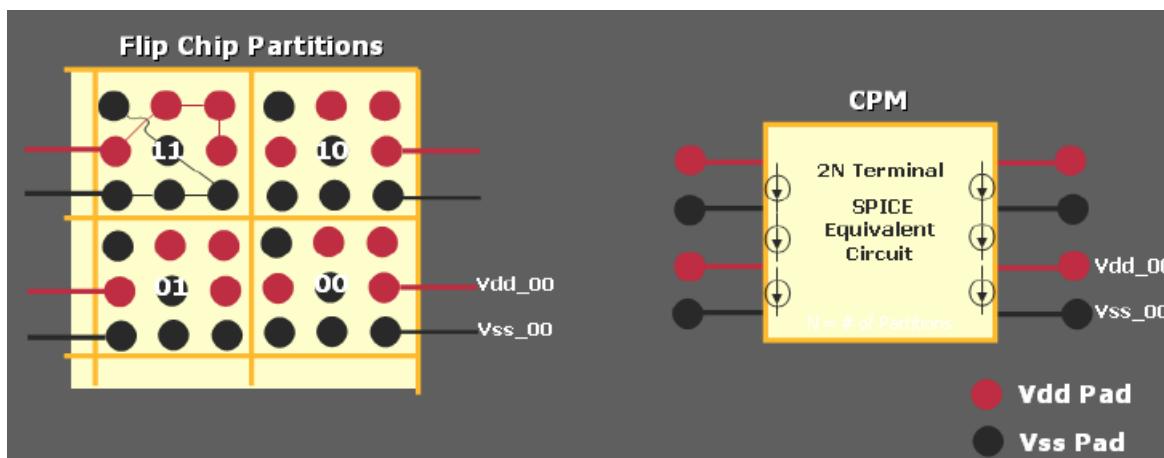
Content of 'Die2PackageMapping.txt':

```
* Bump-name Dont-care Spice-port-name
VDD_p1    1      VDD_spc1
VDD_p2    2      VDD_spc1
VDD_p3    3      VDD_p3
VSS_p1    4      VSS_spc1
VSS_p2    5      VSS_spc1
```

With the setting, the created CPM SPICE netlist has SPICE port names instead of the bump names.

Creating CPM Ports Per Partition

For flip chip package designs with a large number of pad locations, you can group the pad locations and create a CPM with lesser number of ports. To do so, partition the chip bump area into $n_x \times n_y$ tiles. If the chip has m power nets and m ground nets, the maximum number of external terminals of the resulting CPM is $m \times n_x \times n_y$. The following figure shows a flip chip example with $n_x=2$, $n_y=2$, and $m=4$, that is, the total number of partitions is 4 and each partition has one VDD net and one VSS net.



To define the number of partitions in x and y directions, use the `cpm_nx` and `cpm_ny` arguments with the `create_chip_power_model_view` command. The default for both `cpm_nx` and `cpm_ny` is 0, that is, the tool does not group any pads (similar to wirebond designs).

Note: For the `cpm_nx` and `cpm_ny` setting to work, it is a prerequisite to set the `net_based` key to `False` under `parse_ploc_func` with the `create_modified_design_view` command.

```
dv = db.create_modified_design_view
(dv0, eco_commands=package.parse_ploc_func(th_ploc, net_based=False), tag='dv')
```

Otherwise, the tool by default groups and creates a single terminal for all the pad locations of each voltage domain in the output CPM.

Creating CPM Ports Per Region and Layer

You can also group power and ground bumps on a layer or a region of the chip to create CPM ports. To specify a region on a certain layer of the chip and its partitions, use the group configuration file with the following file format:

```
CPM_Port_Grouping {
#region_name layer nx ny llx lly urx ury_in_um
...
}
```

The following example shows the typical contents of a group configuration file. The tool reads this file to create CPM ports. For example, the tool breaks the DIE1_G region (specified with coordinates) on PM0 layer into 20 X 20 partitions and assigns ports for each partition.

```
CPM_Port_Grouping {
#region_name layer nx ny llx lly urx ury_in_um
DIE1_G PM0 20 20 3000.00 -4700.00 10000.00 4800.00
DIE2_G PM0 30 30 3000.00 -16100.00 10000.00 -6400.00
BGA_G UBM 20 40 -2500.00 -30000.00 15000.00 5000.00
}
```

To read the group configuration file for CPM port grouping, specify the file name with the `cpm_port_grouping_file` argument with the `create_chip_power_model_view` command.

10.2.4. CPM Script Example

The following example shows a typical script to generate and query GrmViews, and then output CPM files from the view.

```
ev_rollup_args = dict(
    settings={
        'extract_temperature' : 25,
        # Top-2 rollup is used here
        'reduction' : {'rollup' : ev_utils.create_rollup_settings(dv,
keep_metals=2)}
    }
    tag='ev_rollup',
    options=options)

sv_grm_args = dict(
    options=options,
    settings={'sim_type' : 'cpm'},
    tag='sv_grm')

gv_args = dict(
    options=options,
    tag='gv',
    cpm_nx = 2 ,
    cpm_ny = 2 , )

gv_coupled_args = dict(
    options=options,
    tag='gv_coupled',
    # Turns on coupled CPM generation
    coupled_simulation = True,
    For 2x2 CPM
    cpm_nx = 2 ,
```

```

cpm_ny = 2 ,
)

sv_wo_pkg = db.create_simulation_view(ev, package=None, **sv_wo_pkg_args)
av_dynamic_wo_pkg = db.create_analysis_view
(sv_wo_pkg, scn_no_prop_vectorless,
**av_dynamic_wo_pkg_args)

ev_rollup = db.create_extract_view(design_view = dv,tech_view=nv,
**ev_rollup_args)
sv_grm = db.create_simulation_view(extract_view=ev_rollup,**sv_grm_args)

# For Decoupled CPM
gv = db.create_chip_power_model_view
(simulation_view=sv_grm,analysis_view = av_dynamic_wo_pkg,**gv_args)

# For Coupled CPM
gv_couple = db.create_chip_power_model_view
(simulation_view=sv_grm,analysis_view =
av_dynamic_wo_pkg,**gv_couple_args)

current_spice, passive_spice = gv.get_chip_power_model_in_cpm_spice()
write_to_file('Chip_Power_Model.sp', current_spice)
write_to_file('Chip_Power_Model_SubCircuit.sp.inc', passive_spice)
write_to_file('perform_ac.sp',gv.get_chip_power_model_perform_ac_spice())

```

10.3. Creating CPM Using Multiple Workers

The Chip Power Model (CPM) generation flow for distributed execution (uses multiple workers) and has similar accuracy and significant performance improvement over the single-worker CPM flow. To enable the multi-worker flow, use the `perform_chip_power_model_generation` command that generates a CPM by using the `SimulationView` and the `ScenarioView`.

```
gv = db.perform_chip_power_model_generation
(simulation_view, scenario_view, settings, options, tag='grm_view')
```

`simulation_view` is the `SimulationView` name and is a mandatory argument. The `SimulationView` might have `sim_type : 'cpm'` as in single-worker CPM flow. For example,

```
sv = db.create_simulation_view
(ev, options=options, settings={'sim_type' : 'cpm'}, tag='sv')
```

`scenario_view` is the `ScenarioView` name and is a mandatory argument.

`settings` is a dict containing both `AnalysisView` and CPM-generation related settings. This is a mandatory argument and has two keys:

- `analysis_view_args`: Contains `AnalysisView`-related settings and CPM-related settings.
- `cpm_args`: Contains CPM-related settings and arguments.

The following example shows how to use the `settings` dict with the `perform_chip_power_model_generation` command:

```
settings = dict()
settings['analysis_view_args'] =
{'time_step': <step_size>, 'duration': <dynamic_simulation_duration>}
initial_condition = 't0' and
```

```

        default current_derate_mode

# Pass CPM view arguments here, Keys of this dict accepts only CPM view args.
settings['cpm_args'] = {'cpm_debug':True|False, 'cpm_nx': <x_grid>,
                      'cpm_ny':<y_grid>, ...}

```

The following is a typical example showing how to generate a CPM by using the `perform_chip_power_model_generation` command:

```

settings = dict()

settings['analysis_view_args'] = {'time_step': 50e-12, 'duration': 10e-9}

settings['cpm_args'] =
{'cpm_debug':True, 'cpm_nx': 1, 'cpm_ny':1}

# Assuming this DB is run up to SCN.
# The SCN is passed to the perform_chip_power_model_generation command.
db0 = open_db('db')

db = open_db('db_cpm')

sv = db.create_simulation_view(ev, options=options, settings={'sim_type':'cpm'},
tag='sv')

scn = db0.get('scn')

gv = db.perform_chip_power_model_generation(simulation_view=sv,
scenario_view=scn,
settings=settings, options=options, tag='gv')

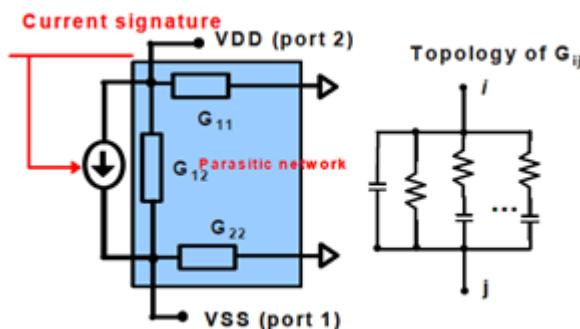
```

10.4. Storing CPM Output

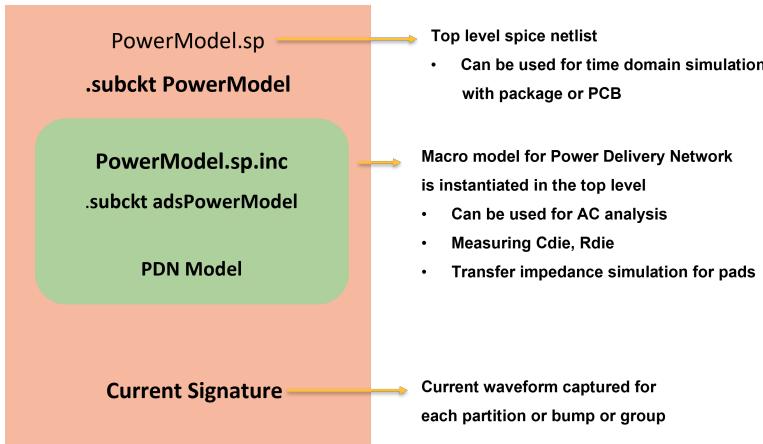
The main output of RedHawk-SC CPM simulation is a hierarchical SPICE netlist that includes piecewise linear switching current signature, as well as the parasitic reduced-order model of the die power delivery network (PDN).

The current signatures are connected to the parasitic model of the die PDN. The parasitic model is contained in a different SPICE netlist file (.inc file). The following figure shows a two-port equivalent model:

Figure 65. Two Port CPM equivalent circuit



The following figure shows the main components of the output CPM including the passive part of the CPM (*.sp.inc) and the current signatures by partition (*.sp).

Figure 66. Hierarchical Structure of CPM Output File

The complexity of the SPICE netlist is determined by the number of partitions over which the CPM model is created.

Outputting CPM Models

To obtain the parasitic and current CPM files from the GrmView, use the `get_chip_power_model_in_cpm_spice` query command as shown in the following example:

```

current_spice, passive_spice = gv.get_chip_power_model_in_cpm_spice()
write_to_file('Chip_Power_Model.sp', current_spice)
write_to_file('Chip_Power_Model_SubCircuit.sp.inc', passive_spice)

```

When you use the `write_to_file` command, the tool outputs the parasitic file and the current file in respective naming formats, `<design_name>.sp.inc` and `<design_name>.sp`.

The following is a typical example of the parasitic file.

```

.SUBCKT PowerModel p1 p2 p3 p4 p5
+ p6 p7 p8
*****
* Ansys RedHawk-SC Chip Power Model [Accurate RC reduction]
* Model Subcircuit of Die PDN
* Copyright (c) 2002-2020 ANSYS, Inc.
*****

* Pad name : port name : net : port id
* VDD_56   VDD_56   VDD      p1
* VDD_160  VDD_160  VDD      p1
* VDD_38   VDD_38   VDD      p1
* VDD_159  VDD_159  VDD      p1
* VDD_158  VDD_158  VDD      p1
* VDD_157  VDD_157  VDD      p1
* VDD_4    VDD_4    VDD      p1
* VDD_5    VDD_5    VDD      p1

```

Die Power Delivery Network (PDN)
Model Subckt

The following example is a simple CPM current file generated for a die with one VDD and one VSS net. The PDN subcircuit file is instantiated in this top-level SPICE (.sp) file.

The Chip Package Protocol (CPP) section contains the mapping information of ports and bumps for other tools to automatically connect to the package model. This section lists the name of the die pad, x and y locations of the pad, the corresponding SPICE node name, as well as the partition and net information of the pad.

The CPM current file also includes average power, average current, and time domain current information (starting with the `Icursig` syntax) for each port. The current signature is in piece-wise linear (PWL) format.

```

.INCLUDE ".Chip_Power_Model_SubCircuit.sp.inc"                                     → PDN subckt included

* Partitioning of flip chip bump pad area - Die Top View
* -----
* | (0 Ny) | (1 Ny) | .... | (Nx Ny) |
* -----
* | (0 ..) | (1 ..) | .... | (Nx ..) |
* -----
* | (0 1) | (1 1) | .... | (Nx 1) |
* -----
* | (0 0) | (1 0) | .... | (Nx 0) |
* -----
* Begin Chip Package Protocol --->
* generated by asim_power_model                                                 → CPP mapping

* Start Units
* Length um
* End Units
* VDD_56   : (212.000000 154.789993)   : p1  = PAR_0_0_VDD
* VDD_160  : (580.000000 154.789993)   : p1  = PAR_0_0_VDD
* ...
* VSS_37   : (166.000000 514.799988)   : p8  = PAR_0_0_VSS
* End_Chip Package Protocol <-->

.subckt adsPowerModel                                         → CPM subckt
+ p1 p2 p3 p4 p5 p6 p7 p8

Xpdn
+ p1 p2 p3 p4 p5 p6 p7 p8                                     → PDN instantiation
+ PowerModel

* CPM Port Name | Average current (A) | Max. magnitude of current | Net voltage *
* p1 0.0483909 0.352188 1.1
* ...
* p8 -0.0396821 0.291764 0
* Average power = 0.21358 W.

Icursig1 p1 0 pwl(                                         → Current signature
+ 0.000000ps 0.13978214
+ 29.999999ps 0.15121435
+ 59.999998ps 0.13539097
+ 90.000001ps 0.14537421
+ 119.999999ps 0.15520955

```

Performing AC Analysis

To obtain the SPICE netlist for AC analysis from the GrmView, use the `get_chip_power_model_perform_ac_spice` query command as shown in the following example:

```
write to file('perform ac.sp',gv.get chip power model perform ac spice())
```

The `write_to_file` command outputs the `perform_ac.sp` file. The following is a typical example of the `perform_ac.sp` file.

```

* Calculate impedance(zin) between nets VDD and VSS
* by the AC analysis of the passive part of the CPM model.
* For use with nspice simulator. Real and imaginary parts of zin are plotted as function of freq
* For RC grids Rdie(freq) and Cdie(freq) plots also created
.include "Chip_Power_Model_SubCircuit.sp.inc"
Xdie VDD VDD VSS VDD VSS
+ VDD VSS VSS  PowerModel
Rlarge1 VDD 0 1e9
Rlarge2 VSS 0 1e9
Isrc VSS VDD AC 1.0
.ac dec 50 5e7 2.5e9
* Set the frequency you want to calculate impedance at below:
.param freq=50e6
.meas ac real_v find vr(VDD, VSS) at=freq
.meas ac imag_v find vi(VDD, VSS) at=freq
.meas ac Re_Zin param='real_v'
.meas ac Im_Zin param='imag_v'
.option probe post
.probe Im_Zin_freq=par('vi(VDD, VSS)')
.probe Re_Zin_freq=par('vr(VDD, VSS)')
.probe Cdie freq=par('-1.0/(2*3.14159265*hertz*vi(VDD, VSS))')
.probe Rdie_freq=par('vr(VDD, VSS)')
.end

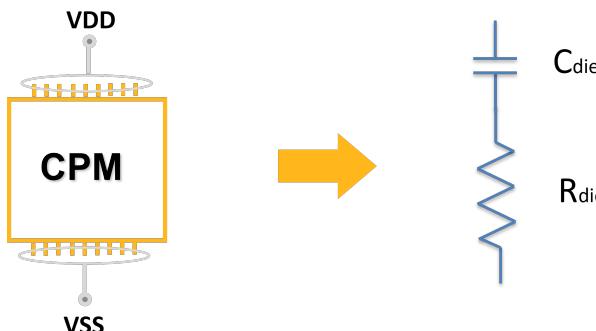
```

By default, the output file references and instantiates the parasitic file, Chip_Power_Model_SubCircuit.sp.inc. If the file name of your parasitic CPM is different, you must change the default parasitic name.

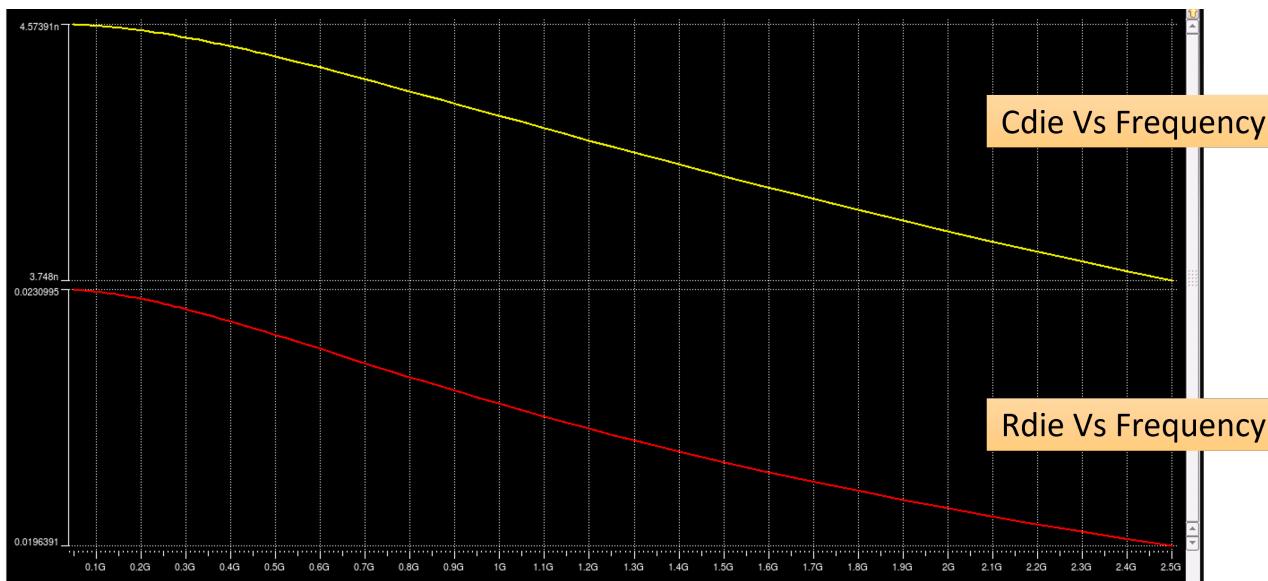
The file includes an AC current source between VDD and VSS for a frequency analysis from 50 MHz to 2.5 GHz. The next section measures the input impedance (Z_{in}) between VDD and VSS, and then computes R_{die} and C_{die} from the real and imaginary components of impedance:

$$Z_{in} = R_{die} + 1/(j\omega C_{die})$$

$\omega = 2\pi f$ where f is the frequency at which the chip input impedance is measured.



You can use the SPICE netlist to determine R_{die} and C_{die} at the specified frequency (50 MHz by default, unless you modify it in the get_cdie.sp file), and to plot C_{die} and R_{die} for frequencies ranging from 1 MHz to 1 GHz. The C_{die} vs frequency plot enables a package designer to determine the C_{die} value at the dominating operating frequency of the chip.



Using this file eliminates user errors, which are likely for a CPM model with a large number of ports, and saves time that is needed to manually create the netlist.

For designs with multiple power or ground nets, the tool generates a separate output file for each pair of nets. If there are only two nets, the file name is **get_cdiele.sp**. Otherwise, file names of the form **get_cdiele_<pwr_net>_<gnd_net>.sp** are written to the work directory.

Reporting Die Parasitics

To obtain the equivalent C_{die} and R_{die} value for the chip from the GrmView, use the `get_chip_power_model_cdiele_info` query command as shown in the following example:

```
cdiele_file = gv.get_chip_power_model_cdiele_info()
write_to_file(test+'.cdiele',cdiele_file)
```

When you run the `create_chip_power_model_view` command with `cpm_cdiele=True`, or `cpm_nx=1` and `cpm_ny=1`, the tool generates a *.Cdie file in the `<workdir>/adsRpt/CPM/apache.Cdie` directory. The file contains the effective capacitance and effective series resistance of the die PDN at different frequencies. This is useful to estimate die parasitics.

The following is an example of a *.Cdie file:

```
Cdiele and Rdiele between nets VDD and VSS
4.700000e+08 Hz, Cdiele=2.662143e-11 F, Rdiele=2.902836e+00 Ohm
6.250000e+08 Hz, Cdiele=2.635451e-11 F, Rdiele=2.874828e+00 Ohm
9.350000e+08 Hz, Cdiele=2.571951e-11 F, Rdiele=2.811440e+00 Ohm
1.250000e+09 Hz, Cdiele=2.501593e-11 F, Rdiele=2.746562e+00 Ohm
1.875000e+09 Hz, Cdiele=2.362147e-11 F, Rdiele=2.634761e+00 Ohm
2.500000e+09 Hz, Cdiele=2.229990e-11 F, Rdiele=2.548309e+00 Ohm
```

Note: The *.Cdie file reports results only at frequencies where the Y-matrix calculation is performed. The tool does not rely on an equivalent circuit to calculate R_{die} and C_{die} .

Reporting Pad Port Information

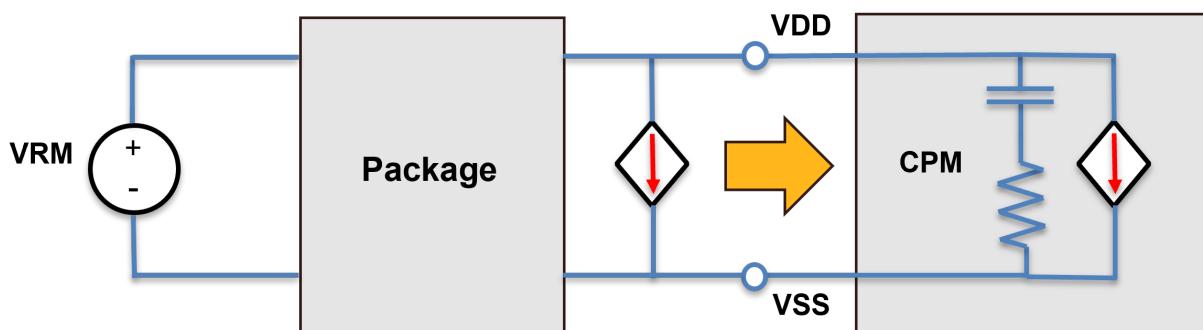
To obtain the pad grouping information when the `cpm_port_grouping_file` argument is specified, use the `get_chip_power_model_pad_port_info` query command as shown in the following example:

```
write_to_file('pad_port_info.txt', gv.get_chip_power_model_pad_port_info())
```

This outputs a text file named `Pad_port_info.txt` containing pad grouping information.

10.5. Validating CPM

The following figure represents a CPM connected to the package to measure the system response. The SPICE netlist example following the figure shows how the package and CPM are connected through a current source to perform an AC analysis and then measure the input impedance.



```
* Calculate impedance(Zin) between nets VDD and VSS
.include "CPM.inc"                                CPM
Xdie VDD VSS PowerModel

.inc 'redhawk_wrap_0p85v.sp'                      Package
Xpkg VDD VSS REDHAWK_PKG

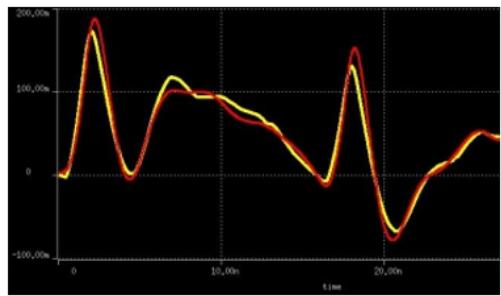
Isrc VDD VSS AC 1.0
.ac dec 50 1e6 5e9

.option probe post
.probe Zin_freq=par('sqrt((vi(VDD,VSS)*vi(VDD,VSS))+(vr(VDD,VSS)*vr(VDD,VSS)))')
.probe Im_Zin_freq=par('vi(VDD,VSS)')
.probe Re_Zin_freq=par('vr(VDD,VSS)')
.end
```

Using this netlist, you can run a SPICE simulation with CPM representation of the die and package SPICE model. Measure the current at the connection points (die pads) of the CPM model to the package model. Similarly, you can input the package SPICE model to RedHawk-SC and perform a full flat analysis, and then compare the measured current profiles.

The following figure shows the typical current waveforms. The red current profile is the bump current measured from a RedHawk-SC run, while the yellow current profile is the bump current measured from the SPICE run with the CPM model. The close match between these two waveforms validates CPM.

Figure 67. Current Profiles for CPM Versus RedHawk-SC Full-Chip Simulation



11: Multichip Analysis

A three-dimensional integrated circuit (3DIC) is an emerging technology that can overcome the limitations of Moore's law. Multiple silicon wafers or chips are stacked and interconnected using structures such as through-silicon vias (TSVs), to enable them to function as a single device. This meets requirements, such as higher performance, lower power consumption, reduced area, smaller form factor, improved inter-chip bandwidth causing faster data exchange.

A three-dimensional die or chip stack that uses through-silicon vias (TSVs) provides the flexibility to connect chips with different functions (memory, processor, power management) and fabricated using different processes in the same package. The chip stack has a significantly closer form than other multichip packages, such as, multi chip modules (MCMs) and 3DICs that do not have TSVs.

With 3DIC, power delivery network (PDN) design challenges increase in ensuring power integrity and thermal integrity. For example, power noise to the top chip is now due to both switching within the chip and noise from the bottom chip. PDN of both the chips are not mutually exclusive. The count, design, and placement of TSVs also play a key role in ensuring power integrity and thermal integrity.

Comprehensive chip, package and system co-design and co-analysis is necessary, that is, multichip analysis is necessary.

This chapter describes the following topics to perform multichip analysis by using the RedHawk-SC tool:

- [Multichip Concepts](#) on page 329
- [Multichip Flow Overview](#) on page 331
- [Design Data Requirements](#) on page 332
- [Creating Multichip Configuration File Using Setup Utility](#) on page 334
- [Creating Command File for Multichip Analysis](#) on page 351
- [Multichip Analysis Flows](#) on page 354
- [Multichip-Specific GUI Features](#) on page 364
- [Reporting Multichip Analysis Results](#) on page 366
- [SPR for Multichip Designs](#) on page 369

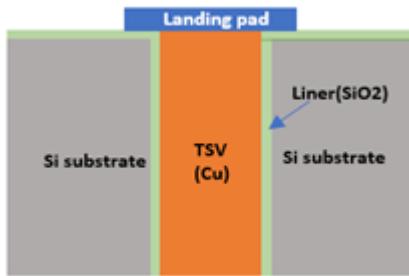
11.1. Multichip Concepts

This section describes the following multichip concepts:

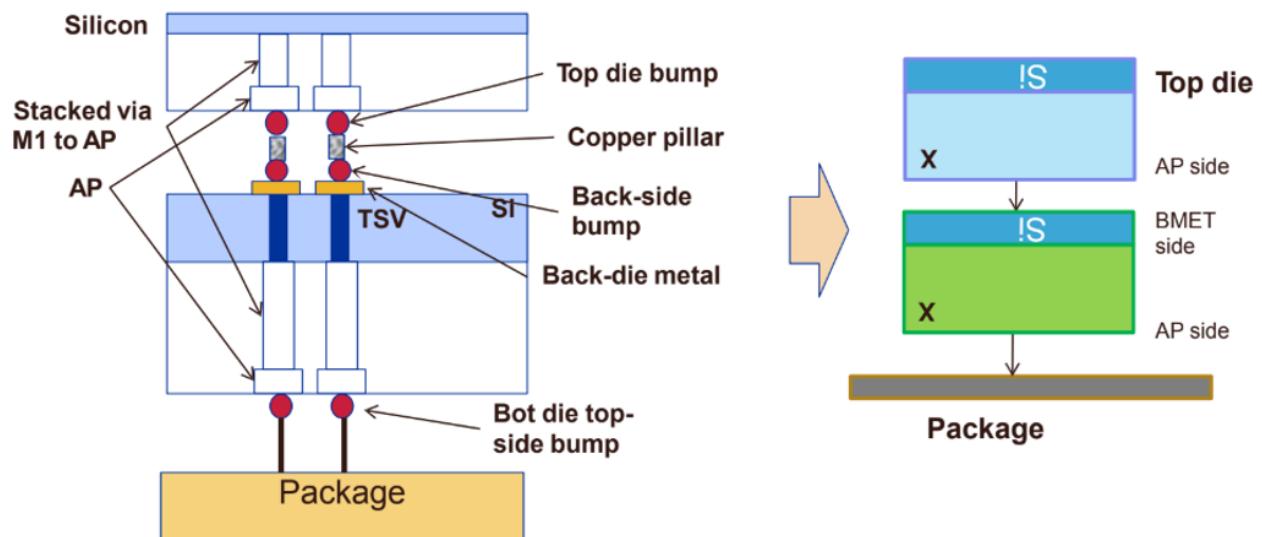
- [Through Silicon Via \(TSV\)](#) on page 329
- [Difference between 2.5DIC and 3DIC](#) on page 330
- [Need for 3DIC](#) on page 330

Through Silicon Via (TSV)

A through-silicon via is a vertical via connection that passes completely through a silicon chip or wafer to enable chip stacking. These are the building blocks that enable three-dimensional integration of chips. The main advantage of TSV interconnect is the shortened path for the signal to travel from one chip to the next, or one layer of circuitry to the next.



A copper pillar is an interconnect, used specifically in flip-chip technology, to connect two micro bumps on the surface of a chip or of different chips.

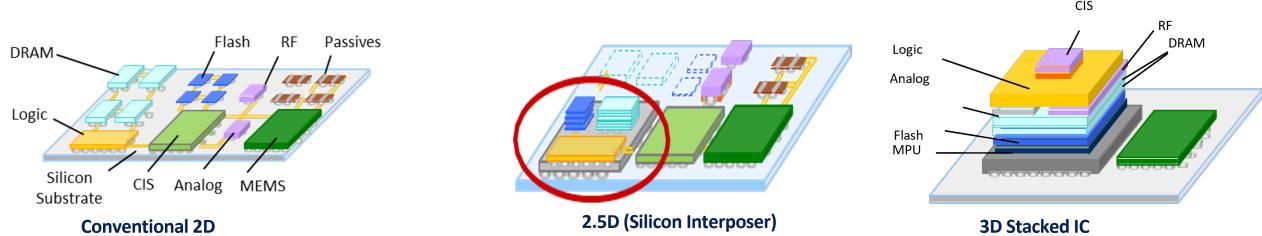


Difference between 2.5DIC and 3DIC

2.5DIC is a configuration where multiple chips are mounted horizontally on a structure called interposer. Communication between the chips is possible through the interposer, which increases the bandwidth. A chip is not stacked on another chip, but chips stacked are on the interposer.

3DIC is a configuration of either 3D stacked chips where multiple chips are stacked one over another vertically and connected through TSVs, or monolithic 3DICs with multiple device layers stacked together to form a single chip that might use TSVs for interconnection.

Need for 3DIC



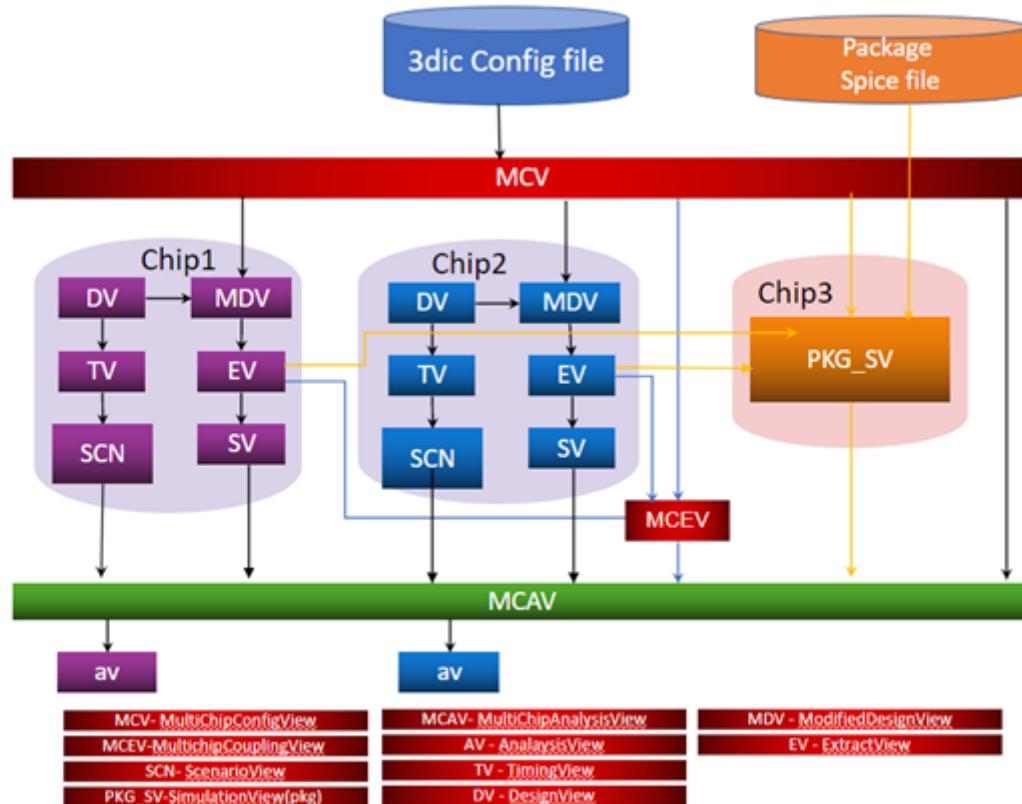
Benefits	Challenges
Can use different process nodes different components	Small heat sink area causes heat accumulation
Small form factor	Design implementation is complex
Short channel length and propagation delay	High complexity
Lower power and higher bandwidth	Difficult to analyze, verify, and signoff
High system performance	Noise coupling
Better yield	Mechanical stress

11.2. Multichip Flow Overview

The following figure shows the high-level flow of RedHawk-SC multichip analysis and the main views to perform these steps.

You must create the base views before performing these steps. Creating base views for multichip analysis is similar to single chip analysis except for minor differences. See [Creating Base Views](#) on page 32.

Once you create base views for each chip including the interposer, you should create views specific to multichip analysis.



There are slight differences in the multichip flow compared to single chip runs. Multichip flow requires three additional views, namely, MultichipConfigView, MultichipCouplingView, and MultichipAnalysisView.

- **MultichipConfigView:** Loads and stores the configuration file created by the setup 3dic utility. The view contains connection and orientation information for the tool to connect multiple chips. The configuration file contains information about how chips are arranged on top of interposer, and how the chip pad locations are connected to the interposer.
- **MultichipCouplingView:** Extracts the coupling parasitic between the neighboring chips. This is not a mandatory view.
- **MultichipAnalysisView:** Performs analysis for the entire multichip system. You cannot use this view as a regular AnalysisView. To obtain results from a regular AnalysisView, query the chip-specific AnalysisView. Functions to query the MultichipAnalysisView are available.
- **Power MultichipElectromigrationView:** Optional. Performs power electromigration analysis for the entire multichip system.

11.3. Design Data Requirements

Required data for RedHawk-SC multichip analysis is slightly different from single-chip analysis. You require the following specific design data.

- [Connection Point Locations of Each Chip](#) on page 332
- [Interchip Interconnect Parasitics of Connection Points](#) on page 332
- [TSV Parasitics](#) on page 333

Connection Point Locations of Each Chip

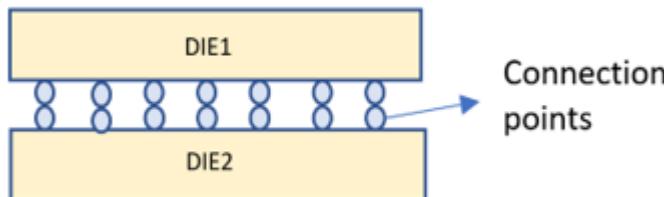
Connection point locations are similar to pad locations in single-chip analysis, and are required for each chip of a multichip system. The format of a connection point location file is same as that of the pad location (PLOC) file.

For an interposer, two sets of connection point locations are associated with a single chip. One set is for the chip-interposer interface, and another for the interposer-package interface. If a package is added to the interposer, input a connection point location file with six columns where the sixth column represents the package port name. The connection point name should be unique. The following example shows a snippet of a connection point location file.

```
#Connection_Point_Name    x      y      metal_layer   net_name
frontBumpAP_6_7           1750  1550  AP            VDD
frontBumpAP_6_14          3150  1550  AP            VDD
frontBumpAP_6_16          3550  1550  AP            VDD
frontBumpAP_7_5           1350  1750  AP            VDD
```

Interchip Interconnect Parasitics of Connection Points

Connection points (such as, copper pillars) are structures that connect different chips as shown in the following figure.



There can be power grid connections as well as signal net connections. You must define the dimensions (length, width, height) and the RLC values associated with these structures.

The following example shows RLC values and dimensions (in microns) of a connection point. The tool uses HEIGHT to calculate capacitive and inductive coupling between two dies. When HEIGHT reduces, the coupling capacitance increases.

```
R 1e-4
L 1e-12
C 1e-15
LENGTH 10
WIDTH 10
HEIGHT 5
```

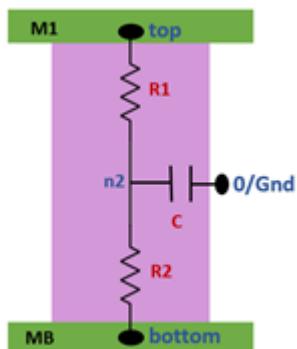
TSV Parasitics

You can input TSV parasitics from two sources:

- Technology file from foundry
- TSV sub-circuit

In general, a technology file contains TSV parasitic information. It is also possible to define parasitics using a SPICE sub-circuit file.

In the sub-circuit file top and bottom are mandatory nodes. The tool uses these nodes to understand which node to connect to which layer.



The following example shows a TSV sub-circuit file.

```
.subckt tsv_rc top      bottom
        r1    top      n2      0.012
        r2    n2      bottom   0.012
        C     n2      0       1.548E-13
.ends
```

To input the TSV sub-circuit file, use the `tsv_subckts` argument of the `create_extract_view` command:

```
#TSV subcircuits definition
tsv_subckts = [
{'layer_name' : '<layer1>', 'subckt_name' : '<subckt_name>', 'subckt_file' :
':<path_to_subckt_file>'},
{'layer_name' : '<layer1>', 'subckt_name' : '<subckt_name>', 'subckt_file' :
':<path_to_subckt_file>'},
.... ]
```

```
create_extract_view(..., tsv_subckts=tsv_subckts, ...)
```

The following example shows how to define TSV sub-circuits for different layers.

```
tsv_subckts = [
    {'layer_name' : 'TDV', 'subckt_name' : 'tdv_rlc', 'subckt_file' :
     './tsv_rlc.subckt'},
    {'layer_name' : 'TSV', 'subckt_name' : 'tsv_rlc', 'subckt_file' :
     './tsv_rlc.subckt'},]
```

11.4. Creating Multichip Configuration File Using Setup Utility

A multichip configuration file contains the following information:

Table 20: Contents of a Multichip Configuration File

Content	Description
Information about individual chips of the multichip system	<ul style="list-style-type: none"> • Die size • DEF names and instantiation names of chips • Interface definitions • Position, layer, and net names
Details of connection points	Dimensions and parasitics of the connection points
Multichip connection information	<ul style="list-style-type: none"> • Connections between individual chips with locations • Individual connection point pairs • Electrical models for connection points
System interface	Defines the package interface

To create the multichip configuration file, follow these steps:

1. [Invoke Multi-Die Setup](#) on page 336
2. [Setup Path and Work Directory](#) on page 336
3. [Create New Project](#) on page 337
4. [Add Dies](#) on page 338
5. [Input Design Data](#) on page 339
6. [Add Off-Chip Model Prototype](#) on page 345
7. [Generate the Config File](#) on page 345

The following example shows a multichip config file generated by using the RedHawk-SC **Multi-Die Setup** interface:

```
#GALAXY CONFIG FILE EXAMPLE
DIE Galaxy 1 {           → Original DIE name from DEF
    INSTANCES CHIP1 CHIP2 → Instantiations
    NET {
        VDD POWER IN
        VSS GROUND IN
    }
    INTERFACE IF_DIE_INT { → All the interface locations
        VDD_DIE_1 28 1220 metal12 VDD
    }
}
```

```

        ...
        ...
SIZE 0 0 1226.83 1226.4
    SHRINK 1
}
DIE GalaxyInterposer 2 {
    INSTANCES Interposer
    NET {
        VDD POWER INOUT
        VSS GROUND INOUT
    }

    INTERFACE IF_INT_CHIP1 { → Interfaces to CHIP1
    ...
}

INTERFACE IF_INT_CHIP2 { → Interfaces to CHIP2
    VDD_INT2_1 2624 1220 UBM VDD
    ...
}

INTERFACE IF_INT_C4 { → Interfaces to C4 (package)
    VDD_PKG_1 146 1216 UBMB VDD VDD_VRM
}

CONNECTION Interposer_CHIP1 {
    REF_DIE Interposer IF_INT_CHIP1 → DIE taken for reference
    CONNECT_DIE CHIP1 IF_DIE_INT
    PLACE 21.4292 -0.616852 → Relative placement information
    ORIENT N □ Relative Orientation
    CONNECTION_POINTS { → Connection between interfaces
        VDD_INT_1 VDD_DIE_1 PGCP → Connection parasitic type
        ...
    }
}

CONNECTION Interposer_CHIP2 { → Connection information to DIE2
    REF_DIE Interposer IF_INT_CHIP2
    CONNECT_DIE CHIP2 IF_DIE_INT
    PLACE 1446 -6.10352e-05
    ORIENT FN
    CONNECTION_POINTS {
        VDD_INT2_1 VDD_DIE_1 PGCP
        ...
    }
}

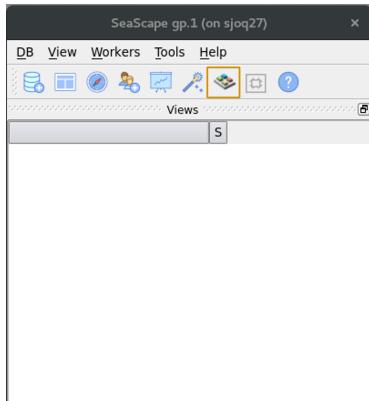
CP PGCP { → Connection point parasitic and dimensions
    R 0.001
    L 2e-11
    C 5e-12
    LENGTH 0.8
    WIDTH 0.8
    HEIGHT 0.001
}

SYSTEM {
    PACKAGE_CONN { → Interface connecting to package
        PLOC_INTERFACE Interposer IF_INT_C4
    }
}

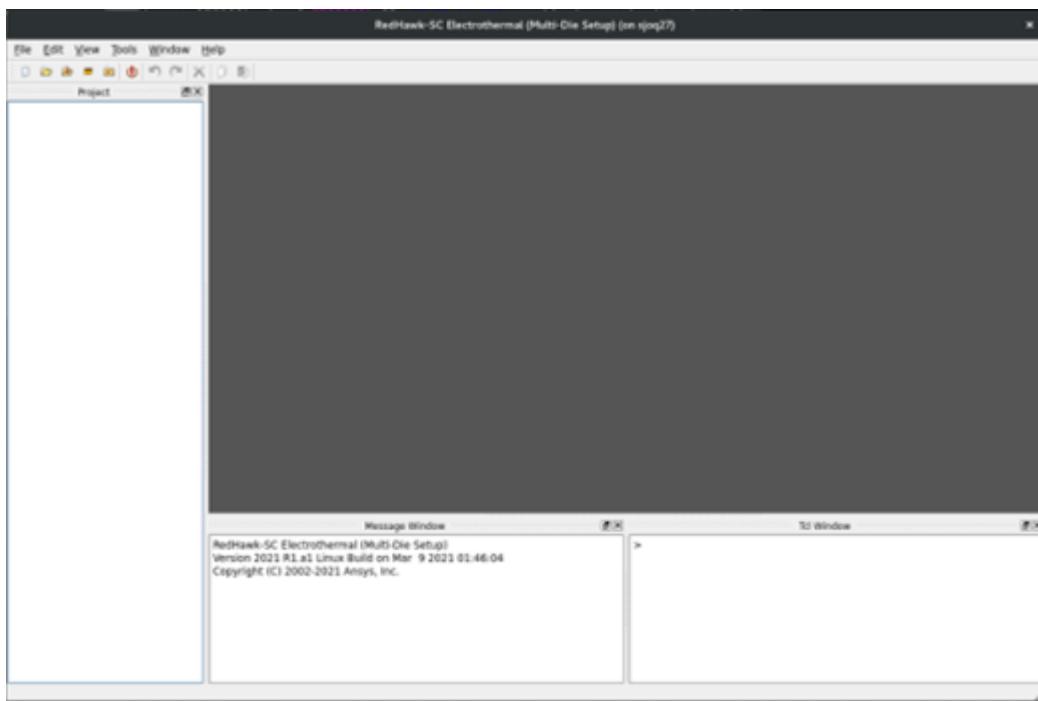
```

11.4.1. Invoke Multi-Die Setup

1. Launch the RedHawk-SC console (`redhawk_sc --console`). See [Launching the RedHawk-SC Console](#) on page 463.
2. Click the **Multi-Die Setup** icon.

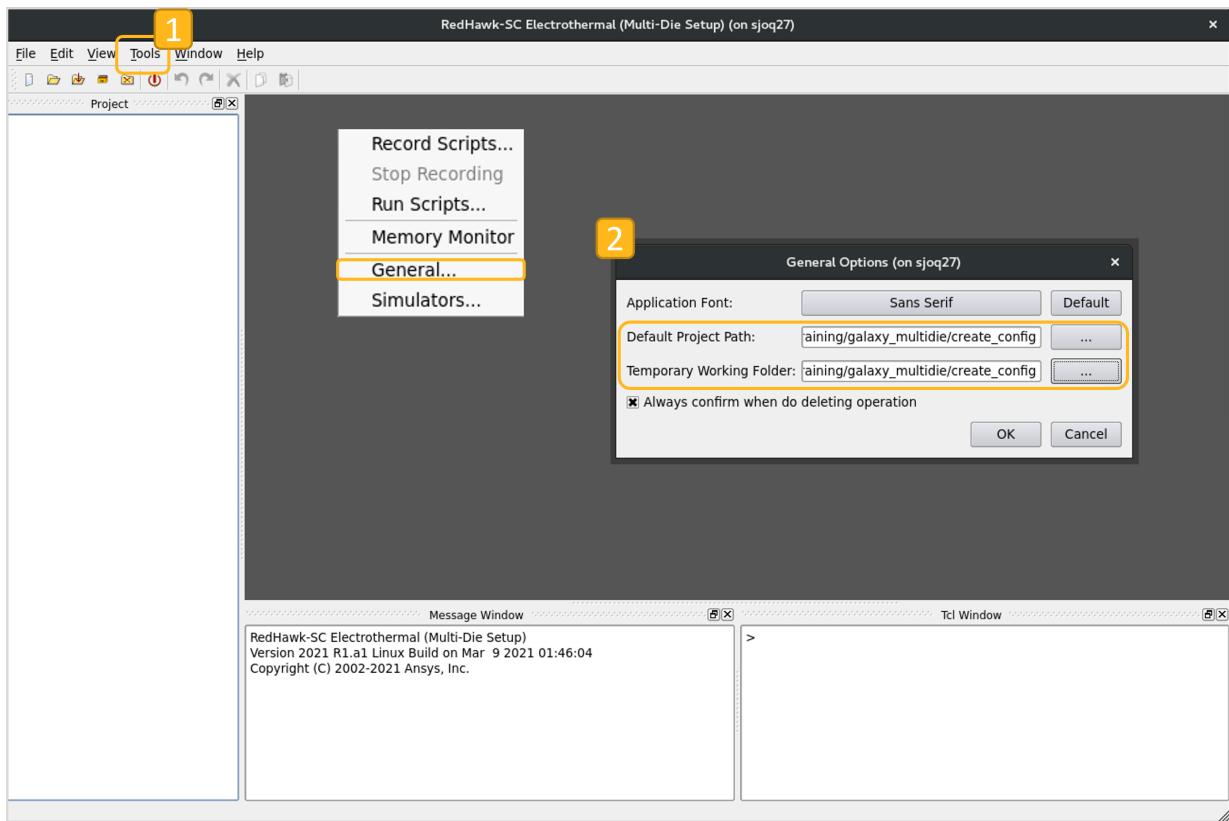


This opens the **RedHawk-SC Electrothermal (Multi-Die Setup)** window.



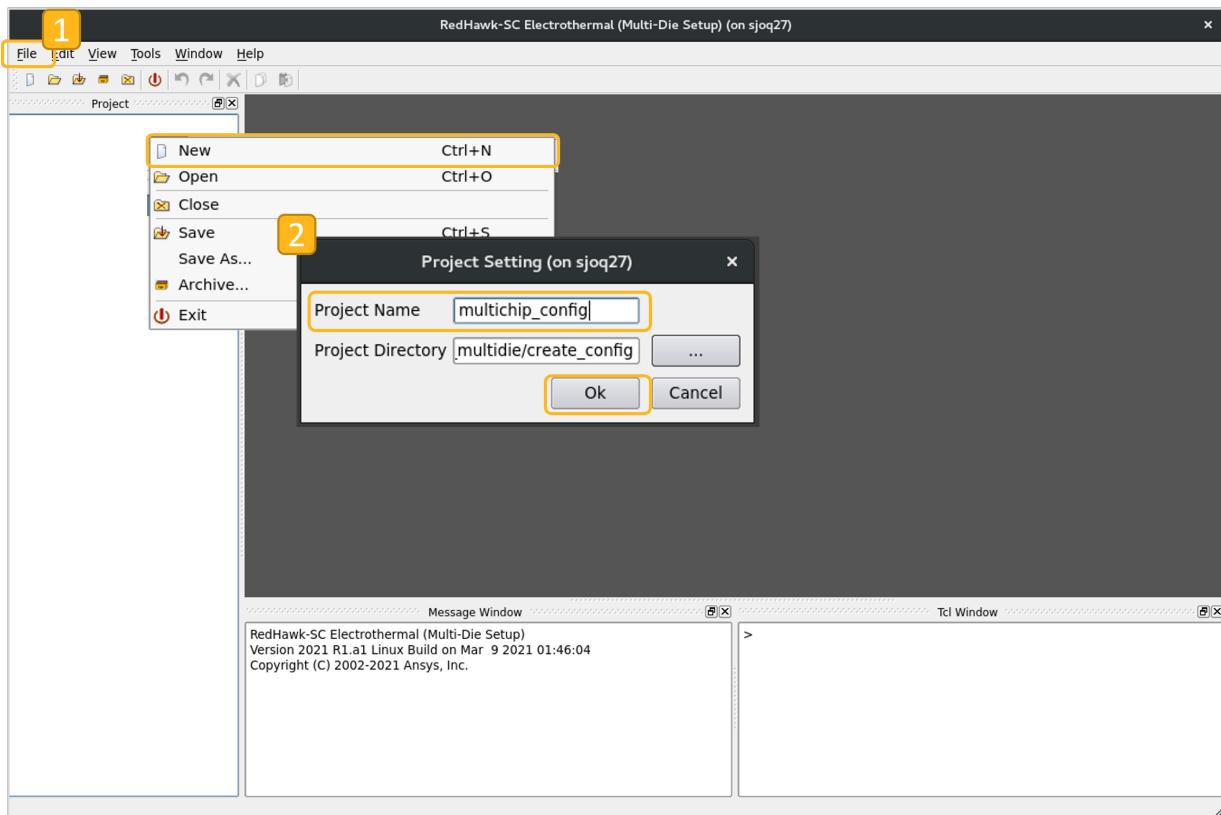
11.4.2. Setup Path and Work Directory

1. Select **Tools>General....**
2. Set the **Default Project Path** and **Temporary Working Folder**.



11.4.3. Create New Project

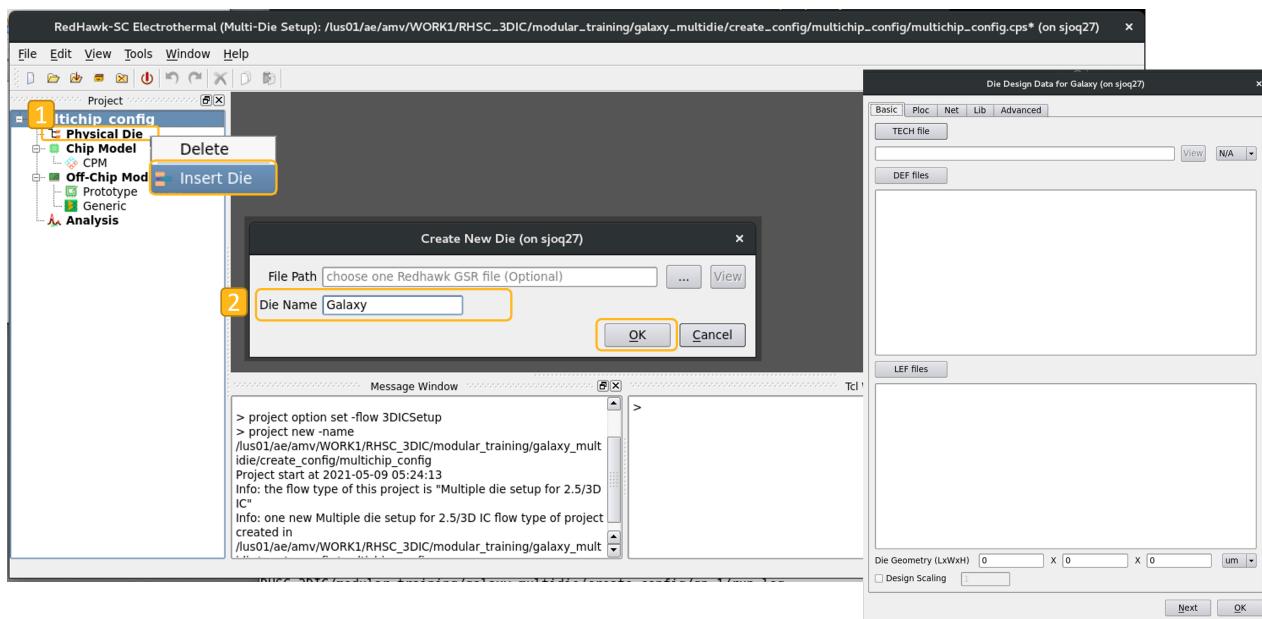
1. Select **File>New** from the main menu or press the **Ctrl+N** keys. This opens the **Project Setting** dialog box.
2. Input the **Project Name** and the **Project Directory** path.



This opens the **Project** window on the left.

11.4.4. Add Dies

1. Right-click **Physical Die**>**Insert Die**. The **Create New Die** dialog box appears.
2. Input the **Die Name** as per the top DEF name. This instantiates the die under **Physical Die** and opens the **Die Design Data** window (shown on the right) that you use to input design data.

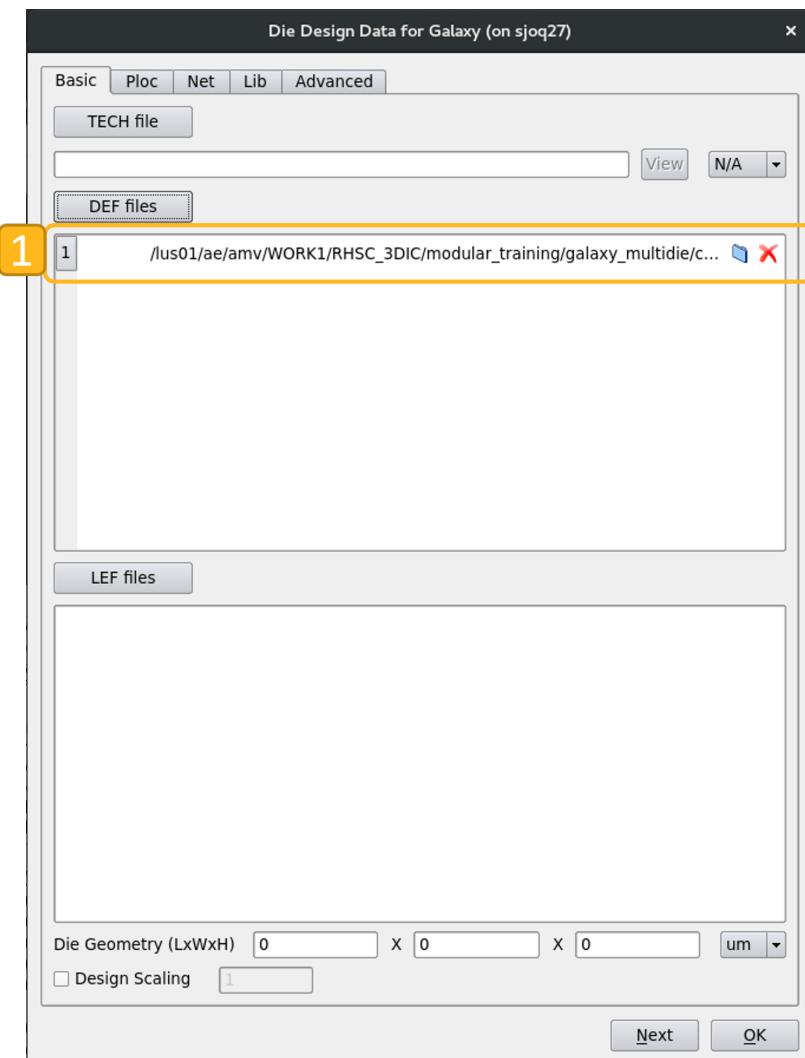


11.4.5. Input Design Data

1. Add DEF file.

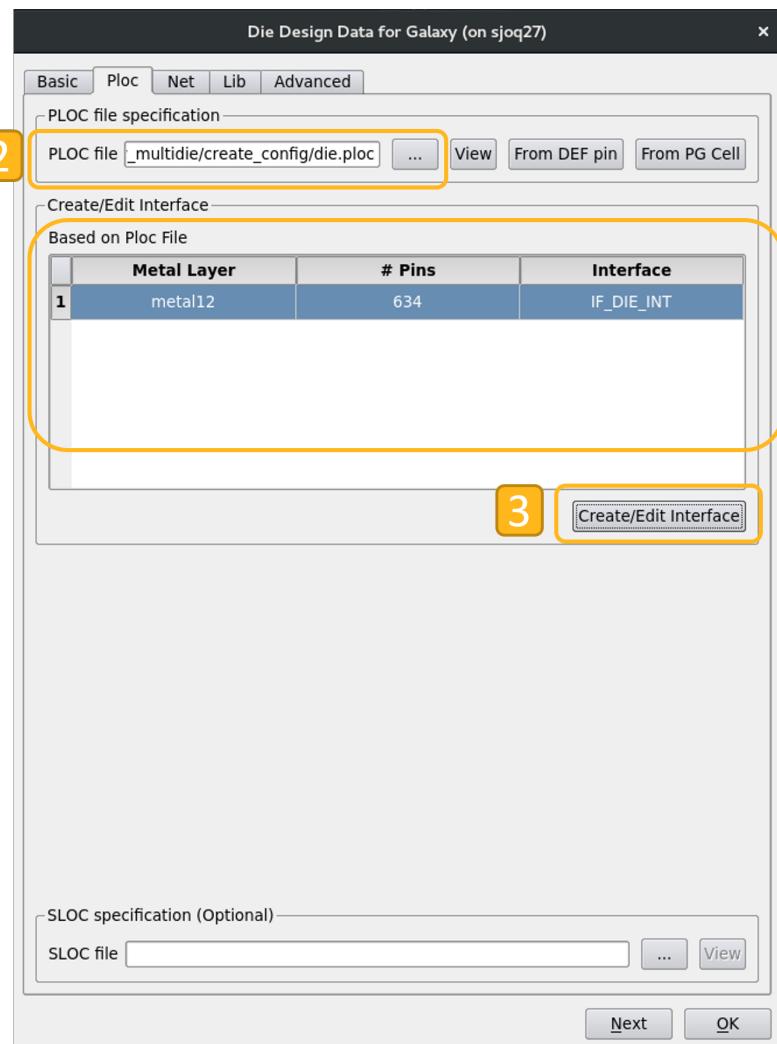
On the **Basic** tab, click **DEF files** to add the top DEF file. Then click **Next** to move to the **Ploc** tab.

Note: If the DEF file is large, that is, takes a long time to load, you can directly input the **Die Geometry (LxWxH)** coordinates to save time.



2. Add PLOC file.

On the **Ploc** tab, click the button next to the **PLOC file** field to add the top PLOC file that includes pad locations of all interfaces. On adding the PLOC file, you can view each **Metal Layer** where the pad locations are instantiated, **# Pins** (number of pins), and the **Interface** name. By default, the tool automatically generates the interface name.

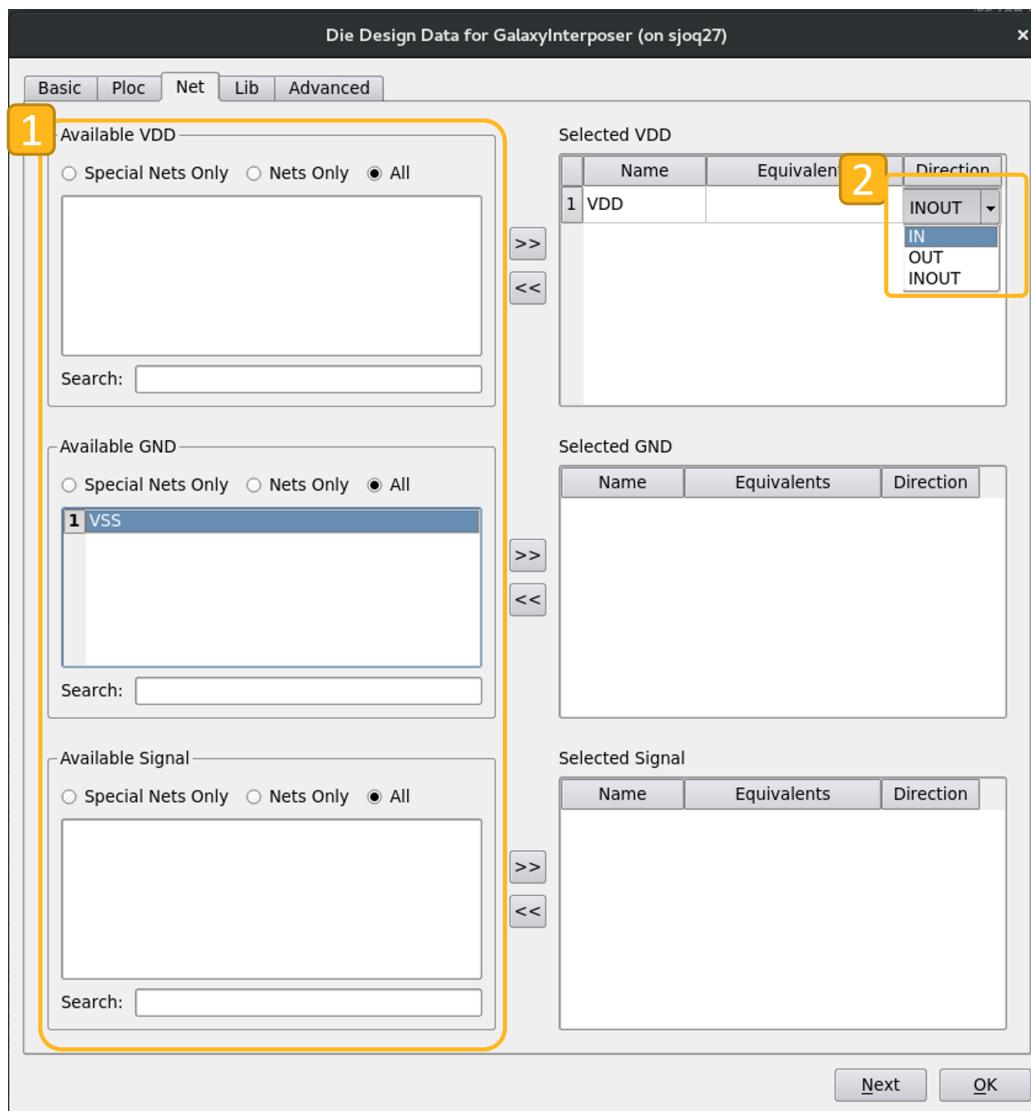


- To change the default interface name, click the **Create/Edit Interface** button to open the **Create Interface** dialog box and select the specific interface. The **Input Interface Name** dialog box opens. Type the new name of the interface.

3. Add nets.

On the **Net** tab,

- Select the nets to analyze.
- Select the net direction. If you have loaded a DEF file, this is automatically updated according to the DEF file.



Group Pad Locations

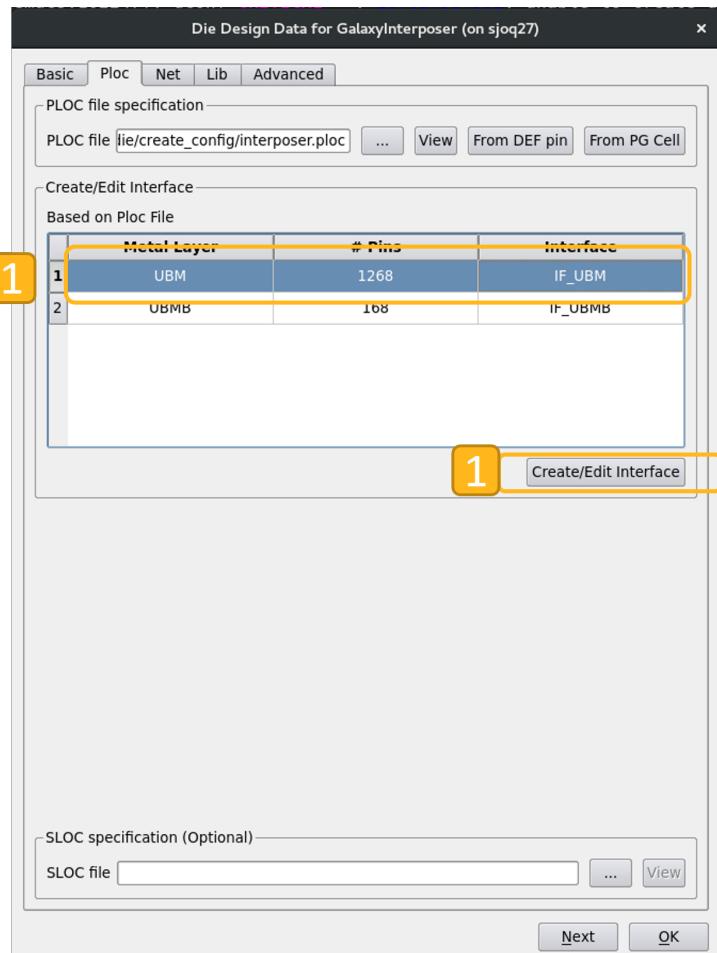
The top-level PLOC file is a single file that contains pad location information of all the interfaces of the die. If the die has only one interface, you need not group the pad locations.

An interposer die can have multiple interfaces. For example, two interfaces to connect to DIE1 and DIE2 and another interface to connect to the package. In such a case, you need to group the pad locations to create a single interface per die on each layer (DIE1 and DIE2 for layer IF_UBM).

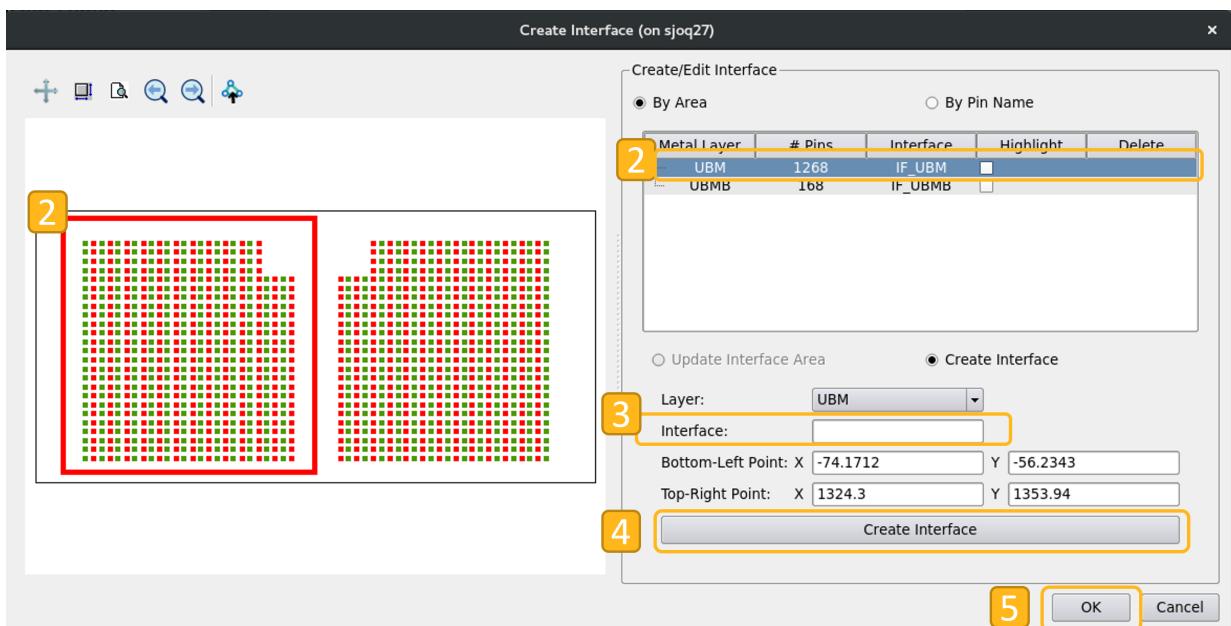
You can group pad locations by area or by pin names.

To group pad locations by area,

1. Select the interface and then click **Create/Edit Interface**. This opens the **Create Interface** dialog box.



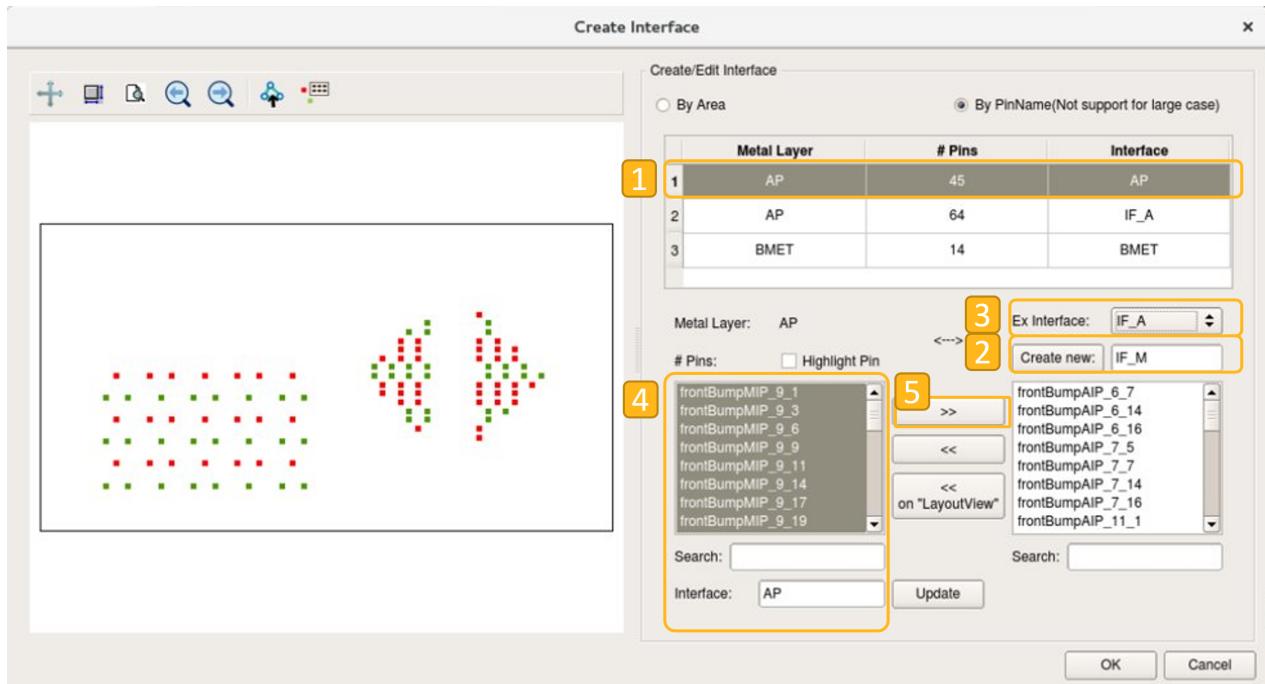
2. Select the interface. In the following example, there are two sets of interface points for connection to two different dies. Select one set of interface points by dragging the left mouse button to create a bounding box. This creates a new interface.



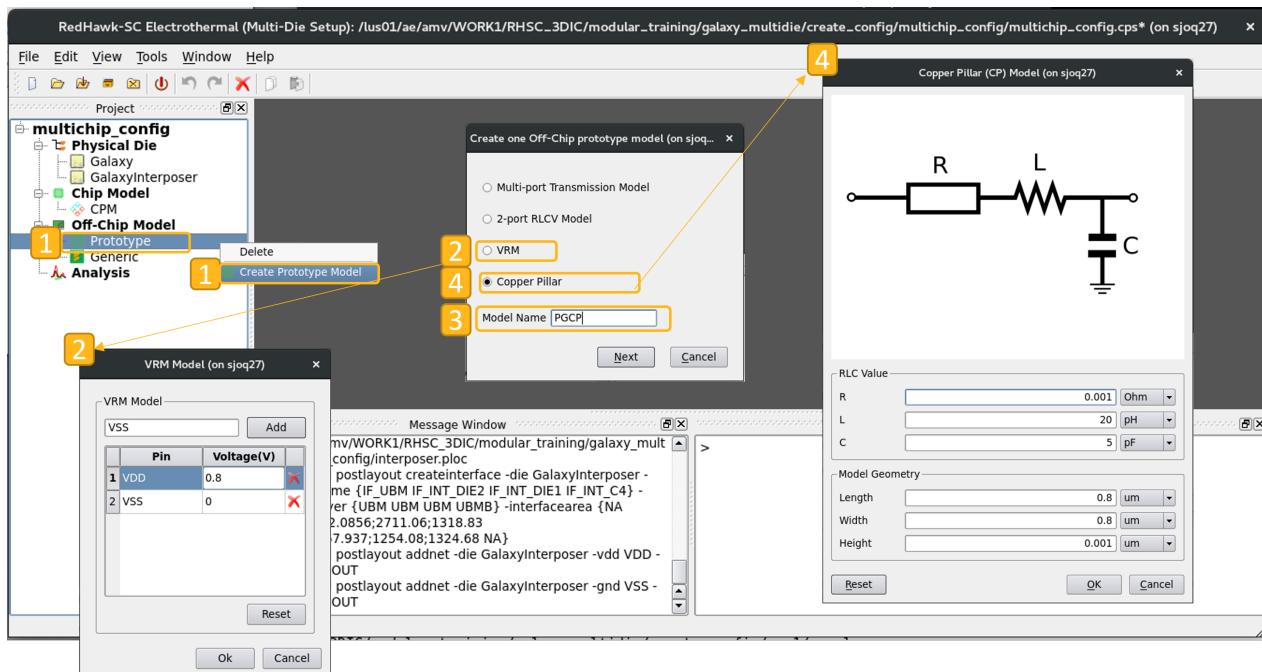
3. Enter the name of the new interface, for example, IF_INT_DIE1 (Interposer to DIE1 interface).
4. To create a separate interface for each die connection, repeat the steps **2** and **3** and then click **Create Interface**.
5. Click **OK** and move to the **Net** tab.

To group pad locations by pin names,

1. Select the interface to be modified.
2. Enter the new interface name in the **Create new** field.
3. Select an interface which will be assign pins.
4. Search pins based on the name, and select pins.
5. Add to the selected interface.



11.4.6. Add Off-Chip Model Prototype



To create off-chip model prototype,

1. Expand **Off-Chip Model** under the **Project** window and right-click **Prototype**. Now, click **Create Prototype Model** to open the **Create one Off-Chip prototype model** dialog box.
2. To create the VRM model for model-based flow, select **VRM**. This opens the **VRM Model** dialog box where you can specify the VDD and VSS values for a reduced model. You need to perform this step only if you add a CPM.
3. To change the model name, enter the model name in the **Model Name** field.
4. To add a copper pillar model, select **Copper Pillar**. The **Copper Pillar (CP) Model** dialog box opens.
 - a. If required, edit the **RLC Value** and **Model Geometry** parameters.

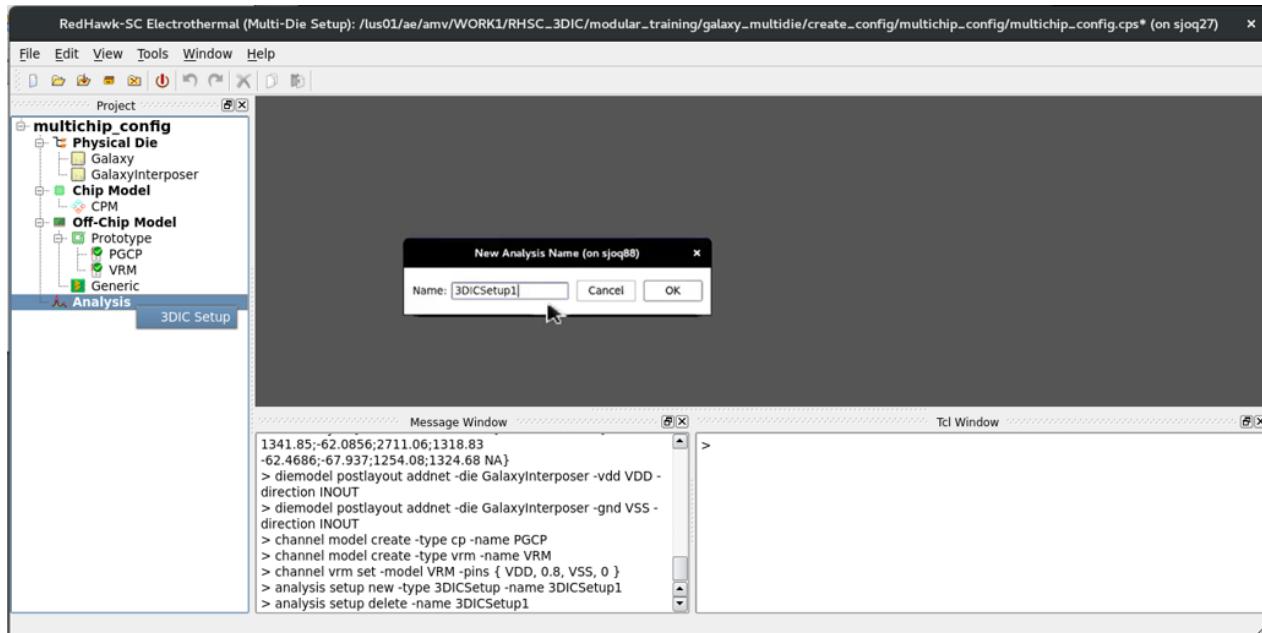
Note: The **RLC Value** and **Model Geometry** parameters are per copper pillar between individual interface points.

11.4.7. Generate the Config File

This topic includes the following sections:

- [Specify Analysis Name](#) on page 346
- [Create Config File for Full Detailed Analysis](#) on page 346
- [Create Config File for Designs With Reduced Models \(CPM\)](#) on page 349

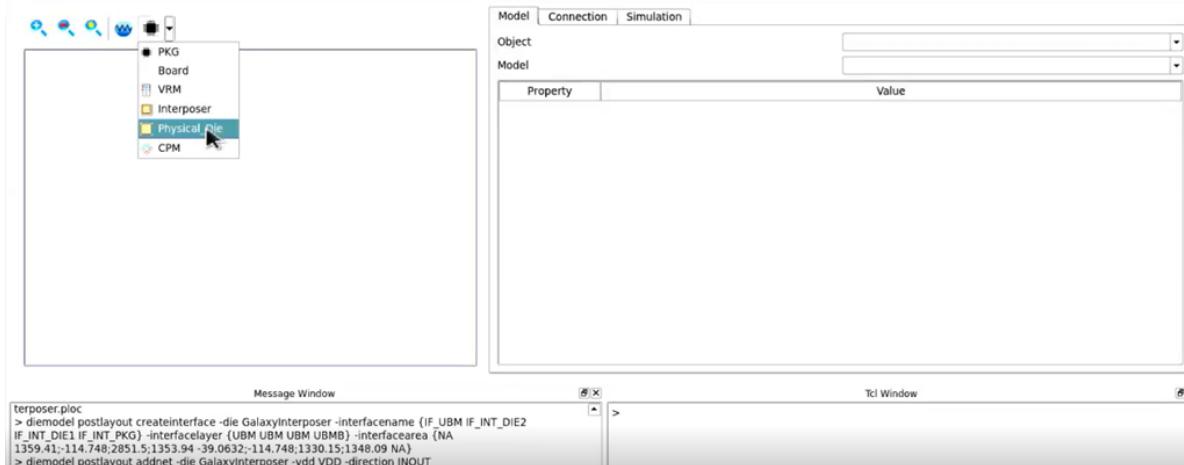
Specify Analysis Name



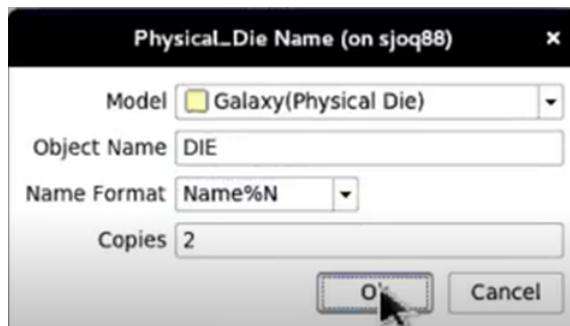
1. Right-click **Analysis**. This displays the **3DIC Setup** icon.
2. Click the **3DIC Setup** icon to open the **New Analysis Name** dialog box.
3. Enter the name, for example, **3DICSetup1**. The analysis name now appears under **Analysis** in the Project window.

Create Config File for Full Detailed Analysis

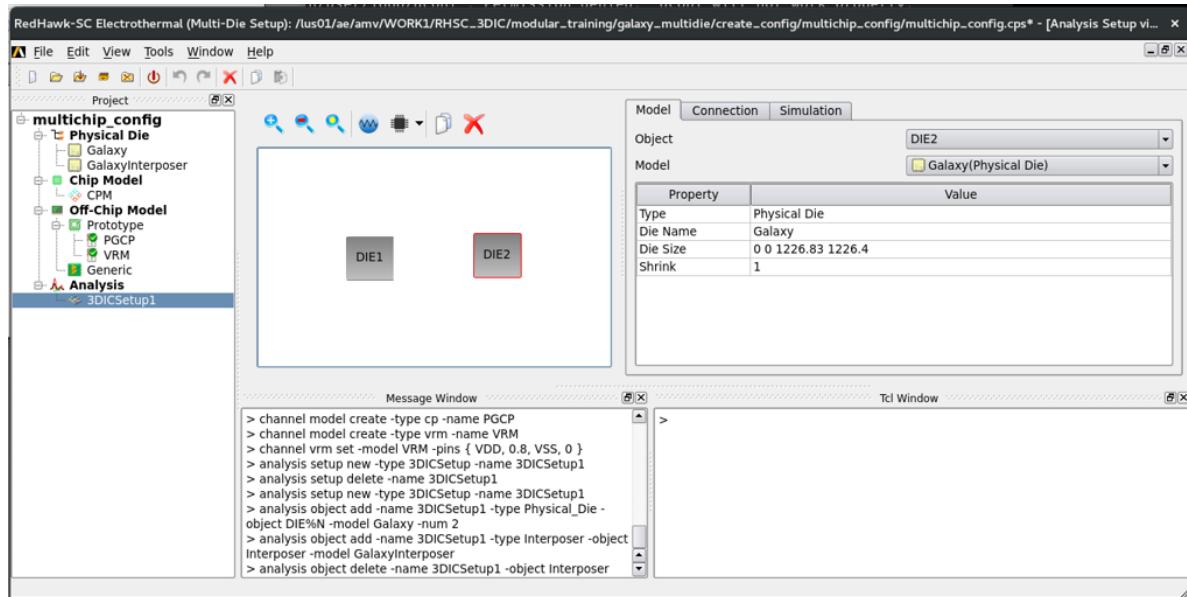
1. Add objects and select model.
 - a. Double-click the analysis name, for example, **3DICSetup1**. This opens the following window.



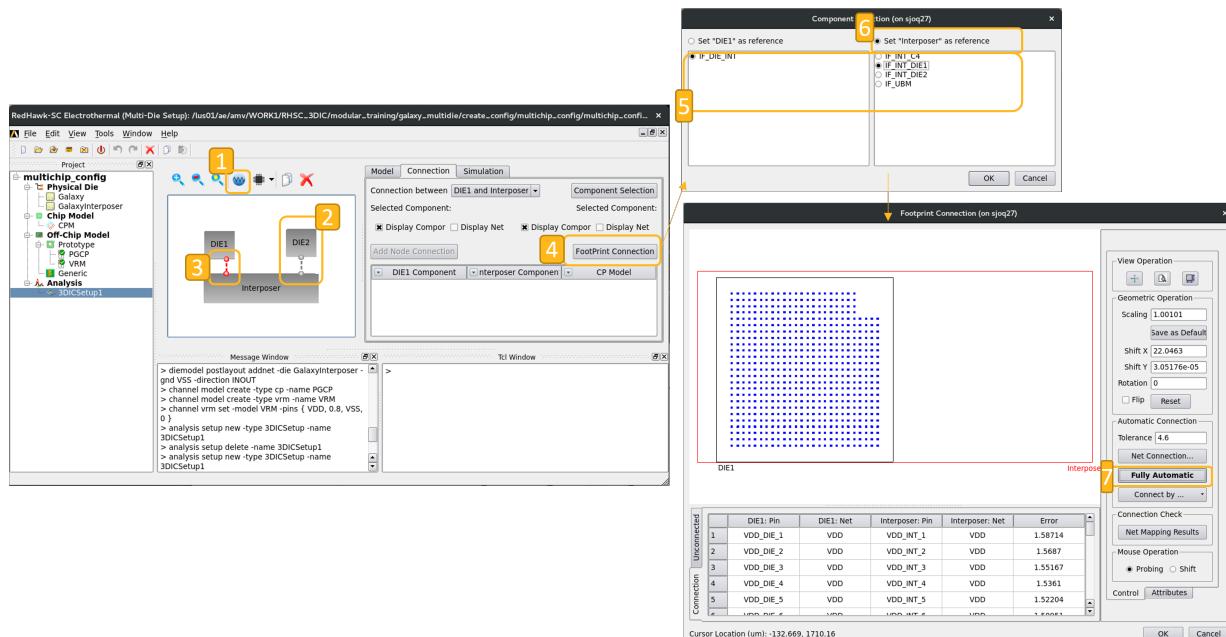
- b. Click the drop-down list as shown and add the required objects. To add all the dies and interposer, select **Physical_Die**. This opens the **Physical_Die Name** dialog box as shown:



- c. Select the **Model** from the drop-down list, change the **Object Name** if required, select the **Name Format**, and specify the number of instantiations using **Copies**. The objects are now displayed.

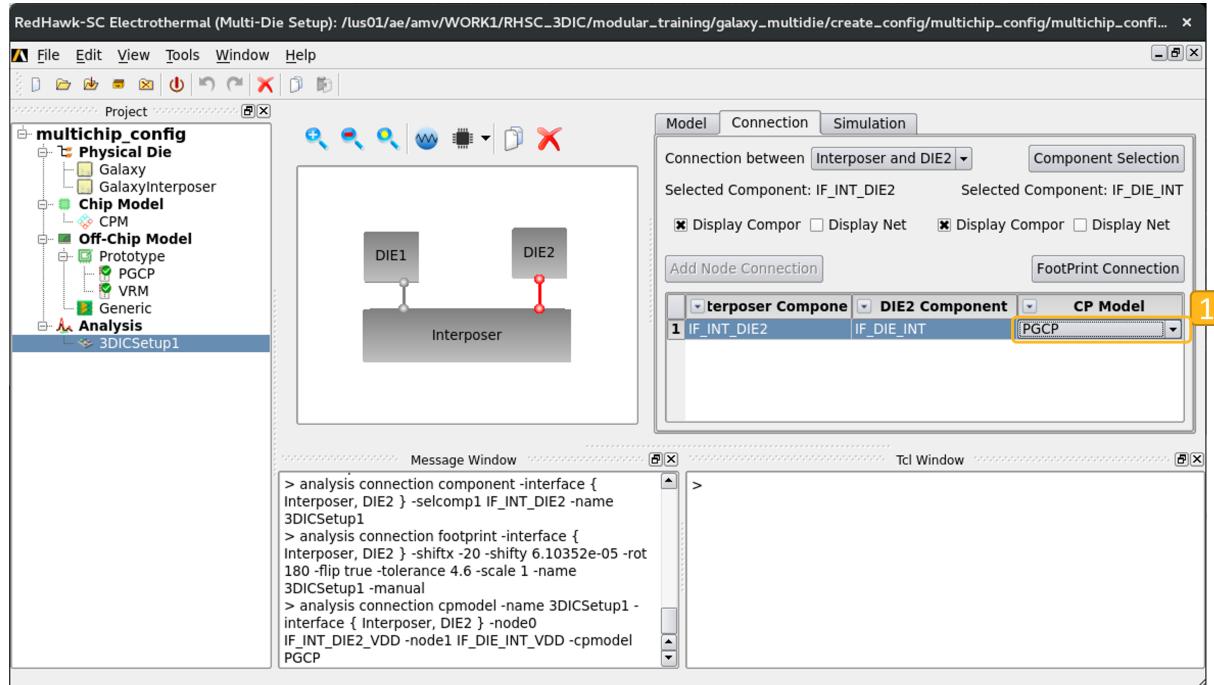


2. Connect the objects.



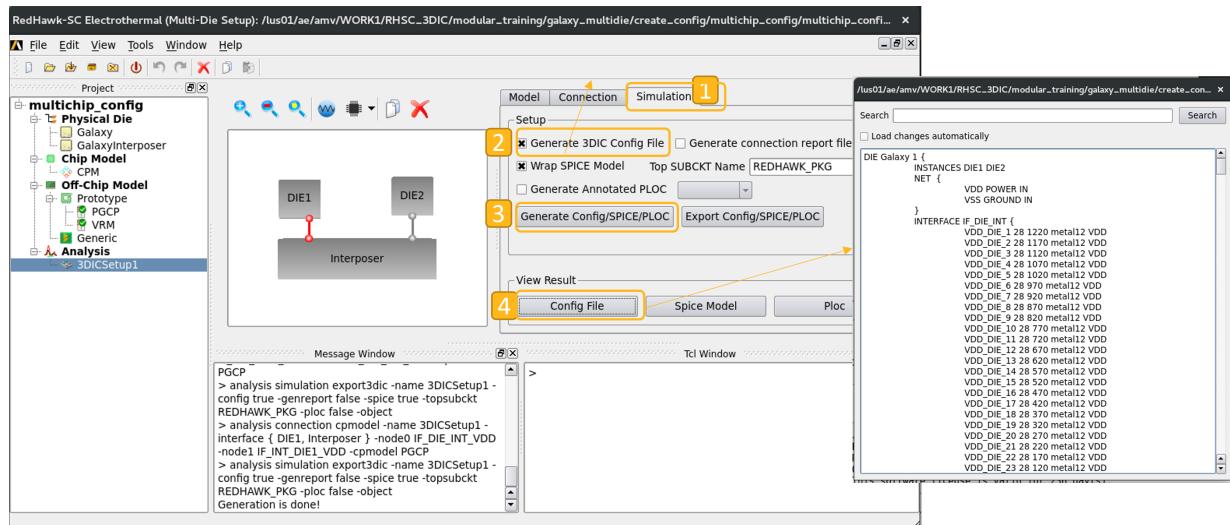
- a. Click the **Add Connection** icon.
- b. Select the objects to connect. This connects the objects by inserting a bus line. After all the models are connected, press the **Esc** key or click the **Add Connection** icon again.
- c. To move to the **Connection** tab, double-click the bus line.
- d. Click **FootPrint Connection**.
- e. Select the two interfaces to connect.
- f. Ensure the object that is below another object is taken as reference, and click **OK**. This opens the **FootPrint Connection** dialog box.
- g. To connect all dies, click **Fully Automatic**. When all the connections are made, the dotted line changes into a solid line.

3. Assign the CP model.



- a. After you have connected the objects, select **CP Model** from the drop-down list.

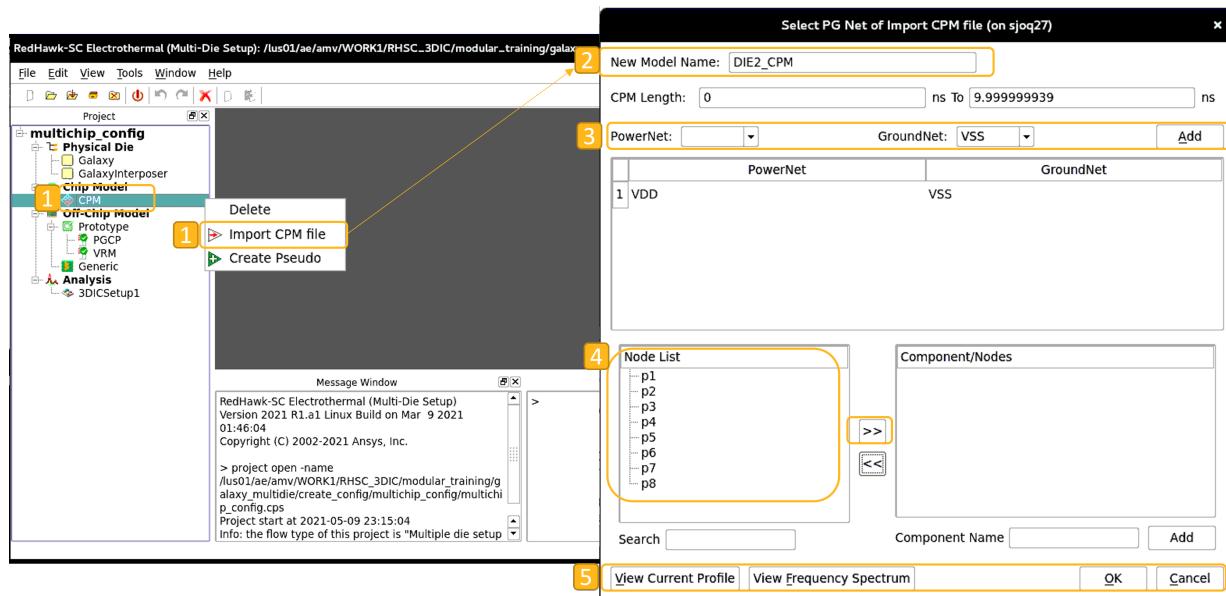
4. Generate the 3DIC config file.



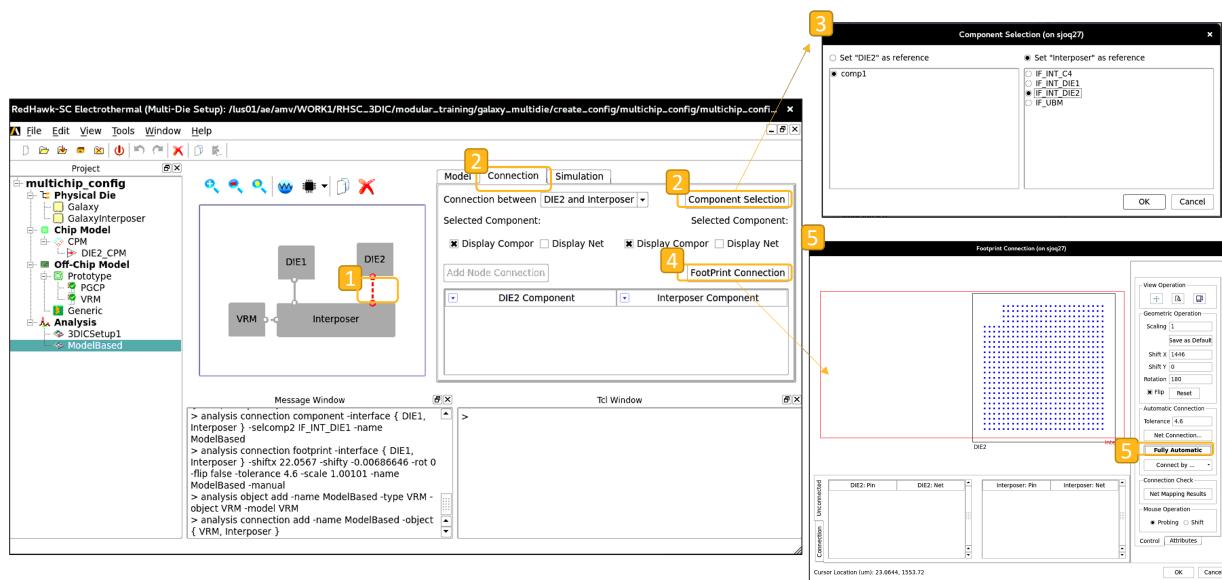
- Click the **Simulation** tab.
- Select **Generate 3DIC Config File**.
- Click **Generate Config/SPICE/PLOC**. This generates the multichip configuration file in the <project_name>/analysis/<setup_name>/output/ directory. To output the file to a different directory, click **Export Config/SPICE/PLOC**.
- Click **Config File**. This opens the multichip configuration file.

Create Config File for Designs With Reduced Models (CPM)

- Import CPM models.

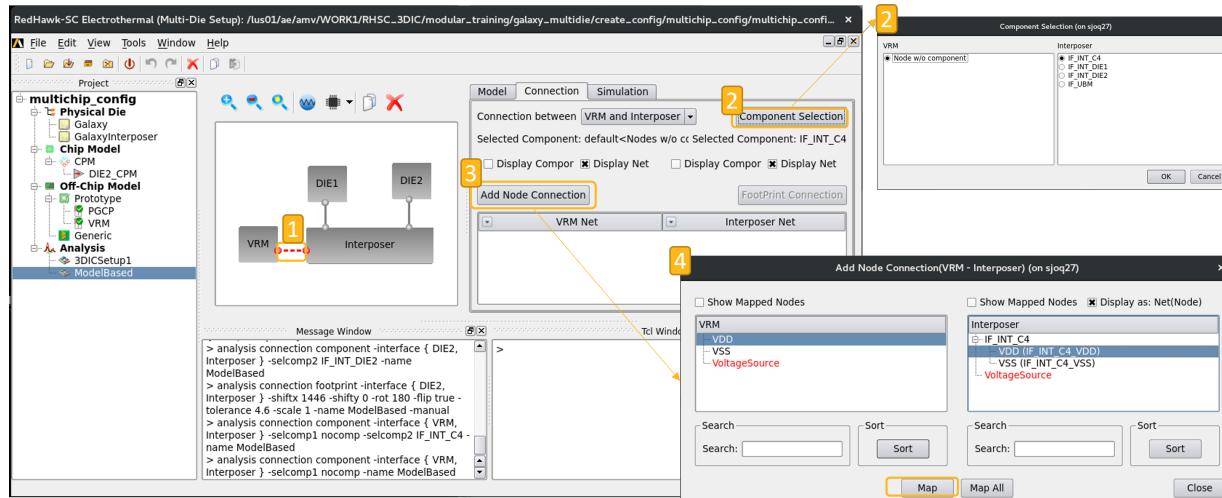


- Right-click **CPM** and select **Import CPM file**.
 - Enter a model name for the imported CPM (DIE2 CPM imported).
 - Select **PowerNet** and **GroundNet** values from the corresponding drop-down lists, and click **Add**.
 - Select all the nodes under **Node List** and click **>>** to move these under **Component/Nodes**.
 - You can **View Current Profile** and **View Frequency Spectrum** by clicking the corresponding buttons.
- Connect the CPM model.



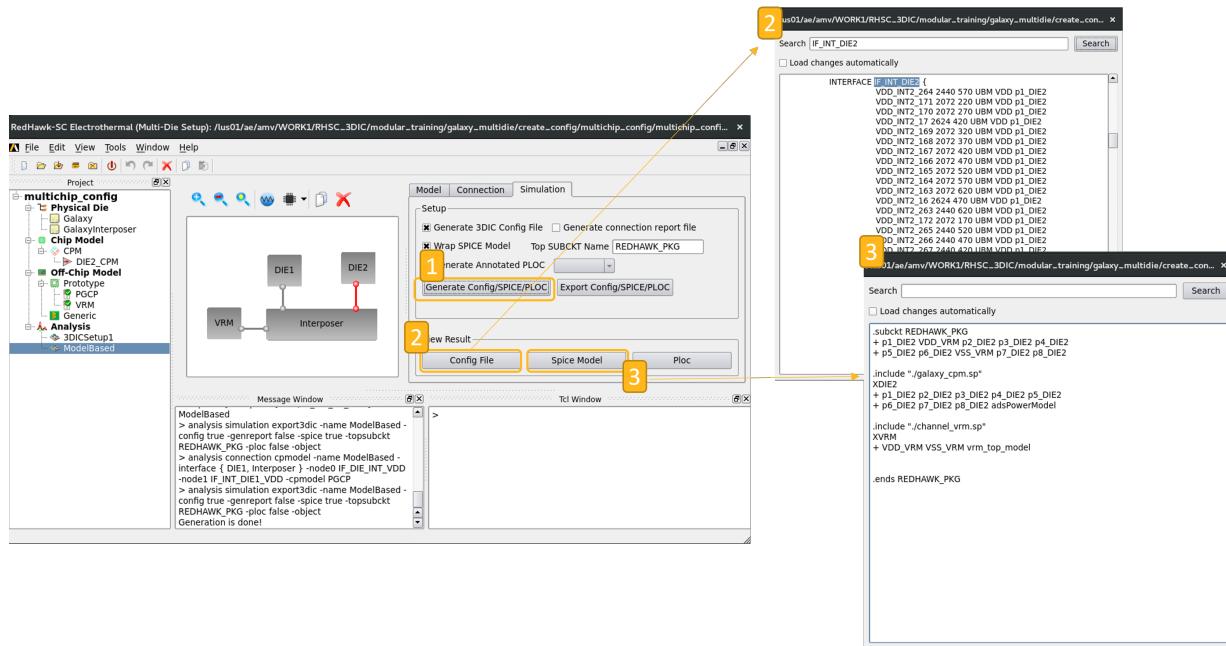
- Double-click the connection between CPM and **Interposer**.
- In the **Connection** tab, click **Components Selection**. The **Component Selection** dialog box opens.
- Select the interfaces to be connected to the CPM.
- In the **Connection** tab, click **Foot Print Connection**. The **Footprint Connection** dialog box opens.
- For automatic connection, select **Fully Automatic**.

3. Connect the VRM model.



- Double-click the **VRM–Interposer** connection.
- In the **Connection** tab, click **Components Selection**. The **Component Selection** dialog box opens.
- Select the components and interfaces.
- Click **Add Node Connection**. The **Add Node Connection (VRM–Interposer)** dialog box opens.
- Map** VRM nodes with Interposer VDD or VSS bump groups.

4. Create SPICE wrapper and six-column reference PLOC (config).



- To create a wrapper SPICE file, click **Generate Config/SPICE/PLOC**.
- Config file can be used to take interposer ploc to CPM connection reference (<project_name>/analysis/<setup_name>/output/).
- Click **Config file** and search for interposer to DIE interface to see the same.
- You can view the SPICE wrapper clicking **Spice Model**.

11.5. Creating Command File for Multichip Analysis

The following shows the steps to create a command file for RedHawk-SC multichip analysis.

modified DesignView for each chip integrates the INTERFACE details from MultichipConfigView. You must input the `die_name` and `die_sub_id` (both from multichip config file) to this view.

```
# Create MultichipConfigView
multichip_config = db.create_multichip_config_view(config_file,
tag='multichip_config',
options = options)

# Modified DesignView for each chip integrates the INTERFACE details from
# MultichipConfigView. You must input the die_name and die_sub_id (both from
# multichip config file) to this view.
dv_interposer = db.create_modified_design_view(dv0_interposer,
config_view=multichip_config, die_name='GalaxyInterposer', die_sub_id=2,
options=options, tag='dv_interposer')

# Create base views
# 1. Create all other base views as a single chip flow.
# 2. Pass the chip-specific ScenarioView and Simulation View to the
MultichipAnalysisView.

# Create MultichipCouplingView
coupling_views=[{'chip_name':'CHIP1', 'ev':ev_top},
{'chip_name':'CHIP2', 'ev':ev_top},
{'chip_name':'Interposer', 'ev':ev_interposer},
```

```

    {'config_view':multichip_config}
]
multichip_coupling_view=db.create_multichip_coupling_view(coupling_views,
tag='multichip_coupling_view')

# Create MultichipAnalysisView
multichip_views =
[{'chip_name':'DIE1', 'sv':sv_top, 'scn':scn_no_prop_top, 'detailed':True},
 {'chip_name':'DIE2', 'sv':sv_top, 'scn':scn_no_prop_top, 'detailed':True},
 {'chip_name':'Interposer', 'sv':sv_interposer, 'scn':scn_noProp_interposer,
 'detailed':True},
 {'config_view': multichip_config},
 {'coupling_view':multichip_coupling_view},
 ]
av_3d=db.create_multichip_analysis_view(multichip_views=multichip_views,
settings={'duration':10e-9, 'step_size':30e-12,'keep_stats_level':'Full'},
tag='av_3d', options=options)

# Create Power MultichipElectromigrationView
emv_3d_dc = db.create_multichip_electromigration_view(av_3d,
tag='emv_3d_dc', options=options, mode='DC')

```

Before performing multichip analysis, create the required base views for all the chips of the system. The following sections show command script examples to create base views and packages in multichip systems.

- [Creating Die Base Views](#) on page 352
- [Creating Interposer Base Views](#) on page 353
- [Creating Package for a Multichip System](#) on page 353
- [Creating Package Per Bump in Multichip System](#) on page 354

Creating Die Base Views

The following snippet shows the command script to create base views of an individual chip for multichip analysis.

```

include('input_files_die.py')

lv_top = db.create_liberty_view(liberty_file_names=lib_files_top,
apl_switch_files= switch_files_top,options=options,tag = "lv_top")

nv_top = db.create_tech_view(tech_file_name=tech_file_top,
options=options,tag = "nv_top")

dv_top =
db.create_modified_design_view(design_view=dv0_top,config_view=mc_config,
die_name='Galaxy',die_sub_id=1,options=options,tag = "dv_top")

ev_top = db.create_extract_view(design_view = dv_top,tech_view = nv_top,
settings={'calculate_spr': True, 'extract_temperature' : 25.0},
options=options,tag = "ev_top")
...

```

Creating Interposer Base Views

The following snippet shows the command script to create base views of an interposer multichip analysis.

```
include('input_files_interposer.py')

nv_interposer =
db.create_tech_view(tech_file_name=tech_file_interposer,options=options,
tag='nv_interposer',settings={'via_variation':'typical'})

dv_interposer =
db.create_modified_design_view(dv0_interposer,config_view=mc_config,
die_name='GalaxyInterposer',die_sub_id=2,options=options, tag= 'dv_interposer')

extract_settings=
{
    'extract_temperature':25,
    'calculate_spr':True,
    'layers_for_inductance': True
}
ev_interposer = db.create_extract_view(dv_interposer,tech_view=nv_interposer,
options=options,tag='ev_interposer',settings=extract_settings)
...

```

Creating Package for a Multichip System

The tool supports simulating a package connected to multiple chips. You can connect multiple individual chips or chip power models (CPMs) to a package by using the `create_package_simulation_view` command with the `create_multichip_analysis_view` command.

The `create_package_simulation_view` command returns a `SimulationView` that exclusively stores only package-related circuit data. The `create_package_simulation_view` command takes the bump information from the `MultiChipConfigView` and not from the upstream views in the flow. The bumps can be either inter-die bumps or package connecting bumps depending on if the package is for a single chip or for multiple chips.

Because the bump information (that is, if the bump is for a single chip or for multiple chips) is not taken from upstream views, you can now reuse the same views for both single and multichip analysis. For example, you can reuse single-chip base views, such as `DesignView`, `ExtractView`, `ScenarioView`, and `SimulationView` in multichip analysis and vice-versa.

The following example shows the steps to create the `MultichipAnalysisView` to simulate the package of multiple chips.

Use the `multichip_views` argument of the `create_package_simulation_view` command to input multichip views for multichip analysis. To create the `MultichipAnalysisView`, use the `create_multichip_analysis_view` command:

```
multichip_views = [
    {'chip_name': 'chip1' , 'ev': ev1},
    {'chip_name': 'chip2' , 'ev': ev2},
    ...
    {'config_view': cfg_view}
]

pkgs=package.parse_ploc_spice_func(package_spice_file,
multichip_views=multichip_views, pad_file_name=pad_file_name)

pkgs_sv=db.create_package_simulation_view(multichip_views, package=pkgs,
```

```

probes=[list of probe nodes], tag='pkg_sv', options=options)

multichip_views = [
    {'chip_name': 'chip1' , 'scn':scn1, 'sv': sv1},
    {'chip_name': 'chip2' , 'scn':scn2, 'sv': sv2},
    ...
    {'package_simulation_views': [pkg_sv] },
    {'config_view': config_view},
    {'coupling_view': coupling_view},
]

mcav = db.create_multichip_analysis_view(multichip_views, options=options,
tag=tag, ...)

```

Define the chips and interface connected to the package or CPM under the `SYSTEM PACKAGE_CONN` section in the multichip config file.

For the example, when CPM is present in the setup, it is wrapped in the package netlist. Multichip config file must have the following `PACKAGE_CONN` information:

```

SYSTEM {
  PACKAGE_CONN {
    PLOC_INTERFACE Interposer-1      package-Interposer-1-Interface
    PLOC_INTERFACE Interposer-1      CPM-1-Interposer-1-Interface
    PLOC_INTERFACE Interposer-2      package-Interposer-2-Interface
  }
}

```

Creating Package Per Bump in Multichip System

- To create a package per bump, use the `package.create_per_bump_spice_func` function as shown in the following example:

```

package = package.create_per_bump_spice_func
(multichip_views=multichip_views_for_pkg, r=1.1, l=1e-10, c=1e-12)

```

`r`, `l`, and `c` are the bump resistance, bump inductance, and bump capacitance values in SI units.

- To reuse views, use the syntax with the `create_package_simulation_view` command:

```

pkg_sv_multichip =
db.create_package_simulation_view(multichip_views_for_pkg,
package=package, options=options, tag='pkg_sv_multichip')

```

11.6. Multichip Analysis Flows

You can use the following flows to perform multichip analysis:

- [Full Detailed Flow](#) on page 355
- [On-the-Fly Model Based Flow](#) on page 356(Recommended)
- [Model Based Approach](#) on page 358
- [Interposer PDN Quality Checks \(Interposer Only Flow\)](#) on page 360

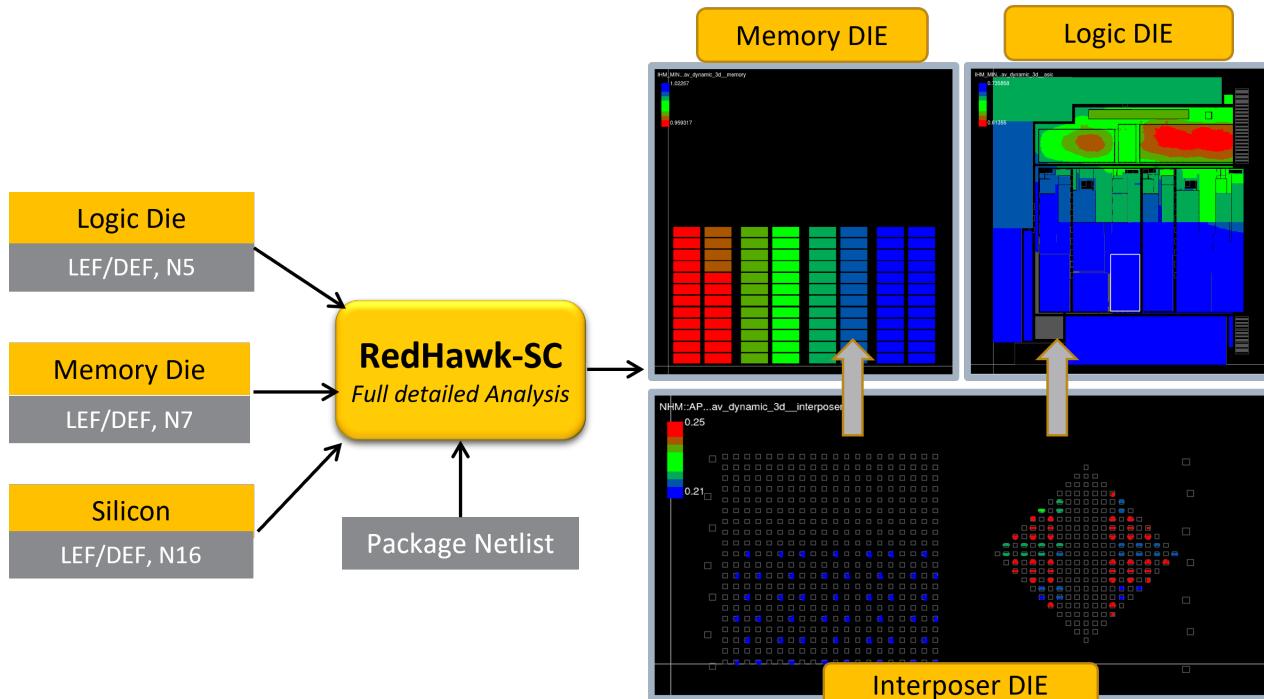
11.6.1. Full Detailed Flow

In this flow, you simultaneously input all the individual chip designs and corresponding process data (that can be of different technology nodes). The flow models coupling between chips in detail. The tool factors the impact of shared power and ground nets and decoupling capacitors in interposer to all the other chips, such as, memory and logic.

However, this methodology has the following limitations:

- Needs more resources compared to other methods
- Requires LEF and DEF files for all the chips, which may not be available if the chip is an IP.

The following figure is a simple representation of the RedHawk-SC full detailed flow.



If the design database is available for all the chips and resources are not limited, you can analyze all the chips together in a single run.

To enable the full detailed flow, set the `detailed` key to `True` for each chip under the `multichip_views` dictionary of the `create_multichip_analysis_view` command as shown in the following example:

```
multichip_views = [
    {'chip_name': 'memory', 'sv': sv_static_memsoc,
     'scn': scn_static_memsoc, 'detailed': True},
    ...
    {'config_view': multichip_config}
]

av_static_3d =
db.create_multichip_analysis_view(multichip_views=multichip_views,
settings={'force': 'True', 'keep_stats_level': 'Full'},
tag='av_static_3d', options=options)
```

The following snippet example shows a multichip command script to execute the full detailed flow of multichip analysis. You must create the base views for the individual chips and interposers before executing this script.

In this example, the multichip system contains three chips, CHIP1, CHIP2, and Interposer. The Interposer is connected to a package (file). The tool performs detailed analysis for all these chips, that is, uses fully extracted circuits for all the chips and simulates them together. The data of all the instances of each chip, such as voltage waveforms and node voltages, are available.

```

multichip_views_pkg = [
    {'chip_name': 'CHIP1', 'dv': dv_chip1},
    {'chip_name': 'CHIP2', 'dv': dv_chip2},
    {'chip_name': 'Interposer', 'dv': dv_interposer},
    {'config_view': multichip_config},
]

pkg=package.parse_ploc_spice_func
('wrapper.sp', multichip_views=multichip_views_pkg)

pkg_sv= db.create_package_simulation_view(package=pkg,
multichip_views, tag='pkg_sv', options=options)

multichip_views =[

    #Detailed analysis for all chips
    {'chip_name':'CHIP1','sv':sv_chip1,
     'scn':scn_no_prop_chip1, 'detailed': True},
    {'chip_name':'CHIP2','sv':sv_chip2,
     'scn':scn_no_prop_chip2, 'detailed': True},
    {'chip_name':'Interposer','sv': sv_interposer,
     'scn':scn_noProp_interposer, 'detailed': True},
    {'config_view': multichip_config},
    {'coupling_view':multichip_coupling_view},
    {'package_simulation_views': [pkg_sv]}
]

#MultichipAnalysisView
av_3d=db.create_multichip_analysis_view (multichip_views=multichip_views,
settings={'duration':10e-9, 'step_size':30e-12, 'keep_stats_level':'Full'},
tag='av_3d', options=options)

#get individual chip AnalysisViews (single chip) using get_chip_view
av_top1=av_3d.get_chip_view(Chip('CHIP1'))
av_top2=av_3d.get_chip_view(Chip('CHIP2'))
av_inter=av_3d.get_chip_view(Chip('Interposer'))

# Power MultichipElectromigrationView
emv_3d_dc = db.create_multichip_electromigration_view(av_3d, tag='emv_3d_dc',
options = options, settings={'mode':'DC'})

#get individual chip ElectromigrationViews (single chip) using get_chip_view
emv_dc_top1 = emv_3d_dc.get_chip_view(Chip('CHIP1'))
emv_dc_top2 = emv_3d_dc.get_chip_view(Chip('CHIP2'))
emv_dc_interposer = emv_3d_dc.get_chip_view(Chip('Interposer'))

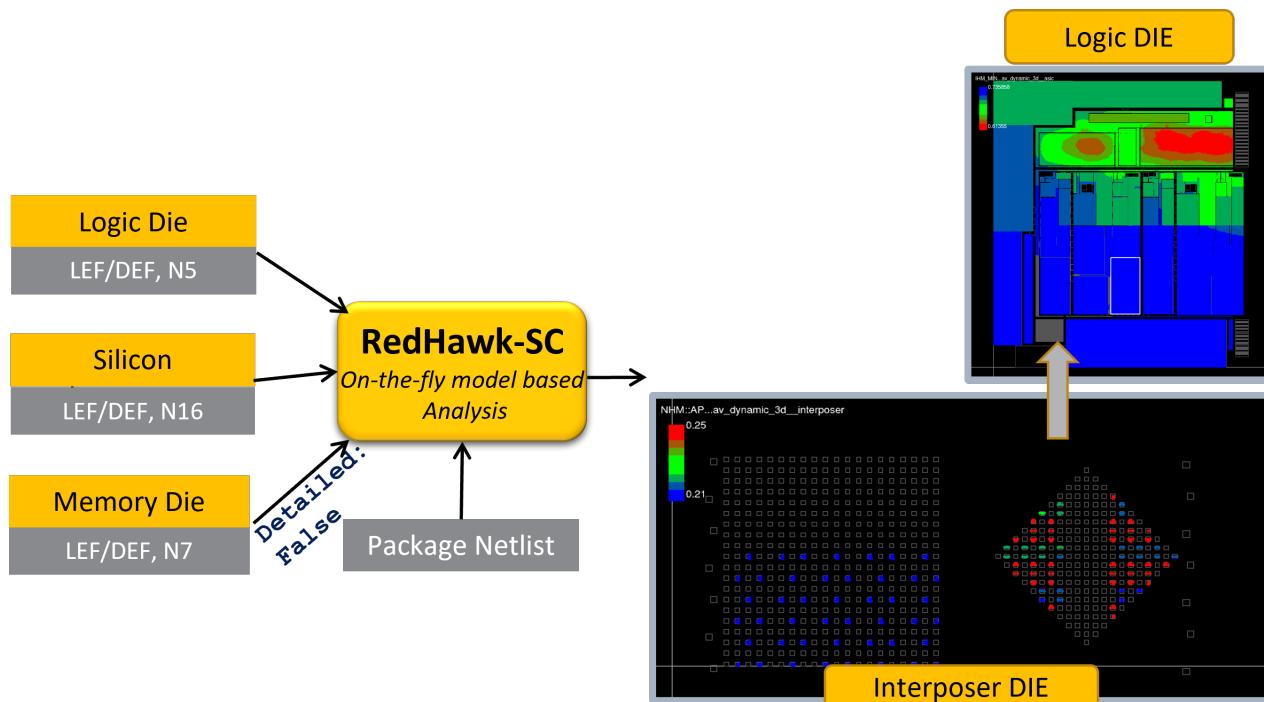
```

11.6.2. On-the-Fly Model Based Flow

This flow is suitable when you analyze only one or few chips in detail, and can use reduced models for other chips of the multichip system. Therefore, this method has fast turn around time (TAT). The tool automatically generates reduced chip model on the fly that are used for the analysis, and analyzes the internally reduced chip models with the impedance (R, L, C) network and currents. Coupling between chips is modeled in detail.

However, this methodology requires LEF and DEF files for all the chips, which may not be available if the chip is an IP.

The following figure is a simple representation of the RedHawk-SC on-the-fly model based flow. During multichip analysis, the tool creates a reduced model of the **Memory Die** and performs full detailed analysis only for the **Logic Die**.



To enable the chips to be reduced, set the `detailed` key to `False` for those chips under the `multichip_views` argument of the `create_multichip_analysis_view` command. In the following example, the chips, `memory` and `interposer`, are reduced.

When `detailed` is set to `False`, the tool performs reduced analysis for that chip.

```

multichip_views =
[{'chip_name': 'memory', 'sv': sv_static_memsoc,
 'scn': scn_static_memsoc, 'detailed': False},
 {'chip_name': 'asic', 'sv': sv_static_generic,
 'scn': scn_static_generic, 'detailed': True},
 {'chip_name': 'interposer', 'sv': sv_static_si_interposer,
 'scn': None, 'detailed': False},
 {'config_view': multichip_config}]

```

The following snippet example shows a multichip command script to execute the on-the-fly model-based flow of multichip analysis. You must create the base views for the individual chips and interposers before executing this script. The multichip system contains three chips, `CHIP1`, `CHIP2`, and `Interposer`. The `Interposer` is connected to the package. The tool performs detailed analysis for `CHIP2` and `Interposer`, and generates a reduced model of `CHIP1`.

```

multichip_views_pkg = [
    {'chip_name': 'CHIP1', 'dv': dv_chip1},
    {'chip_name': 'CHIP2', 'dv': dv_chip2},
    {'chip_name': 'Interposer', 'dv': dv_interposer},
    {'config_view': multichip_config}
]

```

```

pkg=package.parse_ploc_spice_func('wrapper.sp',
multichip_views=multichip_views_pkg)
pkg_sv= db.create_package_simulation_view(package=pkg, multichip_views,
tag='pkg_sv', options=options)

multichip_views =
[
    #On the fly Reduction for CHIP1
    {'chip_name':'CHIP1','sv':sv_top,'scn':scn_no_prop_top, 'detailed': False},
    {'chip_name':'CHIP2','sv':sv_top,'scn':scn_no_prop_top, 'detailed': True},
    {'chip_name':'Interposer','sv': sv_interposer,
     'scn':scn_noProp_interposer, 'detailed': True},
    {'config_view': multichip_config},
    {'coupling_view':multichip_coupling_view},
    {'package_simulation_views': [pkg_sv]}
]

# MultichipAnalysisView
av_3d=db.create_multichip_analysis_view(multichip_views=multichip_views,
settings={'duration':10e-9, 'step_size':30e-12, 'keep_stats_level':'Full'},
tag='av_3d', options=options)

#get individual chip AnalysisViews(single chip) using get_chip_view
av_top2=av_3d.get_chip_view(Chip('CHIP2'))
av_inter=av_3d.get_chip_view(Chip('Interposer'))

# Power MultichipElectromigrationView
env_3d_dc = db.create_multichip_electromigration_view
(av_3d, tag='env_3d_dc', options = options, settings={mode:'DC'})

#get individual chip ElectromigrationViews (single chip) using get_chip_view
env_dc_top2 = env_3d_dc.get_chip_view(Chip('CHIP2'))
env_dc_interposer = env_3d_dc.get_chip_view(Chip('Interposer'))

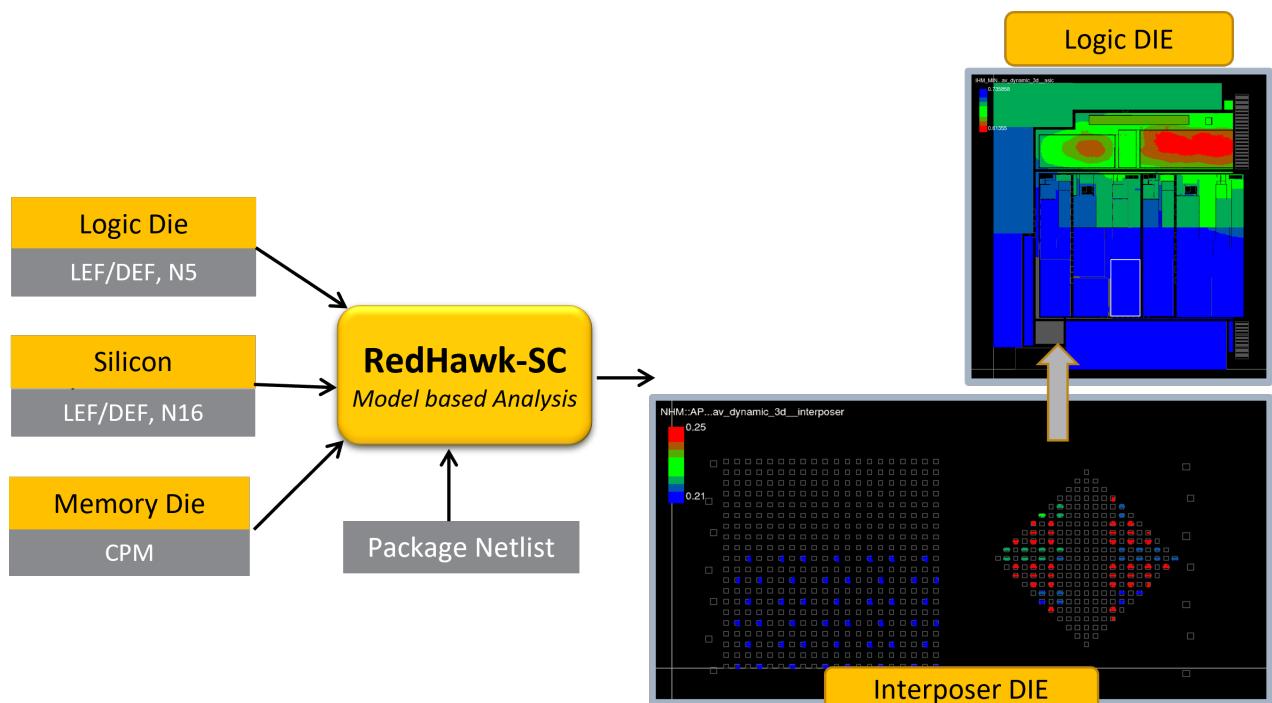
```

11.6.3. Model Based Approach

You use this flow when you do not have access to the complete design database (such as, LEF and DEF data) for any of the individual chips of the multichip system. This is useful when your multichip system includes memory intellectual property chips (IPs) from external vendors. This flow uses the reduced model (CPM) of such chips. The CPM can be generated by the RedHawk-SC tool with R, L, and C network and current profile. This enables simple hand-off and fast turn-around-time.

It is difficult to execute this flow for designs without an interposer (such as, WoW and SoIC) where all the chips are not connected to a single die.

The following figure is a simple representation of the RedHawk-SC model based flow. The tool uses a reduced model (CPM) of the **Memory Die** and performs full detailed analysis only for the logic die and interposer.



The following example shows a typical script to execute the model-based flow of multichip analysis. CHIP2 is replaced with its CPM model. You must input the wrapper package (of the dies with reduced models) to the PackageSimulationView.

```

sv_interposer_cpm = db.create_simulation_view
(ev_interposer, settings={'enable_reduction':'False'},
options=options, tag='sv_interposer_cpm', dv=dv_interposer)

#Input wrapper package (of the chip with reduced model)
#to PackageSimulationView
multichip_views_pkg = [
    {'chip_name': 'CHIP1', 'dv': dv_chip1},
    {'chip_name': 'Interposer', 'dv': dv_interposer},
    {'config_view': multichip_config}
]

pkg=package.parse_ploc_spice_func
('wrapper.sp', multichip_views=multichip_views_pkg)

pkg_sv= db.create_package_simulation_view(package=pkg,multichip_views,
tag='pkg_sv', options=options)

multichip_views =
[{'chip_name':'CHIP1','sv':sv_top,'scn':scn_no_prop_top, 'detailed': True},
 {'chip_name':'Interposer','sv': sv_interposer,
 'scn':scn_noProp_interposer, 'detailed': True},
 {'config_view': multichip_config},

 # CPM model used for CHIP2
 {'coupling_view':multichip_coupling_view},
 {'package_simulation_views': [pkg_sv]}]

```

```

]

# MultichipAnalysisView
av_3d=db.create_multichip_analysis_view(multichip_views=multichip_views,
settings={'duration':10e-9,'step_size':30e-12,'keep_stats_level':'Full'},
tag='av_3d', options=options)

# get individual chip AnalysisViews (single chip) using get_chip_view
av_top1=av_3d.get_chip_view(Chip('CHIP1'))
av_inter=av_3d.get_chip_view(Chip('Interposer'))

# Power MultichipElectromigrationView
emv_3d_dc = db.create_multichip_electromigration_view(
(av_3d, tag='emv_3d_dc', options = options, settings={'mode':'DC'}))

# get individual chip ElectromigrationViews (single chip) using get_chip_view
emv_dc_top1 = emv_3d_dc.get_chip_view(Chip('CHIP1'))
emv_dc_interposer = emv_3d_dc.get_chip_view(Chip('Interposer'))

```

11.6.4. Interposer PDN Quality Checks (Interposer Only Flow)

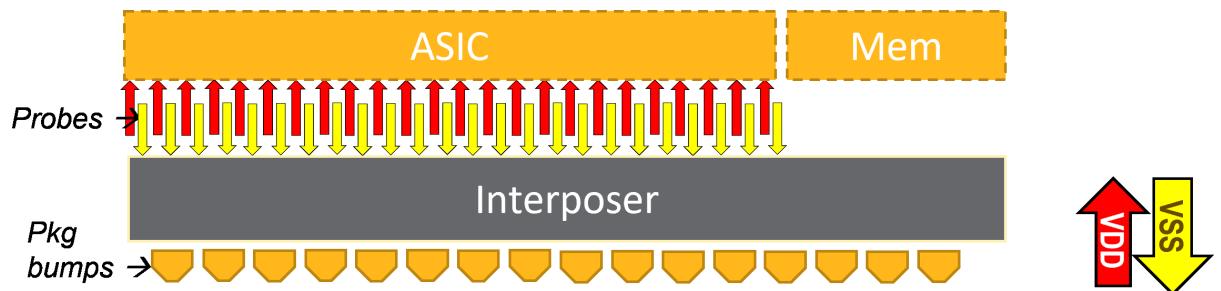
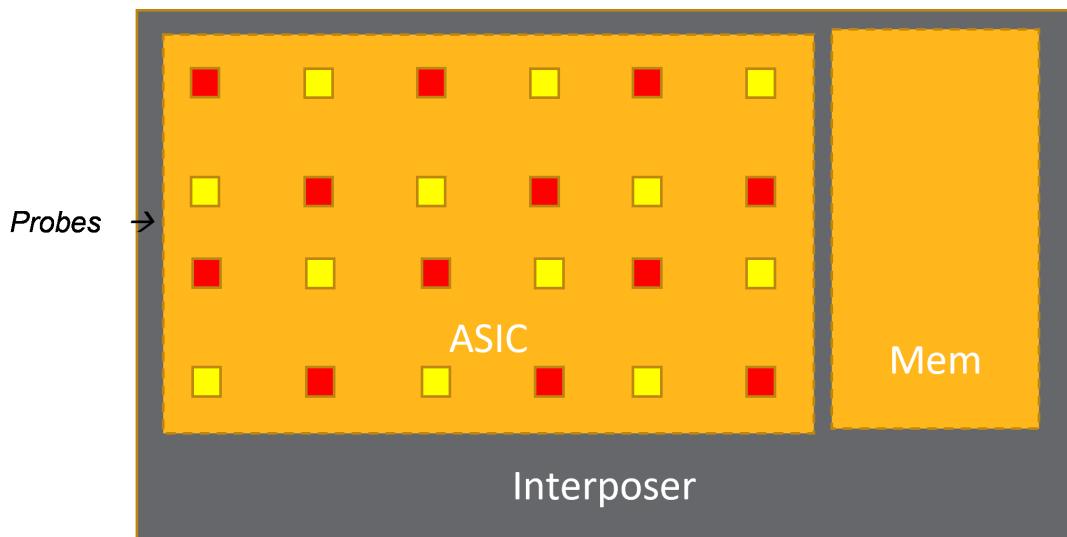
The following custom flows are available to check interposer PDN quality checks:

- [Static Flow With Constant Current](#) on page 361
- [Dynamic Flow With Current Waveforms](#) on page 363

These check grid density, bump placement, and connection point placements of an interposer chip during early analysis. Only the following data is required to get the grid analysis metrics:

- Interposer DEF
- Bump locations
- Connection point locations
- TSV Sub-circuit (optional)

The following figure shows probes inserted at connection points of an interposer with other chips, and the package bumps.



Static Flow With Constant Current

The following figure shows the high-level flow to check interposer PDN quality with constant current input. The example following the figure shows the command script.

Find all the connection points which are expected to connect to the DIE and create probes on that location



Place a constant value current source at the probes.



Perform a static solve using these probes with constant currents.



Generate unique power and ground heatmaps to show the relative weaknesses

```
#Static flow with constant current source

#static total bump current input is of float or int type
current_waveforms_static = {
    'IF_SOC1' : { Net('VDD_SOC'): 5.0 , Net('VSS'): 5.0 },
    'IF_SOC2' : { Net('VDD_SOC_2'): 5.0 , Net('VSS'): 5.0 },
    'IF_SOC3' : { Net('VDD_SOC_3'): 5.0 , Net('VSS'): 5.0 },
}

#Get micro bump information
multichip_config = db.create_multichip_config_view(config_file,...)

#Creating probes and probe sources (current information)
static_probes_data = generate_multichip_current_probes(multichip_config,
Chip('InFO'), current_waveforms=current_waveforms_static)

dv_settings = dict(
    die_name = 'die_connecting_to_package',
    die_sub_id = die_connecting_to_package_subid,
    #Plocs with C4 bumps
    interfaces = ['Interface_of_die_connecting_to_package'],
)
dv = db.create_modified_design_view(..., settings=dv_settings)

#Inputting probes into ExtractView
```

```

ev_settings = dict(..., probes = static_probes_data)
ev =db.create_extract_view(...,design_view=dv,settings=ev_settings)

#Adding probe sources to ScenarioView. This creates external current sources
#at a given location. If instances in the bottom die are connected to
#package, the scenario type need not be external. Both regular ScenarioView
#and NPV ScenarioView with probe_sources are supported.
scn = db.create_scenario_view(..., design_view=dv, extract_view=ev,
scenario_type="External", settings = {'probe_sources':static_probes_data})

#AnalysisView node voltages and edge currents show chip PDN quality.
av = db.create_analysis_view(..., scn)

```

Dynamic Flow With Current Waveforms

The following figure shows the high-level flow to check interposer PDN quality with dynamic current waveform input. The example following the figure shows the command script.

Find all the connection points which
are expected to connect to the DIE
and create probes on that location



Stamp total demand or total bump
current waveforms of nets at the
probes.



Perform Dynamic solve using these
probes



Generate unique power and ground
heatmaps to show the relative
weaknesses

```

#Dynamic flow with current waveforms as input

#dynamic total bump current input
current_waveforms_dynamic = {
    'IF_SOC1':{Net('VDD_SOC'):Waveform([(0.0,5.0)]),
    Net('VSS'):Waveform([(0.0,5.0)])},

```

```

'IF_SOC2':{Net('VDD_SOC_2'):Waveform([(0.0,5.0)]),
Net('VSS'):Waveform([(0.0,5.0)])},
'IF_SOC3':{Net('VDD_SOC_3'):Waveform([(0.0,5.0)]),
Net('VSS'):Waveform([(0.0,5.0)])},
}

#Get micro bump information
multichip_config = db.create_multichip_config_view(config_file,...)

#Creating probes and probe sources (current information)
dynamic_probes_data = generate_multichip_current_probes(multichip_config,
Chip('InFO'), current_waveforms=current_waveforms_dynamic)

dv_settings = dict(
    die_name = 'die_connecting_to_package',
    die_sub_id = die_connecting_to_package_subid,
    #Plocs with C4 bumps
    interfaces = ['Interface_of_die_connecting_to_package'],
)
dv = db.create_modified_design_view(...,settings=dv_settings)

#Inputting probes into ExtractView
ev_settings = dict(..., probes = dynamic_probes_data)
ev =db.create_extract_view(...,design_view=dv,settings=ev_settings)

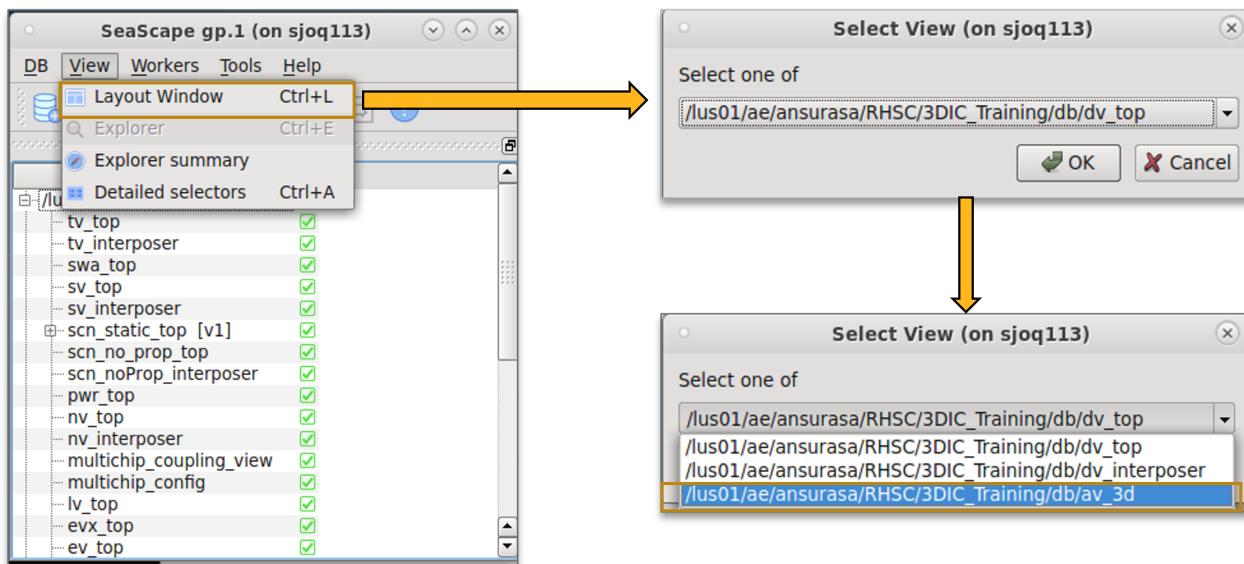
#Adding probe sources to ScenarioView. This creates external current sources
#at a given location. If instances in the bottom die are connected to
#package, the scenario type need not be external. Both regular ScenarioView
#and NPV ScenarioView with probe_sources are supported.
scn = db.create_scenario_view(...,design_view=dv,extract_view=ev,
scenario_type="External",settings = {'probe_sources':dynamic_probes_data})

#AnalysisView node voltages and edge currents show chip PDN quality.
av = db.create_analysis_view(...,scn)

```

11.7. Multichip-Specific GUI Features

To open the multichip GUI in RedHawk-SC, select **View>Layout Window** from the menu and then select the MultichipAnalysisView from the from drop-down list in the **Select View** dialog box.

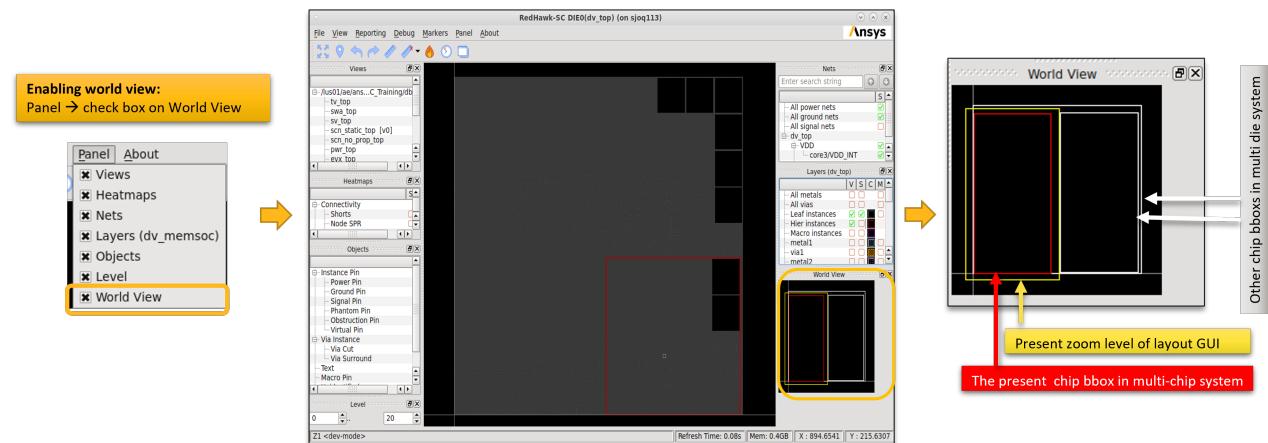


This automatically opens individual layout GUIs for each chip in your design.

World View

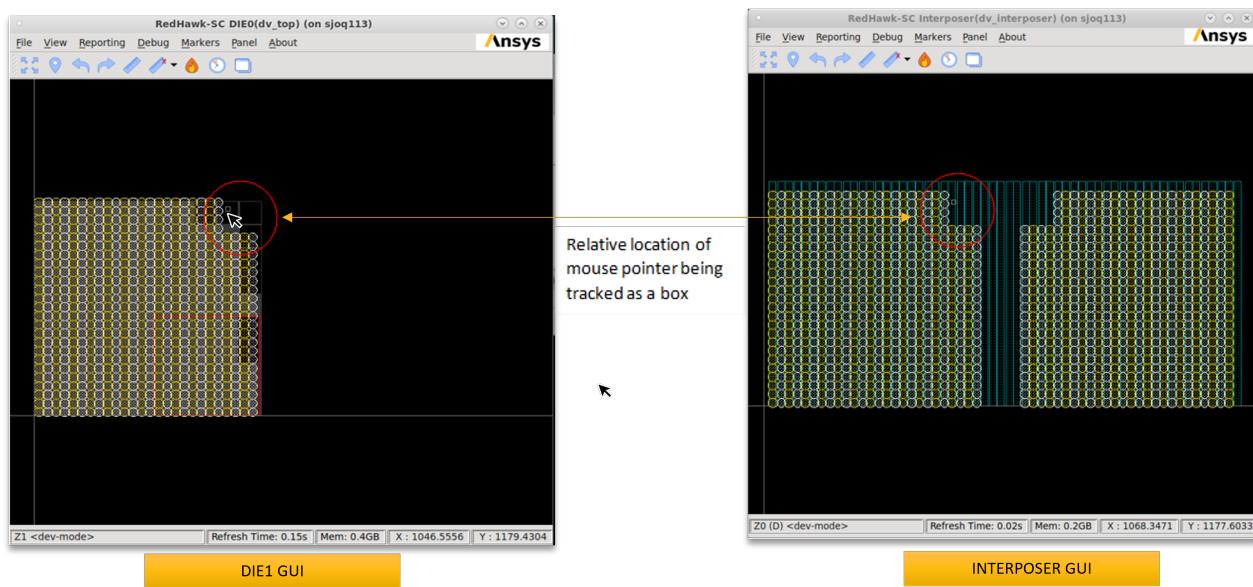
World View shows the relative position of an individual chip in the multi-chip system. This is a simplified layout of chip boundaries, where the present chip boundary is represented as a red rectangle and other chip boundaries are shown as white rectangles. The yellow boundary shows the present zoom level.

To enable viewing World View, select **Panel>World View**.



Relative Mouse Pointer Tracking

The mouse pointer in one chip's layout is automatically tracked in the other chip's layout. The locations of the box present in the other chip's layout are relative and depend on each chip's orientation.



11.8. Reporting Multichip Analysis Results

The following sections describe the functions to query multichip views and report multichip analysis results.

- [Querying MultichipConfigView](#) on page 366
- [Querying MultichipAnalysisView](#) on page 367
- [Reporting EMIR Results](#) on page 369

Querying MultichipConfigView

The following examples show the functions to query the MultichipConfigView and the query results:

- `get_chips`

```
>>> multichip_config.get_chips()
[Chip('Interposer'), Chip('DIE0'), Chip('DIE1')]
```

- `get_dies`

```
>>> multichip_config.get_dies()
[Die('Galaxy', 1), Die('GalaxyInterposer', 2)]
```

- `get_package_connections`

```
>>> multichip_config.get_package_connections()
[(Chip('Interposer'), DieInterface(Die('GalaxyInterposer', 2), 'IF_PKG'))]
```

- `get_inter_chip_connection_parasitics`

```
>>> mccfg.get_inter_chip_connection_parasitics()
{'CP_1': {'capacitance': 1.0000000036274937e-15,
```

```
'height': 5.0,
'inductance': 9.99999960041972e-13,
'length': 20.0,
'resistance': 9.99999747378752e-05,
'width': 20.0},
'RHSCDEFAULTTCP': {'capacitance': 0.0,
'height': 0.0,
'inductance': 0.0,
'length': 1.0,
'resistance': 9.99999974752427e-07,
'width': 1.0}}
```

Querying MultichipAnalysisView

The following examples show how to query the MultichipAnalysisView and the query results:

- `get_multichip_info`

Obtains multichip information from AnalysisView. For example,

```
>>> av_3d.get_multichip_info().keys()
['DIE0', 'DIE1', 'Interposer']

>>> av_3d.get_multichip_info()['DIE0']
{'dv': <gp.ModifiedDesignView object at 0x7f8f0ac4eb50>, 'chip_id': 0,
 'angle': 0.0, 'flip': False, 'coord': RealCoord(22.000000, 0.000000)}

>>> av_3d.get_multichip_info()['DIE1']
{'dv': <gp.ModifiedDesignView object at 0x7f8f0ac4eb50>, 'chip_id': 1,
 'angle': 180.0, 'flip': True, 'coord': RealCoord(2652.829956, -12.999900) }

>>> av_3d.get_multichip_info()['Interposer']
{'dv': <gp.ModifiedDesignView object at 0x7f8f0ac541d0>, 'chip_id': 2,
 'angle': 0.0, 'flip': False, 'coord': RealCoord(0.000000, 0.000000)}
```

- `get_average_bump_voltages` returns the average bump voltages of the system per chip per interface per net. Returned object is a dict. For example,

```
multichip_analysis_view.get_average_bump_voltages(base_level=False)
```

Here, `base_level` is `False` by default. When enabled, the waveform is subtracted from the ideal net voltage source.

The following show the output format and a typical output example returned by the `get_average_bump_voltages` function.

```
{(Chip('chip_name'), 'Interface_name'): {Net('net_name'): waveform}}
```

```
{(Chip('DIE0'), 'IF_D1'): {Net('VDD'): Waveform([(0.0, 0.8)])}}
```

- `get_total_bump_currents` returns the total bump currents of the system per chip per interface per net. Returned object is a dict. For example,

```
total_bump_current = multichip_analysis_view.get_total_bump_currents()
```

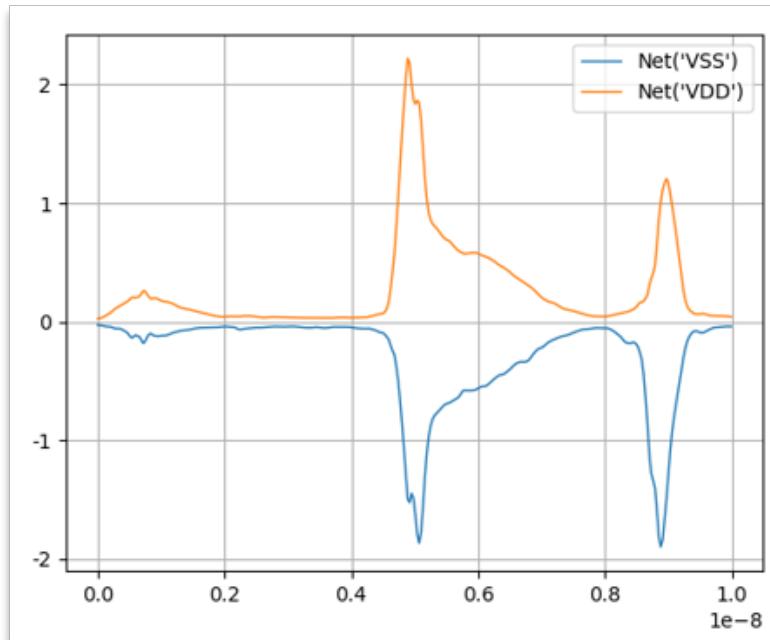
The following show the output format and a typical output example returned by the `get_total_bump_currents` function.

```
{(Chip('chip_name'), 'Interface_name'): {Net('net_name'): waveform}}
```

```
{(Chip('DIE0'), 'IF_D1'): {Net('VDD'): Waveform([(0.0,10)])}}
```

Use the `plot` command to view the output waveform. For example,

```
plot(total_bump_current)
```



- `get_demand_currents` returns the total demand currents seen from the system at the C4 interface. Returned object is a dict. For example,

```
total_demand_currents = multichip_analysis_view.get_demand_currents()
```

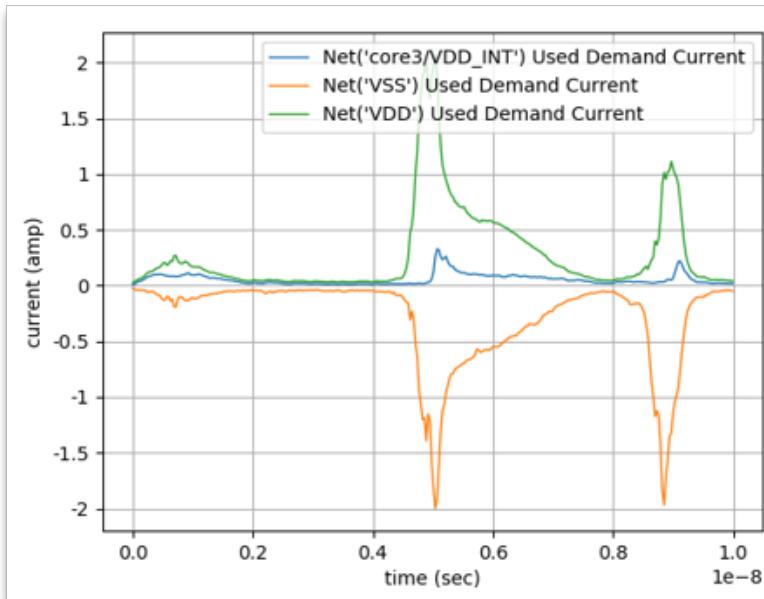
The following show the output format and a typical output example returned by the `get_demand_currents` function.

```
{(Chip('chip_name'), 'Interface_name'): {Net('net_name'): waveform}}
```

```
{(Chip('DIE0'), 'IF_D1'): {Net('VDD'): Waveform([(0.0,10)])}}
```

Use the `plot` command to view the output waveform. For example,

```
plot(total_demand_currents)
```



- `<MultichipAnalysisView>.get_chip_view(Chip('<chip_name>'))`

Returns the AnalysisView object for the specified chip. This enables you to access the individual AnalysisViews of each chip. You can perform all queries available for a single chip, such as, `get_current` and `get_voltage` for the specified chip. `Chip` is the chip object for which the AnalysisView is required. For example,

```
>>>av_memory = av_3d.get_chip_view(Chip('memory'))
>>>av_asic = av_3d.get_chip_view(Chip('asic'))
>>>av_interposer = av_3d.get_chip_view(Chip('interposer'))
```

Reporting EMIR Results

You cannot directly obtain EMIR reports from the MultichipAnalysisView. To get the EMIR reports, you need to get the individual chip AnalysisViews from the MultichipAnalysisView. For example,

```
emir_reports.write_all_instance_voltages(av_asic)
emir_reports.write_em_metal_report(em_asic)
```

11.9. SPR for Multichip Designs

You can determine the shortest path resistance (SPR) through multiple chips of the multichip system. For an instance pin or node (location) of one chip, the tool traces the full-path SPR up to the C4 bumps that connect to the package.

To enable SPR tracing for multiple chips, use the `perform_multichip_shortest_resistance_path_check` command as shown in the following example:

```
chip_views = [{config_view': config_view,
  'chip_name':chip1, 'ev': ev1, }, ...]
mcev =
```

```
db.perform_multichip_shortest_resistance_path_check(multichip_views=chip_views,
options=options, settings={'include_ron_in_spr': True})
```

The `perform_multichip_shortest_resistance_path_check` command returns a `MultichipExtractView`. The `settings` dict of the command includes these keys with following default values:

```
{
    'trace_to' : None,
    'include_ron_in_spr' : False,
    'heatmaps' : {'node_spr': True, 'instance_pin_spr': True, 'probes': True}
}
```

To support querying SPR data for multichip designs, the following functions are available to query `MultichipExtractView`. For more details, use the `help` command at the tool command prompt:

```
help(SeaScapeDB.create_multichip_extract_view)
```

- `get_instance_pin_spr(self, chip_name, instance, pin, spr_report_type='worst')`
- `get_shortest_resistance_path(self, chip_name, layer=None, x=None, y=None, net=None, instance=None, pin=None, analysis_view=None, report_type=None, spr_report_type='worst', voltage_report_type='worst', search_radius=10.0)`
- `get_instance_pin_spr_heatmap(self, chip_name)`
- `get_spr_heatmap(self, chip_name=None)`
- `get_spr_histograms(self, chip_name)`
- `get_instance_pin_spr_histograms(self, chip_name)`

You can view the SPR paths and SPR heatmaps of the different chips of a multichip system in the GUI:

- [Viewing Multichip SPR Data in GUI](#) on page 370

Viewing multichip SPR paths and heatmaps is similar to viewing single chip SPR paths and heatmaps with slight differences.

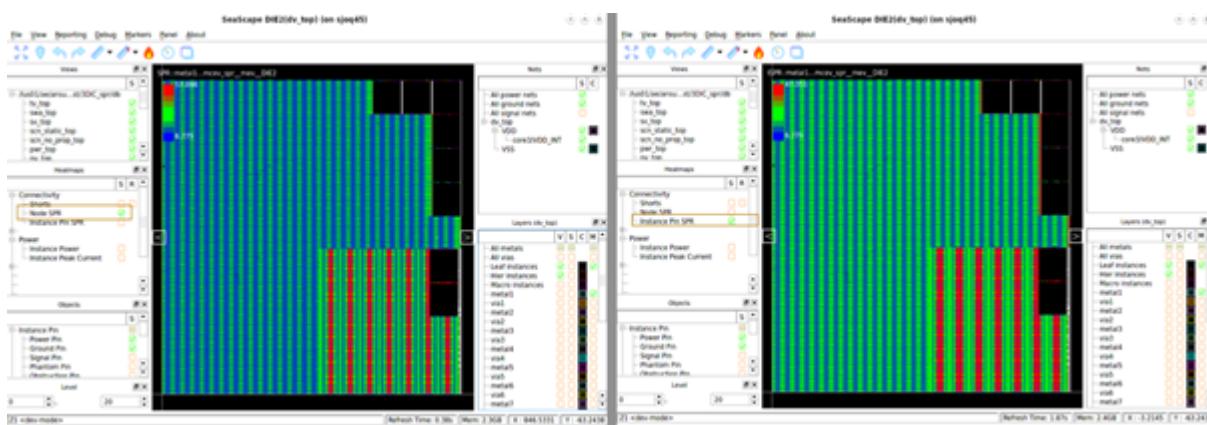
11.9.1. Viewing Multichip SPR Data in GUI

To view multichip SPR data,

1. Select the `MultichipExtractView` from the drop-down list in the **Select View** dialog box while opening the **Layout Window**. See [Viewing the Layout Window](#) on page 468.

This opens the design layouts of all the individual chips of the multichip system. In each of the design layouts, ensure that `MultichipExtractView` of the individual chip is selected under the **Views** panel.

2. Based on the SPR data from `MultichipExtractView`, the layout GUI has pre-populated SPR heatmaps as shown in the following figure. To view these heatmaps, select **Node SPR** or **Instance Pin SPR** in **Heatmaps**.



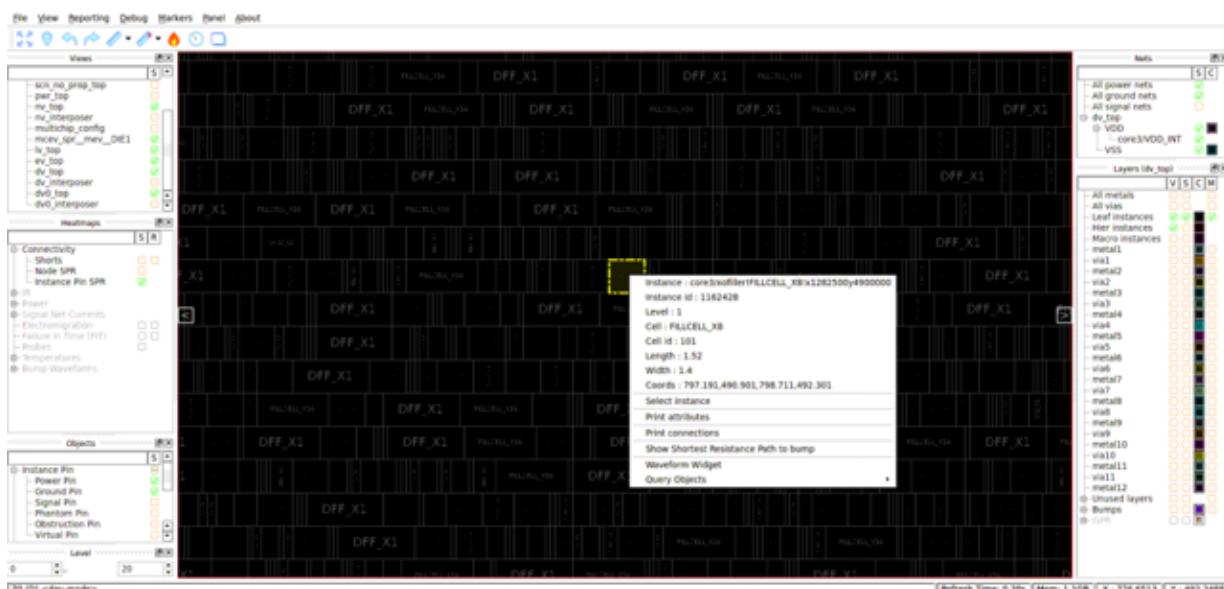
The following sections describe how to view traced instance and node SPR paths in GUI:

- [Viewing Pin SPR Values in a Multichip System](#) on page 371
- [Viewing Multichip SPR Path For a Location](#) on page 373

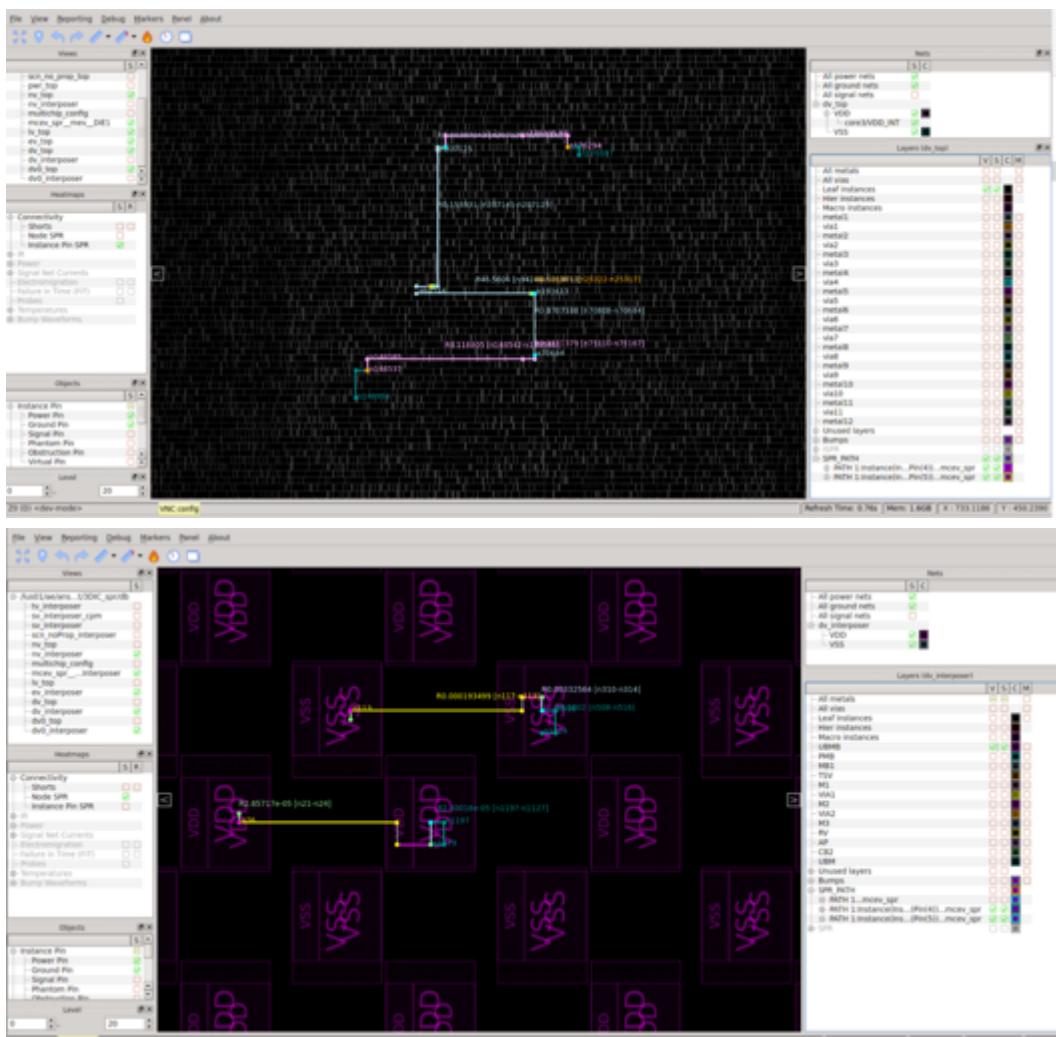
Viewing Pin SPR Values in a Multichip System

To trace the SPR path for an instance of a particular chip in GUI, follow these steps:

1. Select the instance in the layout of the individual chip (for example, DIE1).
2. Right-click and select **Show Shortest Resistance Path to bump** as shown:



The GUI displays the SPR path from instance PG pins to bumps across the design layouts of different chips. For example, the following figures show the complete SPR paths traced from PG pins of an instance of chip, DIE1, to C4 bumps of chip, Interposer.



The SPR path can span through multiple chips as shown. A browser showing instance pin SPR path data in tables is automatically opened. [Viewing Pin SPR Values](#) on page 77.

Each row of the table corresponds to a line segment in the traced path. The multichip SPR table contains an additional column head, **Chip**, than the single-chip SPR table. This column shows the individual chip (of the multichip system) where the line segment is located.

From: 0001 Instance('core0/xxFiller/000CELL_00')/Pin('VSS') RealCoord(778.435000,492.305000) Layer('metal1') Net('VSS') Cost(0.51700)

To: Bump Interposer Pin(VDD_P00_23) RealCoord(748.090000,539.060000) Layer('000B') Net('VSS') Cost(0.00000)

Table

Location(x,y)	Chip	Layer	Cost(mil)	ResDiff	Length(m)	Width(m)	Drop(m)	DropOff	Net	Comment
1 776..0003	0HE1	metal1	9.52	—	—	—	—	—	VSS	—
2 779..0003	0HE1	metal1	7.72	1.795	2.795	0.17	—	—	VSS	wire: metal1
3 779..0003	0HE1	metal2	7.02	0.7	—	—	—	—	VSS	via: via1_4_3_27
4 779..0003	0HE1	metal2	6.94	0.084	0.325	0.14	—	—	VSS	wire: metal2
5 779..0003	0HE1	metal3	6.38	0.16	—	—	—	—	VSS	via: via2_4_3_25
6 779..0003	0HE1	metal3	5.82	0.16	—	—	—	—	VSS	via: via2_3_3_25
7 779..0003	0HE1	metal4	5.78	0.034	0.05	0.16	—	—	VSS	wire: metal4
8 779..0003	0HE1	metal5	5.43	0.35	—	—	—	—	VSS	via: via4_0_3_13
9 779..0003	0HE1	metal6	4.97	0.459	—	—	—	—	VSS	via: via5_0_3_13
10 779..0003	0HE1	metal7	4.32	0.459	—	—	—	—	VSS	via: via5_0_3_13
11 779..0003	0HE1	metal8	4.51	0.001	0.63	0.4	—	—	VSS	wire: metal7
12 779..0003	0HE1	metal9	3	1.514	—	—	—	—	VSS	via: via7_0_3_3
13 779..0003	0HE1	metall0	3	0.061	0.62	0.4	—	—	VSS	wire: metal8
14 779..0003	0HE1	metall1	2.99	0.011	0.4	0.8	—	—	VSS	wire: metal8
15 779..0003	0HE1	metall2	1.47	1.514	—	—	—	—	VSS	via: via10_0_3_4
16 779..0003	0HE1	metall3	1.47	0.001	0.62	0.4	—	—	VSS	wire: metal9
17 780..0003	0HE1	metall4	1.46	0.011	0.4	0.8	—	—	VSS	wire: metal9
18 780..0003	0HE1	metall5	1.08	0.378	—	—	—	—	VSS	via: via10_0_3_2
19 780..0003	0HE1	metall6	1.08	0.087	0.84	2.8	—	—	VSS	wire: metal10
20 780..0..0.905	0HE1	metall7	0.923	0.156	28.0	4.0	—	—	VSS	wire: metal10
21 780..0..521.6	0HE1	metall8	0.917	0.004	0.695	4.0	—	—	VSS	wire: metal10
22 782..4..521.6	0HE1	metall9	0.962	0.017	1.6	1.148	—	—	VSS	wire: metal10
23 782..4..521.6	0HE1	metall10	0.522	0.378	—	—	—	—	VSS	via: via10_0_4_3

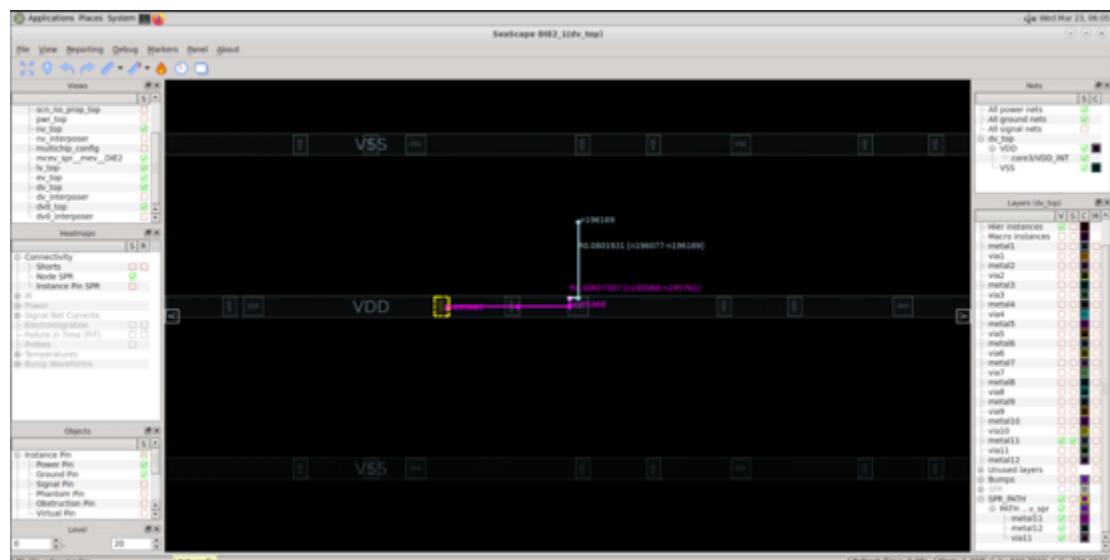
Locatoin(x,y) ▾ Contain ▾

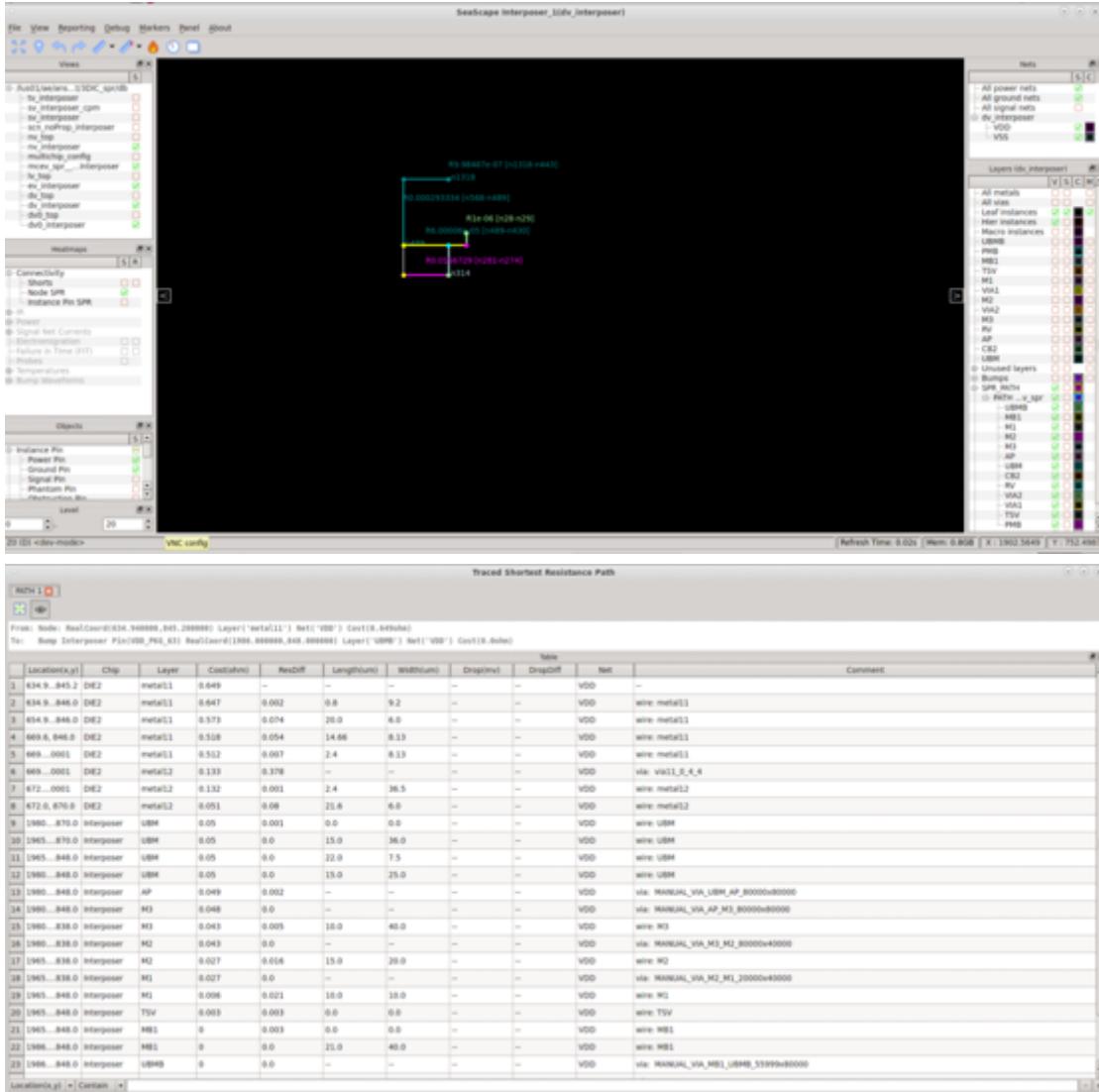
Apply

Viewing Multichip SPR Path For a Location

With node SPR information shown on PG routes, you can trace SPR path from those nodes. After selecting a PG route, right-click **Show Shortest Resistance Path to Bump**.

Figure 68. SPR Path from a Node Location Spanning Across Multiple Chips





12: Package and Board Analysis in RedHawk-SC

RedHawk-SC performs package and board-level simulation for dynamic voltage drop analysis. A detailed model including the interconnect resistances, parasitic capacitances, and inductance of the package is necessary when analyzing the power distribution network. Package models are generated using package extraction tools and are normally available as SPICE netlists. These package models are imported into RedHawk-SC during simulation.

This chapter describes various package models and their implementation for effective simulation using RedHawk-SC:

- [RLC Model](#) on page 375
- [RLCK SPICE Netlist](#) on page 377
- [S-Parameter Model](#) on page 380
- [Important Functions of Package Module](#) on page 383

Note: For better accuracy of RedHawk-SC simulation, it is recommended to perform port grouping in package extraction tools to limit the number of ports to few hundreds to thousands. If the package port count exceeds than ten thousand, there is less improvement in the final dynamic voltage drop.

12.1. RLC Model

A simple package circuit model can be annotated in RedHawk-SC in two ways:

- Connecting lumped package to the die ports
- Connecting per bump package with user defined same RLC to all die ports

Connecting Lumped Package to Die Ports

RedHawk-SC creates the lumped SPICE netlist for the package with user defined RLC values and stitches the same to die ports using the `create_package_wrapper` function from the `package` module. Use the `package_simulation_view` function to create the package-related circuit data and pass it to `AnalysisView` as shown in the following example.

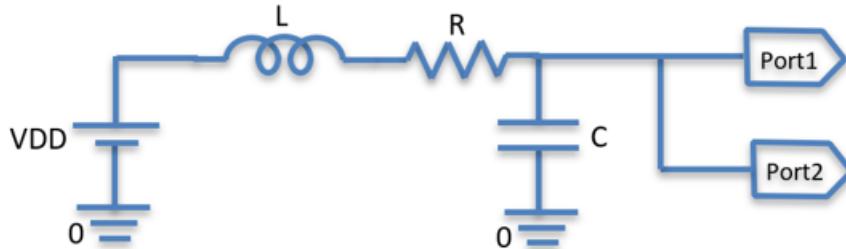
```
>>> package.create_package_wrapper(<ploc_file>, mapping_type='lumped',
net_data=<per_net_RLC info>, pkg_file_name=<output_package.sp file name>)
Where, net_data: List of tuples with PG net, voltage and RLC values.
[(Net1,Voltage1,R1,C1,L1),(Net2,V2,R2,C2,L2)...]

>>> pkg = package.parse_ploc_spice_func(<output_package.sp filename>,
dv=<modifieddesignview>)
>>> pkg_sv = db.create_package_simulation_view(<extract_view>, package=pkg,
, settings=pkg_settings, tag='pkg_sv', options=options)
>>> av = db.create_analysis_view(<simulation_view>, <scenario_view>,
package_simulation_views = [pkg_sv], options=options, tag=tag, ...)

Example:
>>> package.create_package_wrapper('top.ploc', mapping_type='lumped',
net_data=[(Net('VDD'),1.0,1e-3,1e-9,1e-9),( Net('GND'),0.0,1e-3,1e-9,1e-9)]
pkg_file_name='rhsc_spice_wrapper.sp')

>>> pkg = package.parse_ploc_spice_fun('rhsc_spice_wrapper.sp',
```

```
dv=modified_design_view)
>>> pkg_sv = db.create_package_simulation_view(extract_view, package=pkg,
tag='pkg_sv', options=options)
```

Figure 69. Simple RLC Model

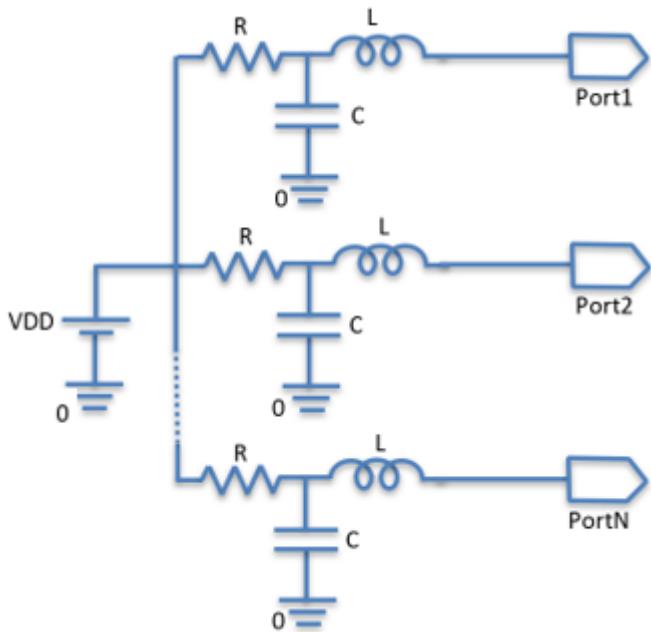
Note: With `create_package_wrapper` function, you can assign different lumped RLC values per domain.

Connecting Per Bump Package to All Die Ports

RedHawk-SC creates the per bump SPICE netlist with user defined RLC values and stitches the same to all die ports using the `create_per_bump_spice_func` function from `package` module. Use the `package_simulation_view` function to create the package-related circuit data and pass it to `AnalysisView` as shown in the following example.

```
>>> pkg_sv = create_package_simulation_view(extract_view,
package=package.create_per_bump_spice_func(dv=modified_design_view,
r=<res per bump in Ohms>, l=<inductance per bump in henrys>,
c=<capacitance per bump in Farads>), tag = <tag>, options=options))

Example:
>>> pkg_sv = create_package_simulation_view(extract_view,
package=package.create_per_bump_spice_func(dv=modified_design_view, r=1.1,
l=1e-10, c=1e-12)
, tag='pkg_sv', options=options))
```

Figure 70. Per Bump Package RLC Model

Note: Voltage supply values which are defined as part of the ScenarioView are used for at the package VRM (VDD definition in the preceding figure). ModifiedDesignView stores the PG voltage source locations (ploc file) that have mapping information to connect the package ports and per bump RLC values. The `create_per_bump_spice_func` picks RLC values from ploc file if available, otherwise uses the RLC values passed to `create_per_bump_spice_func` as arguments.

12.2. RLCK SPICE Netlist

For accurately analyzing the dynamic voltage drop variations across different package ports, the package parasitics must be represented in the form of a subcircuit description following conventional SPICE syntax. Ideal voltages are applied at the package balls through the subcircuit, which is connected to the die during analysis. A list of ports in the subcircuit definition should include all power and ground ports. The subcircuit netlist supports the following linear independent and dependent circuit elements.

Symbol	Element
R	Resistance
L	Inductance
C	Capacitance
K	Mutual inductance
I	Current source
V	Voltage source
E	Linear voltage controlled voltage source
F	Linear current controlled current source
G	Linear voltage controlled current source

Symbol	Element
H	Linear current controlled voltage source
X	Subcircuit instantiation
N	S-parameter model instantiation

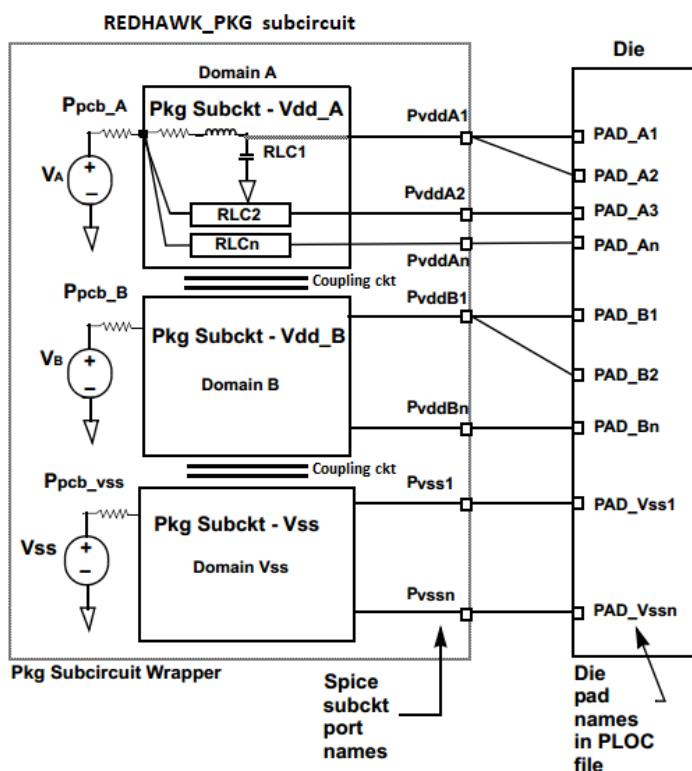
All the elements must be fixed, with no dependencies on any other parameters. In addition, the subcircuit netlist supports the following functions.

Function	Purpose
.inc	Inclusion of other SPICE compatible files
.subckt	Definition of additional subcircuits
.model	Inclusion of other s-parameter touchstone files

The entire off-chip network must be captured in a SPICE subcircuit netlist named `REDHAWK_PKG`. In addition, the power and ground ideal sources must be defined in the top-level netlist.

Note: The mutual inductor coefficient represents the coupling effect among inductors. If the mutual inductance between inductors L1 and L2 is M12, the coefficient K is defined as $M_{12}/\sqrt{L_1 \cdot L_2}$. The mutual inductance coefficient must have an absolute value between 0 and 1.0. In the package SPICE netlist, it appears as `K_L1_L2 L1 L2 0.5`.

Figure 71. Modelling Package Parameters in RedHawk-SC



Example of a top-level subcircuit:

```

* Top level subcircuit
* The file name to be passed to 'package' argument of simulation * view
* All ports are mapped (i.e. connected) to RedHawk-SC pads
*defined in the *.ploc file which passed in modified design view.
* The subcircuit must be named REDHAWK_PKG
.subckt REDHAWK_PKG PvddA1 PvddA2 PvddB1 PvddB2 Pvss1 Pvss2
* PCB voltage supplies
Va Ppcb_A 0 1.1v
Vb Ppcb_B 0 1.0v
Vvss Ppcb_vss 0 0v
* Instantiate domain subcircuits. Connect them to the
* corresponding PCB voltage supplies and output ports of
* REDHAWK_PKG
XVDDA_RLC_Ppcb_A PvddA1 PvddA2 VDDA_RLC
XVDB_RLC ...
XVSS_RLC ...
.ends
* Package RLC for each voltage domain can be captured in a
* separate subcircuit
.subckt VDDA_RLC Ppcb Pdie1 Pdie2
R1 Ppcb N1 1e-9
L1 N1 Pdie1 1e-12
K12 L1 L2 0.3
C1 Pdie1 0 1e-10
R2 Ppcb N2 0.5e-09
L2 N2 Pdie2
.ends
.subckt VDDB_RLC Ppcb Pdie1 Pdie2
...
.ends
.subckt VSS_RLC Ppcb Pdie1 Pdie2
...
.ends

```

For RedHawk-SC to include the subcircuit model in the simulation, the top-level SPICE netlist should be passed using the following command:

```
>>> pkg_sv = create_package_simulation_view(extract_view, package=
    package.parse_ploc_spice_func(<top level package spice netlist>,
    dv=modified_design_view), tag = <tag>, options=options)
```

Example:

```
>>> pkg_sv = create_package_simulation_view(extract_view, package=
    package.parse_ploc_spice_func('top.sp', dv=modified_design_view), tag='pkg_sv',
    options=options)
```

Note:

- RedHawk-SC assigns the voltage at source (PCB side) with ScenarioView voltage levels. The tool does not honor the voltages mentioned in the package netlist. Therefore, to change the ideal voltage, update the `voltage_levels` argument of ScenarioView as it is easy to simulate for different voltage levels without editing the package file and it is flexible to ensure whether package voltage and ScenarioView voltages are matching or not.

- RedHawk-SC accepts the PWL voltage source from the package that is normalized to 0V by subtracting the ideal voltage. For example, ideal voltage of a net is 1V and actual PWL source defined in package netlist as `vvdd n1 0 pwl (0n 1.0 2.3n 1.08 3n 1.1 5.3n 0.92 6n 0.9)`, it can be rewritten as `vvdd n1 0 pwl (0n 0.0 2.3n 0.08 3n 0.1 5.3n -0.08 6n -0.1)`

12.3. S-Parameter Model

As the clock frequency of the chip increases, it is important to capture accurate frequency behavior of the package and PCB over the multi-GHz range. Designers typically utilize full-wave electromagnetic solvers, which create frequency-dependent scattering (S) parameters. The resulting package or PCB model is represented by a frequency-domain S-parameter model, in Touchstone format. This feature allows you to utilize these S-parameters in RedHawk-SC dynamic simulations.

Modelling Methodology

The S-parameter model describes, at each specified frequency point, the voltage and current relationships among all the ports of a black-box network. Conceptually, the S-parameters model the transfer of power from the source to the load, in terms of the incident (a) and reflected (b) waves, $\mathbf{b} = \mathbf{S}\mathbf{a}$ at a particular frequency. Both \mathbf{a} and \mathbf{b} are vectors of size N, and S is a complex N x N matrix, where N is the number of ports. The k-th component of the incident wave a_k and the k-th component of the reflected wave b_k is related to the port voltages V_k and currents I_k through the following equations:

$$a_k = \frac{V_k + Z_0 I_k}{2\sqrt{Z_0}} \quad b_k = \frac{V_k - Z_0 I_k}{2\sqrt{Z_0}}$$

The definition of the S-parameters includes the value of the normalization (reference) impedance Z_0 , as shown in the following example in Touchstone format ($Z_0=2$ Ohms):

```
# HZ S RI R 2.000
```

Note: RedHawk-SC supports scalar reference impedance only, therefore, all ports have the same reference impedance.

In order to perform dynamic modeling of package and PCB using S-parameters, ports need to be defined at the package die pad, BGA pad locations, and/or board voltage regulator module (VRM) location. In package or PCB power delivery network modeling, a port is usually defined across a set of VDD (power) nodes and a set of VSS (ground) nodes. In network analysis terminology, the Vss nodes are called the reference nodes of the defined port.

A typical port setup scheme for a flip-chip package is as follows:

1. Partition the package die pad area into N partitions.
2. Set up a port in each partition between VDD nodes and VSS nodes.
3. On the package BGA side, set up ports as required between all VDD nodes and VSS nodes.

For S-parameter and RLCK models, RedHawk-SC supports automatic pre-simulation time determination for package settling.

Connecting S-Parameter Models in REDHAWK_PKG Subcircuit

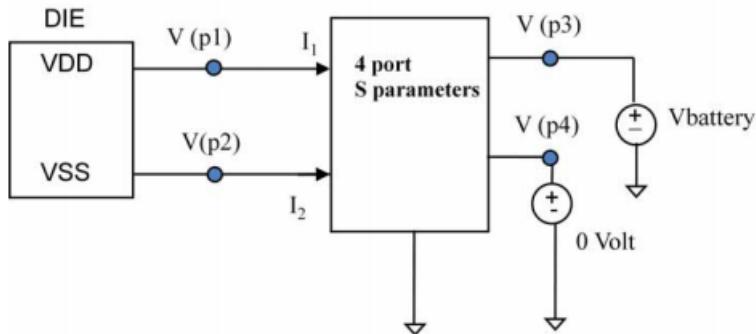
There are two modes of connection, common reference (ports referred to global ground), and differential (floating) reference. In practice, differential connection is used more often, as most electromagnetic solvers produce package or PCB models for ground nets with differential port definitions (between the node pairs, and not referred to global ground).

S-Parameter Models With Common Reference Node

The most intuitive way of connecting the S-parameter models is with all ports referred to global ground (SPICE node 0). In this case, each port is defined between a node N_k and the global ground. The use model for the S-parameters is essentially the same as for RLCK models. Specifically, the absolute node voltages (those with respect to the global ground) make sense, and there is no condition imposed on port currents. The sum of the port currents does not have to be zero, as there is also the reference current, I_{ref} .

The following figure shows the use of S-parameter models with all ports defined between a node and global ground (SPICE node 0). In this case, the simplest possible package is a 4-port S-parameter model. From a user standpoint, there is no difference between using an RLCK package model and using an S-parameter model.

Figure 72. Basic Global Ground Connection S-Parameter Model



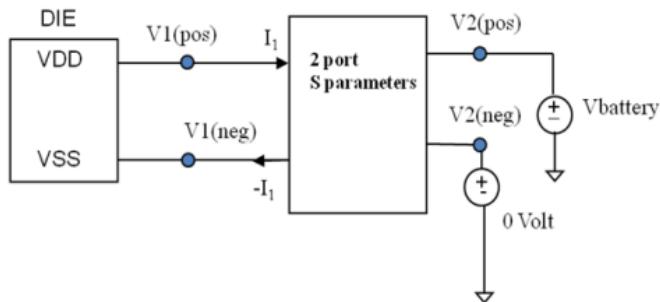
However, there are few disadvantages in using S-parameters with ports referred to global ground:

- Many industrial EM solvers cannot extract S-parameters with ports referred to the global ground, as sometimes the definition of global ground or a common reference node is not clear.
- The number of ports is twice the number for the differential connection.

Differential Connection of S-Parameter Model

For differential connection of S-parameters, each port voltage is defined between a pair of non-ground nodes, $k(pos)$ and $k(neg)$, where k is the node number. Thus, an n-port S-parameter model is connected between $2N$ nodes. The following example with 2 ports and 4 nodes shows differential voltages. All VDD pads are grouped together, and all VSS pads are grouped together.

Only the differential voltages are defined as ($V1 = V1(pos) - V1(neg)$) and ($V2 = V2(pos) - V2(neg)$). S-parameters also force a zero sum of currents $I(VDD) = -I(VSS)$ by construction. In this case, S-parameter model does not provide any DC path to global ground that is generally available through the package and supply voltage sources.

Figure 73. Basic Differential Connection S-Parameter Model

12.3.1. Specifying S-Parameter Models in RedHawk-SC

RedHawk-SC treats an n-port S-parameter model as a 2N terminal device. An n-port S-parameter network is represented in the following format:

```
Nxxxxx S(N) n1p n1n n2p n2n ..... nNp nNn <modelname>
```

where,

- Nxxxxx: network name
- n1p, n1n: nodes for port 1
- n2p, n2n: nodes for port 2
- N: number of ports in the network
- <modelname>: model statement name that contains the S-parameter data

The model statement for an S-parameter model is defined as:

```
.model <modelname> nport file = "<filename>" np = <N_value>
```

where,

- nport file, np: keywords
- <filename>: name of the S-parameter data file
- <N_value>: specifies the number of ports

Example of two-port S-parameter model:

```
.subckt REDHAWK_PKG pVdd pVss
Npkg S(2) pVdd pVss bgaVdd bgaVss PKG_MODEL
.model PKG_MODEL nport file = "pkg.s2p" np = 2
Vvdd bgaVdd 0 1.2
Vvss bgaVss 0 0.0
.ends
```

Note: The same REDHAWK_PKG supports multiple S-parameter models.

Recommendations and limitations in using S-parameter models:

- The number of ports should not exceed 100. Limit to 50-60 ports for better runtime and reliability.
- For static analysis, capture the DC or low-frequency S-parameters.

- Only one reference impedance for all ports is supported (the vector of reference impedances is not supported).
- Touchstone 2.0 format is not yet supported.
- The touchstone file should contain the S-matrix, not Y or Z matrices. The second keyword in the header of touchstone files should be 'S' for S-matrix: # HZ S RI R 2.000

12.3.2. Including S-Parameters in RedHawk-SC Analysis

Perform the following steps if you have S-parameter model available as part of the package or PCB board modeling:

1. Create the `REDHAWK_PKG` subcircuit wrapper by instantiating S-parameter model in it.

```
.subckt REDHAWK_PKG pVdd pVss
Npkg S(2) pVdd pVss bgaVdd bgaVss PKG_MODEL
.model PKG_MODEL nport file = "pkg.s2p" np = 2
Vvdd bgaVdd 0 1.2
Vvss bgaVss 0 0.0
.ends
```

2. Create the `SimulationView` by pointing `cpa_library_path` from RedHawk installation area.

```
>>> pkg_sv = create_package_simulation_view(extract_view,
package=package.parse_ploc_spice_func(<top level package spice netlist>,
dv=modified_design_view, cpa_lib_path=<path to cpa library from redhawk
installation area>)
```

Example:

```
>>> pkg_sv = create_package_simulation_view(extract_view,
package=package.parse_ploc_spice_func('top.sp', dv=modified_design_view,
cpa_lib_path='<RedHawk installation path>/lib/cpa'))
'top.sp' is referring to the file name which include above wrapper subckt
mentioned
```

12.4. Important Functions of Package Module

This section describes some of the important functions of the `package` module that are useful for hooking up the package netlist to die for simulations.

`parse_ploc_spice_func`

This function is mainly used for hooking up the top-level package netlist to the die netlist.

Usage:

```
parse_ploc_spice_func(file_name, dv, net_based=False, pad_file_name=None,
cpa_lib_path=None)
```

Argument	Description
<code>file_name</code>	(Required) A top level package spice netlist with subcircuit named <code>REDHAWK_PKG</code> .

Argument	Description
dv	(Required) DesignView or ModifiedDesignView.
net_based	Net names are used for spice port mapping and this makes lumped connection from package to the die, possible values are 'True' and 'False', Default: 'False'.
pad_file_name	<p>RedHawk pad file name for bump name to spice port mapping. For single chips, use the following pad file format: (* is comment)</p> <pre>* Bump-name Dont-care Spice-port-name VDD_p1 1 VDD_spc1 VDD_p2 2 VDD_spc1 VDD_p3 3 VDD_p3 VSS_p1 4 VSS_spc1</pre> <p>Note: For multiple chips connected to the same package, use the following pad file format:</p> <pre>* Bump-name Chip-name Spice-port-name VDD_p1 chip_top VDD_spc1 VDD_p2 chip_top VDD_spc1 VDD_p3 chip_top VDD_spc1 VSS_p1 chip_top VSS_spc1</pre>
cpa_lib_path	CPA library path from RedHawk installation area.

Examples:

- Connecting RLCK package netlist where the die port to package mapping is used from the ploc file.

```
>>> pkg_sv = create_package_simulation_view(extract_view,
package=package.parse_ploc_spice_func
('top.sp', dv=modified_design_view))
```

- Connecting S-parameter based package netlist.

```
>>> pkg_sv = create_package_simulation_view(extract_view,
package=package.parse_ploc_spice_func
('top.sp', dv=modified_design_view, cpa_lib_path='<RedHawk installation
path>/lib/cpa'))
```

- Using `pad_file_name` for grouping and mapping die bumps to package ports.

Content of 'bump2spiceMapping.txt':

```
* Bump-name Dont-care Spice-port-name
VDD_p1 1 VDD_spc1
VDD_p2 2 VDD_spc1
```

```
VDD_p3 3 VDD_p3
VSS_p1 4 VSS_spc1
VSS_p2 5 VSS_spc1
```

- Making lumped connection to the die from distributed package.

```
>>> pkg_sv = create_package_simulation_view(extract_view, package=
    package.parse_ploc_spice_func('top.sp', dv=modified_design_view,
    net_based=True))
```

Note:

- `pad_file_name` argument has highest priority compared to the `net_based` argument. Both the arguments have higher priority than the mapping coming from the ploc file of modified design view. And it is required to have all the die bumps listed in file to `pad_file_name`.
- `Package.parse_ploc_spice_func` accepts the package wrapper netlist with subcircuit name `REDHAWK_PKG`. `Wrapper netlist` can be created with `package.create_package_wrapper` function.

create_package_wrapper

This function is mainly used to create the package wrapper netlist for n-port and lumped configurations, which can be directly passed to the `package=package.parse_ploc_spice_func()` argument of the `SimulationView`.

Usage:

```
>>> create_package_wrapper(bump_file_name, net_data=None,
    chip_package_pg_pairs=None, spice_package=None, mapping_type='nport',
    pkg_file_name='package_wrapper.pkg', find_leff=False, spice_path='nspice')
```

Argument	Description
<code>bump_file_name</code>	(Required) Name of input bump file. Format: <name> <xloc in u> <yloc in u> <LEF layer> <PG net> <Spice port Name>
<code>net_data</code>	List of PG nets and voltage information. Default=None Format: [(Net(<pg_net1>),<voltage_value1>, R, C, L), (Net(<pg_net2>),<voltage_value2>, R, C, L),...]
<code>chip_package_pg_pairs</code>	List of net properties and mapping information. Default=None Format: [(Net(<pg_net1>),die_spice_port1,'package_spice_port1',voltage1), (Net(<pg_net2>),die_spice_port2,'package_spice_port2',voltage2),...]
<code>spice_package</code>	Name of the input SPICE package model file. Only RLCK package netlist is supported.

Argument	Description
mapping_type	Type of mapping used. <ul style="list-style-type: none"> • nport: If the input bump file has mapping information (default). • singleport: Maps all the plocs of domain to single package node, when bump file has no mapping information. • lumped: To provide lumped package model.
pkg_file_name	Name of the output package wrapper netlist file. Default = 'bumps_in_gps_format.pkg'.
find_leff	Enables the effective inductance calculation when set to 'True'. Default=False.
spice_path	Provides the spice binary path. Default='nspice'.

The following table lists the detailed description of `mapping_type` argument.

Argument	Description
nport	(Required) <code>bump_file_name</code> : input ploc file of the form: <name> <xloc in u> <yloc in u> <LEF layer> <PG net> <Spice port Name> <code>spice_package</code> : input spice netlist file <code>net_data</code> : list of PG net and voltage info Example: <code>net_data = [(Net('VDD'),1.0), (Net('VSS'),0.0)]</code> (Optional) <code>chip_package_pg_pairs</code> : list of nets with mapping information.
singleport	(Required) <code>bump_file_name</code> : input ploc file containing lines of the form: <name> <xloc in u> <yloc in u> <LEF layer> <PG net> <code>spice_package</code> : input spice netlist file <code>chip_package_pg_pairs</code> : list of net properties and mapping info. Example: <code>= [(Net('VDD'), 'VDD-MP_1_1', 'VDD-MP_1_2', 1.0), (Net('VSS'), 'VSS-MP_2_1', 'VSS-MP_2_2', 0.0)]</code> (Optional) <code>net_data</code> : list of PG net and voltage information.

Argument	Description
lumped	(Required) <i>bump_file_name</i> : input ploc file containing lines of the form: <name> <xloc in u> <yloc in u> <LEF layer> <PG net> <i>net_data</i> = list of the form: [(Net Name,Voltage,R,C,L)] Example: <i>net_data</i> = [(Net('VDD'),1.0,1e-3,1e-9,1e-9), (Net('VSS'),0.0,1e-3,2e-9,3e-9)]

The following table lists the optional settings for all the modes.

Argument	Description
<i>pkg_file_name</i>	Output file for the generated gps package file; default = 'bumps_in_gps_format.pkg'
<i>find_leff</i>	To find the effective inductance of the PKG spice netlist. Default = False.
<i>spice_path</i>	Spice-binary path for computing the effective inductance of the package netlist provided.

Examples:

- Creating the lumped package with user RLC inputs. This generates *lumped_pkg.sp* file, which is passed to *package* argument of *SimulationView*.

```
>>> lumped_package_values = [(Net('VDD'),1.07,1e-3,1e-9,5e-12),
(Net('VSS'),0.0,1e-3,1e-9,8e-12)]
>>> package.create_package_wrapper(bump_file_name='top.ploc',
net_data=lumped_package_values,mapping_type='lumped',pkg_file_name='lumped_pkg.sp')
```

- Creating the nport (per bump or grouping information will be taken from the input bump file) package with input package model. This generates *nport_pkg.sp* file, which is passed to *package* argument of *SimulationView*.

```
>>> net_info = [(Net('VDD'),1.0), (Net('VSS'),0.0)]
#'package.sp' file should have the cpp header.
>>> package.create_package_wrapper(bump_file_name='top.ploc',
mapping_type='nport',
net_data=net_info, spice_package='package.sp', pkg_file_name='nport_pkg.sp')
```

- When input bump file (ploc) does not have the spice mapping information, then *singleport* option can be used to create the lumped package spice wrapper file. This generates *singleport_pkg.sp* file, which is passed to *package* argument of *SimulationView*.

```
>>> pgpair_info = [(Net('VDD'),'VDD-MP_1_1', 'VDD-MP_1_2', 1.0), (Net('VSS'),
'VSS-MP_2_1', 'VSS-MP_2_2', 0.0)]
>>> package.create_package_wrapper(bump_file_name='top.ploc',
mapping_type='singleport', chip_package_pg_pairs=pgpair_info,
spice_package='package.sp', pkg_file_name='singleport_pkg.sp')
```

create_per_bump_spice_func

This function can be used directly in SimulationView to stitch package spice netlist to die when there is no package netlist available, but you have per bump R,L,C values.

Usage:

```
>>> create_per_bump_spice_func(dv, r=0, l=0, c=0, nets=None,
v_src_at_subnets=False, short_res_for_subnets=-1, v_sources=None)
```

Argument	Description
dv	(Required) Input DesignView
r	Per bump resistance value in Ohms
l	Per bump inductance in Henry
c	Per bump capacitance in Farad
nets	Create the package model only for the specified list of nets.
v_src_st_subnets	Create per bump package model for subnet ¹ bumps.
short_res_for_subnets	Short the bumps on a subnet ¹ with this resistance. Default: 'no short'.
v_sources	Per bump voltage waveform. Format: [{'bump':bump_name, 'voltage':Waveform()}, ...]

Note: ¹Subnet refers to switchable domain which is connected to internal domain of the switch.

Examples:

- Creating the per-bump package with user defined RLC and including the same for simulation.

```
>>> pkg_sv = create_package_simulation_view(extract_view,
package=package.create_per_bump_spice_func(dv=modified_design_view,
r=2,l=1e-11,c=1e-13))
```

- If RLC values are not defined, v_sources defined are directly connected to the die bumps. The values specified in the v_sources argument are relative to its domain ideal voltage. Providing PWL waveform to the package source with user defined RLC.

```
>>> v_sources = [ {'bump':'pack*', 'voltage':Waveform([(0,0),
(40e-12, 0.01), (80e-12, -0.01), (180e-12, 0.01), (200e-12, 0)])},
{'bump':'package_bump_gp_40', 'voltage': Waveform([(0,0.01),
(40e-12, -0.01), (100e-12, 0.01), (200e-12, -0.01), (300e-12, 0)])}]
>>> pkg_sv = create_package_simulation_view(extract_view,
package=package.create_per_bump_spice_func(dv=modified_design_view,
r=2,l=1e-11,c=1e-13,v_sources=v_sources))
```

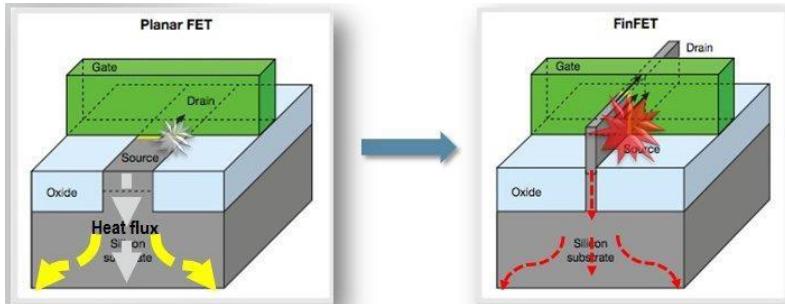
- Shorting the subnet bumps with resistance = 1e-4Ohms.

```
>>> pkg_sv = create_package_simulation_view(extract_view,
package=package.create_per_bump_spice_func
(dv=modified_design_view, r=2,l=1e-11,c=1e-13, short_res_for_subnet = 1e-4))
```


13: SelfHeat Analysis

The ever-increasing demand for speed, functionality, and scalability of an SoC have resulted in sizeable reduction of the interconnect metal pitch and increased number of metallization levels.

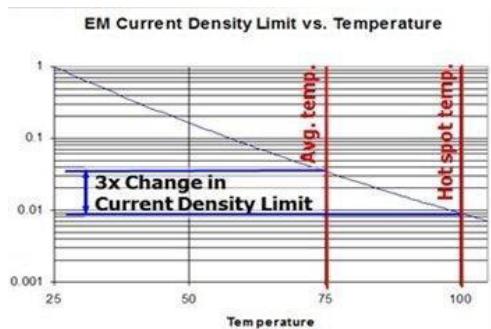
With the transition from planar MOS to FinFET transistor design and aggressive interconnect scaling, current density has increased by more than 25%. Narrow 3D fin structure and low thermal conductivity due to silicon dioxide dominated substrate causes heat traps as shown in following figure:



Thermal effects impact interconnect design and reliability due to following reasons:

- The RMS value of the current density generates heat flux. As the temperature increases, resistance across the interconnect also increases, causing increase in temperature across the edge (wire/via).
- The reliability of an interconnect, which is limited by the electromigration (EM) effect, is also depended on the metal temperature. Change in temperature due to selfheating, limits the maximum allowable RMS current density in interconnects.
- Thermally-induced stress over the life of a device, gradually deteriorates the system reliability.
- Thermally-induced open or short circuit metal failure, under short-duration high peak currents, can introduce latent electromigration damage that has reliability implications.
- Increase in temperature causes power leakage across a transistor junction. This increases overall device temperature, resulting in increased cooling costs.
- Higher drive strength of devices proportionally increases the electromigration impact as a function of temperature.

Without proper thermal modeling, the increase in temperature can degrade lifetime of device and interconnect significantly due to electromigration.



As shown in the above figure, the electromigration limit is also a complex function of temperature in addition to the dependence on width and thickness. Increase in selfheat can cause power electromigration and signal electromigration issues.

To avoid overheating of a FinFET-based design, you must consider all the thermal effects while analyzing digital or analog designs.

The following topics describe the selfheat sources on interconnects and how to perform thermal analysis using the RedHawk-SC tool:

- [Sources of SelfHeat](#) on page 392
- [SelfHeat Flows](#) on page 393
- [SelfHeat Analysis Overview](#) on page 395
- [SelfHeat Reports](#) on page 403
- [Reviewing Results in GUI](#) on page 407

13.1. Sources of SelfHeat

There are three different sources of heat that impact the final temperature on edges (wire/via):

- **Joule Heating:**

When current flows through a wire, electric energy is converted to thermal energy through resistive losses in the material. This results in heat dissipation around the edges, known as joule heating (`deltaT_joule_heating`). `deltaT_joule_heating` is a function of `deltaT_rms` and `emViolation`.

- **Instance/Device Coupling**

Heat dissipated through transistor junctions gets coupled with neighbouring instances. This causes increase in temperature of edges, which results in instance/device coupling (`deltaT_instance_coupling`). `deltaT_joule_coupling` is a function of instance power derived from the `sh_file`.

- **Wire-to-wire Coupling**

Due to high joule heating in wires, heat produced across the wires gets coupled with neighbouring wires. This known source of rise is temperature causes wire-to-wire coupling (`deltaT_wire_to_wire_coupling`). `deltaT_wire_to_wire_coupling` is a function of `deltaT_instance_coupling` and physical parameters of the wires sourced from `trf_file`.

In general flow, a uniform `deltaT_joule_heating` is considered throughout the chip. Ambient temperature is specified through `temperature_environment`, which can also be obtained from the FIT (Failure In Time) file. Herein, the final temperature is the sum of ambient temperature and joule heating.

```
Final temperature = temperature_environment + deltaT_joule_heating
```

In a thermal aware flow, in addition to temperature change caused by `deltaT_joule_heating`, the temperature impacted by `deltaT_wire_to_wire_coupling`, and `deltaT_instance_coupling` thermal coupling are also considered to calculate the final temperature.

Effective edge temperature is calculated using the following formula:

```
Effective edge temperature = temperature_environment + Edge self_heating
```

Where,

```
Edge self_heating = deltaT_Joule_heating + deltaT_instance_coupling
                  + deltaT_wire_to_wire_coupling
```

With the results from thermal analysis, you can also perform a thermally-aware electromigration analysis using RedHawk-SC. The tool can combine local thermal effect and global thermal effect to perform a comprehensive thermal analysis of the design.

13.2. SelfHeat Flows

The existing pessimistic estimation of the interconnect temperature might lead to an overly conservative approach. Thermal Analysis using RedHawk-SC is designed to be more realistic and inclusive. This requires the use of foundry-supported, measurement-based method to handle 3D FinFET structure for better accuracy.

RedHawk-SC has different selfheat flows that provisions specific requirements as per industry standards. You can use the `thermal_calculation_settings` command to enable any of the following pre-configured flows for thermal analysis:

- **TSMC Foundry flow ('thermal_flow':1)**

TSMC Foundry flow is the default flow enabled in Redhawk-SC. It uses thermal equations from TSMC. You can enable it using the `'thermal_flow':1` settings. It uses the Thermal Resistance File (`trf_file`) and SelfHeat (`sh_file`) provided by the foundry as inputs. The TSMC flow is an equation-based flow that calculates the change in temperature based on following factors:

- `deltaT_instance` : Device/Instance thermal coupling
- `deltaT_instance_coupling` : Instance-to-wire thermal coupling
- `deltaT_joule_heating` : Joule heating on edge (wire/via) computed from TSMC RMS equations.

Use the following settings to configure TSMC equation-based flow for thermal analysis:

```
settings = {'temperature_environment':<temperature>,
'calculation':{'thermal_flow':1},
'edge_heatmap_types':[{'temperature','self_heating',
'joule_heating','instance_coupling']}}

create_thermal_view(pg_em_view=emv_pg, signal_em_view=emv_sig,
sh_file=sh_file, tag='thmv', trf_file=trf_file, options=options,
settings = settings)
```

- **TSMC-Ansys Hybrid Flow ('thermal_flow':2)**

TSMC-Ansys Hybrid Flow is an enhanced version of the TSMC flow, which uses the Ansys's Fast Thermal Coupling (FTC) output, and considers an additional source of selfheat for thermal analysis. Use the `'thermal_flow':2` option to enable the TSMC and Ansys Flow. It also uses the Thermal Resistance File (`trf_file`) and SelfHeat (`sh_file`) file provided by the foundry as input. The TSMC and Ansys flow is an equation-based flow that calculates the change in temperature based on following factors:

- `deltaT_instance` : Device/Instance thermal coupling
- `deltaT_instance_coupling` : Instance-to-wire thermal coupling
- `deltaT_wire_to_wire_coupling` : Wire-to-wire thermal coupling computed using Ansys's FTC analysis.
- `deltaT_joule_heating` : Joule heating on edge (wire/via) computed from self-heat equations that estimate delta-T on each wire segment.

Use the following settings to configure TSMC-Ansys equation-based flow for thermal analysis:

```
settings = {'temperature_environment':<temperature>,
'calculation':{'thermal_flow':2},
```

```
'edge_heatmap_types':['temperature','self_heating',
'joule_heating','instance_coupling', 'wire_to_wire_coupling']}}

create_thermal_view(pg_em_view=env_pg, signal_em_view=env_sig,
sh_file=sh_file, trf_file=trf_file, options=options, settings = settings,
tag='thmv')
```

- **Ansys Flow [foundry-independent flow] ('thermal_flow':3)**

If there are no foundry inputs available for thermal analysis, you can use the Ansys Flow in RedHawk-SC by enabling the 'thermal_flow':3 option. Ansys Flow uses instance level power information generated by RedHawk-SC as the heat source input for computing `deltaT_instance`. The Ansys flow is a foundry-independent flow that calculates the change in temperature based on following factors:

- `deltaT_instance` : Device/Instance thermal coupling calculated based on Ansys's FTC analysis.
- `deltaT_instance_coupling` : Instance-to-Wire thermal coupling calculated based on Ansys's FTC analysis.
- `deltaT_wire_to_wire_coupling` : Wire-to-wire thermal coupling computed using Ansys's FTC analysis.
- `deltaT_joule_heating` : Joule heating on each wire segment is calculated based on FTC analysis.

Use the following settings to configure the foundry-independend Ansys flow for thermal analysis:

```
settings = {'temperature_environment':<temperature>,
'calculation':{'thermal_flow':3},
'edge_heatmap_types':['temperature','self_heating',
'joule_heating', 'instance_coupling','wire_to_wire_coupling']}

create_thermal_view(pg_em_view=env_pg, signal_em_view=env_sig, sh_file=None,
trf_file=None, options=options, settings = settings, tag='thmv')
```

- **General Foundry Flow ('thermal_flow':4)**

You can use the 'thermal_flow':4 option to enable thermal equations from other foundries. The methodology might differ from foundry to foundry, but in general the flow uses the Thermal Resistance File (`trf_file`)and SelfHeat (`sh_file`) provided by the foundry as input. The flow is an equation-based flow that calculates the change in temperature based on following factors:

- `deltaT_instance` : Device/Instance thermal coupling
- `deltaT_instance_coupling` : Instance-to-wire thermal coupling
- `deltaT_joule_heating` : Joule heating on edge (wire/via) computed from RMS equations.

Use the following settings to configure the General Foundry Flow for thermal analysis:

```
settings = {'temperature_environment':<temperature>,
'calculation':{'thermal_flow':4},
'edge_heatmap_types':['temperature','self_heating',
'joule_heating','instance_coupling']}

create_thermal_view(pg_em_view=env_pg, signal_em_view=env_sig,
sh_file=sh_file, trf_file=trf_file, temperature = <temperature>,
options=options, settings = settings, tag='thmv')
```

- **Joule Heat Only flow: ('joule_heat_only':1)**

Thermal analysis using Joule Heat Only flow considers a uniform deltaT_rms throughout the chip. deltaT_joule_heating on edge is computed based on RMS equations from TSMC.

Use the following settings to configure Joule Heat Only flow for thermal analysis:

- For joule analysis on signal nets, use

```
settings = {
    'temperature_environment':<temperature>,
    'calculation': {'joule_heat_only':1},
    'edge_heatmap_types': ['temperature', 'joule_heating']}}

create_thermal_view(pg_em_view=None, signal_em_view=env_sig, sh_file=sh_file,
trf_file=None, options=options, settings = settings, tag='thmv')
```

- For joule analysis on power nets, use

```
settings = {'temperature_environment':<temperature>,
            'calculation': {'joule_heat_only':1},
            'edge_heatmap_types': ['temperature', 'joule_heating']}}

create_thermal_view(pg_em_view=env_pg, signal_em_view=None, sh_file=sh_file,
trf_file=None, options=options, settings = settings,
edge_heatmap_types=['temperature', 'joule_heating'], tag='thmv')
```

- For joule analysis on both power and signal nets, use:

```
settings = {
    'temperature_environment':<temperature>,
    'calculation': {'joule_heat_only':1},
    'edge_heatmap_types': ['temperature', 'joule_heating']}}

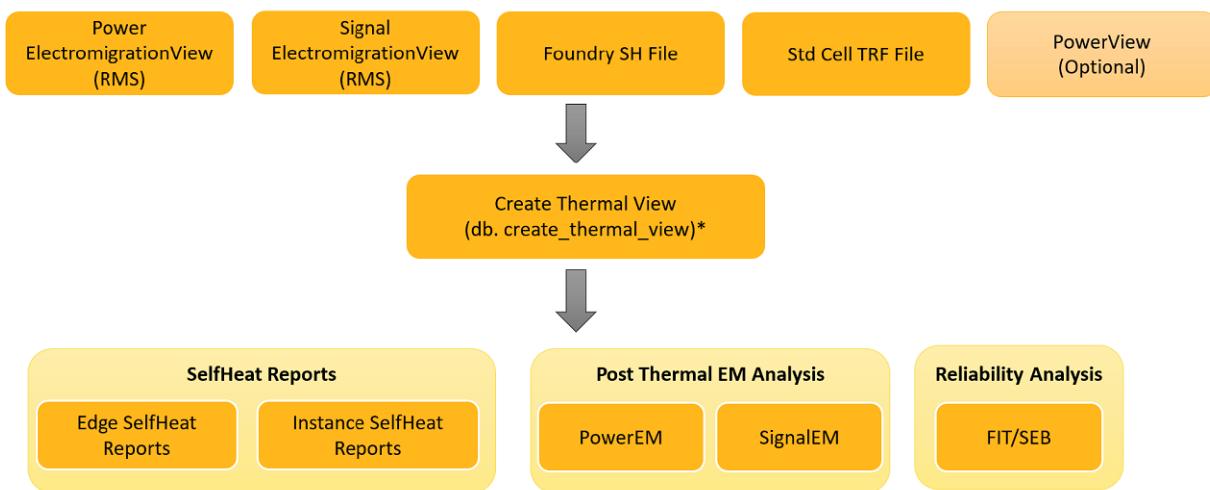
create_thermal_view(pg_em_view=env_pg, signal_em_view=env_sig,
sh_file=sh_file,
trf_file=None, options=options, settings = settings, tag='thmv')
```

The next topic explains the [SelfHeat Analysis Overview](#) on page 395 flow in detail. It helps in performing a thermal aware electromigration analysis to reduce the number of false electromigration violations, caused by pessimistic assumptions of global temperature.

13.3. SelfHeat Analysis Overview

Before proceeding with the thermal analysis, you need to derive the RMS currents flowing through each wire.

The following figure shows an overview of the SelfHeat analysis flow:



As shown in the diagram, the RMS data is used as input along with foundry-provided SelfHeat (SH) and thermal resistance file (TRF). RedHawk-SC performs different thermal flows by selectively using the input data files.

The following topics describe how to perform SelfHeat analysis using the RedHawk-SC tool:

- [Input Data Preparation](#) on page 396
- [Setting Up the Environment](#) on page 399
- [Creating Thermal View](#) on page 400
- [Post Thermal Electromigration Analysis](#) on page 401

13.3.1. Input Data Preparation

Apart from the ambient temperature, the key data requirements for SelfHeat Analysis needs to be obtained or prepared.

Layout and Timing Information

Data containing physical layout information can be obtained from input files such as LEF and DEF. Current waveform and switching power calculations can be derived from SPEF files. STA files provide net parasitics, frequency, and timing window information.

Refer [Setting Up the Environment](#) on page 399 for setup procedures.

ElectromigrationViews

To derive the RMS currents flowing through each wire, you need to create the Power ElectromigrationView and Signal ElectromigrationView.

Use the `db.create_electromigration_view` command and specify the `mode` argument with `RMS` option.

Arguments for Power ElectromigrationView (`pg_em_view`) and Signal ElectromigrationView (`pg_em_view`) are as follows:

```
signal_em_view = (type=ElectromigrationView, default_value=None)
```

```
pg_em_view = (type=ElectromigrationView, default_value=None)
```

For more details, refer [Creating Electromigration View](#) on page 272

The `db.create_thermal_view` command uses the RMS data obtained from ElectromigrationViews to compute Joule Heating.

Foundry Inputs

To compute device-instance coupling, or wire-to-wire coupling, you might need additional input data other than RMS data. Foundry provides technology files with equations for Joule Heating, and FEOL-BEOL coupling coefficients through foundry technology files.

Following are the foundry inputs required for performing selfheat analysis in RedHawk-SC.

- Thermal Resistance File (`trf_file`):

Self-heating aware electromigration flow requires cell thermal resistance data for calculating delta temperature for each standardcell. The `trf_file` contains data for all the standard cells representing various heat loss paths for heat conduction along each segment of the fins, channels, and metal wires. Thermal resistance in FinFET design is a function of core resistance, fin number, finger number and power dissipated through that fin. Arguments for the TRF file are as follows:

```
trf_file = (type=str, default_value=None)
```

Following is a snippet of TRF file:

```
AN2D1BWP16P90ULVT 2.395386e+05
AN2D2BWP16P90ULVT 1.904217e+05
AN2D4BWP16P90ULVT 1.068306e+05
```

- SelfHeat Technology File (`sh_file`)

The `sh_file` contains data for FEOL (Front-End-Of-Line) and BEOL (Back-End-Of-Line) self-heat calculation. BEOL wire self heating is a function of width, Irms and metal layers, considering hierarchical heat dissipation through the transistor channel and contact. FEOL-BEOL coupling equation is a function of metal layer information and coupling coefficients.

In order to extract thermal parameters, Ansys 3D finite element method (FEM) simulations are performed by solving Fourier equation for heat transport. Simulated device parameters are obtained from the Predictive Technology Model (PTM) high-performance FinFET model.

Alpha and Beta coefficients derived from analysis are used for SelfHeat calculation of vertical thermal coupling from device-instance to edge.

Following is a snippet of a SH file:

```
SH_C 2.0 M7 1. 0.050 0.050 2. 0.030 0.06
SH_C 2.0 M6 1. 0.050 0.050 2. 0.030 0.06
SH_C 2.0 M5 1. 0.050 0.050 2. 0.030 0.06
SH_C 2.0 M4 2. 0.050 0.050 2. 0.030 0.06
SH_C 2.0 M3 2. 0.050 0.050 2. 0.030 0.06
SH_C 2.0 M2 3. 0.050 0.050 2. 0.030 0.06
SH_C 2.0 M1 6. 0.050 0.050 2. 0.030 0.06
```

```
Aplha 0.9 0.5 0.4 0.3 0.2 0.2 0.1 0.1 0.1 0.1 0.1 0.1 0.1
Beta 0.1 0.2 0.3
```

- Failure in Time (FIT) Tech File (`fit_tech_file`)

FIT technology file is used in foundry-dependent thermal flows and post thermal analysis.

It contains details of `MTF_HOUR`, `SIGMA`, `Ea` and `EXP_N` values.

Following is an example for SEB iRCX `fit_tech_file` format:

```
HEADER {
FILE :=SEB
DATE := 2020-08-06
}

* FIT_TABLE 22 5 (LAYER:V, MTF_HOUR:V, SIGMA:V, Ea:V, EXP_N:V)
FIELD MTF_HOUR SIGMA Ea EXP_N
LAYER
metal11 4459740 0.29 2.15 0.1
metal10 4459740 0.29 2.15 0.1
meta19 4459740 0.29 2.15 0.1
meta18 4459740 0.29 2.15 0.1
meta17 4459740 0.29 2.15 0.1
meta16 4459740 0.29 2.15 0.1
meta15 4459740 0.29 2.15 0.1
meta14 4459740 0.29 2.15 0.1
meta13 8268150 0.4 2.15 0.1
meta12 8268150 0.4 2.15 0.1
meta11 8268150 0.4 2.15 0.1
via11 4459740 0.29 2.15 0.1
via10 4459740 0.29 2.15 0.1
via9 4459740 0.29 2.15 0.1
via8 4459740 0.29 2.15 0.1
via7 4459740 0.29 2.15 0.1
via6 4459740 0.29 2.15 0.1
via5 4459740 0.29 2.15 0.1
via4 4459740 0.29 2.15 0.1
via3 4459740 0.29 2.15 0.1
via2 8268150 0.4 2.15 0.1
vial 8268150 0.4 2.15 0.1
```

Ansys FTC

If there is no foundry input available, you can use Ansys's Fast Thermal Coupling (FTC) analysis to compute `deltaT_instance`. FTC analysis is further derived from Ansys Finite Element Method (FEM) pre-characterization of change in temperature and self-heat induced thermal couplings.

Macros

MacroView contains tile based chip thermal profile inside CMM model, along with physical representation of the power grid. RedHawk-SC uses the device temperature from the thermal profile for top level thermal analysis.

Example of thermal settings to use macros as input:

```
thermal_profile_map_list='./cmm_chip_thermal_profile_map'
thermal_calculation_settings = {'thermal_flow':1,
```

```
'cmm': '/wrk/project/cmm.gsc',
'cmm_thermal_profile_map':
thermal_profile_map_list,
}
```

Sample snippet from a `cmm_chip_thermal_profile_map` file:

```
#<master_name> <state> <chip thermal profile>
ipm_arxs CYCLE1 /wrk/project/RawModel/chip_thermal_profile.bin
```

Sample snippet from a `cmm.gsc` file:

```
#<inst_name> <state>
Inst_Ram_0 CYCLE1
```

PowerView (Optional)

By default, instance power is derived from values in ScenarioView. However, you can also provide user-defined instance power file (.ipf) using PowerView as an input.

Related information

[Setting Up the Environment](#) on page 399

[MacroView](#) on page 52

[ElectromigrationView](#) on page 47

[ThermalView](#) on page 48

13.3.2. Setting Up the Environment

Load worker environment, variables and arguments. Create base views by specifying input data.

1. Invoke the scheduler GUI, refer [Invoking the Scheduler GUI](#) on page 441.
2. Create and launch workers, refer [Creating and Launching Workers](#) on page 431
3. Specify the database location to automatically load the base views from the `db` file.

```
db = open_db('..../db')
```

4. Launch RedHawk-SC, refer [Launching RedHawk-SC](#) on page 25
5. (Optional) In case of incremental runs, use the `populate_view_tags` command to load the view tags and jump-start the run.

```
populate_view_tags
```

6. Set the variable to the central design data path.

```
design_data_path = '..../design_data/
```

7. Specify the input files and settings. You can specify the input files as Python lists or dictionaries.

```
#Specify design files
def_files = [
```

```

design_data_path + '/defs/Galaxy.def'
...
]
lef_files = [
design_data_path + '/lefs/switch_cell.lef',
...
]
trf_file = design_data_path +'/ trf_file
sh_file = design_data_path + '/ sh_file
fit_tech_file = design_data_path +'/ fit_tech_file

```

Note: By default, thermal analysis uses one core per worker. This might result in high runtime for selfheat flows, where computations for each wire-to-wire coupling is required. To improve runtime, set the maximum number of threads per worker using `create_*_launcher` command. For example, `create_grid_launcher('qsub', qsub_command, max_num_threads_per_worker=<max>)`

Related information

[Launching Workers](#) on page 22

[Launching the Scheduler GUI](#) on page 28

[Launching the RedHawk-SC Console](#) on page 27

[Preparing Input Data](#) on page 29

13.3.3. Creating Thermal View

The `db.create_thermal_view` command performs thermal analysis. The command has following arguments:

```

settings = {'temperature_environment':<temperature>,
'calculation':{<thermal_flow>},
'edge_heatmap_types':[<temperature>', 'self_heating', 'joule_heating', 'instance_coupling',
                     'wire_to_wire_coupling']}
create_thermal_view(pg_em_view=emv_pg, signal_em_view=emv_sig, power_view=None,
                   sh_file=sh_file, trf_file=trf_file, settings=settings, options=options,
                   tag='thmv')

```

By default, RedHawk-SC performs thermal analysis using the '`thermal_flow`' :1 settings. Refer [SelfHeat Flows](#) on page 393

Note: Use the following help function for latest/updated syntax and API's available in Thermal View:

```
help([ThermalView, SeaScapeDB.create_thermal_view], detailed=True)
```

Setting Up the Arguments

Specify modules to import arguments, specify thermal specific settings and input file arguments for base view creations. For modularity, these arguments are defined in a different file (`args.py`). Following snippet shows thermal analysis arguments for '`thermal_flow':1` and '`thermal_flow':2`

```
thmv_flow1_args = dict(
    tag='thmv_flow1',
    temperature_environment=125,
    instance_heatmap_types=['temperature', 'self_heating'],
    edge_heatmap_types= ['temperature', 'joule_heating',
        'instance_coupling', 'self_heating'],
    thermal_calculation_settings={'thermal_flow':1},
    options=options)

thmv_flow2_args = dict(
    tag='thmv_flow2',
    temperature_environment=125,
    instance_heatmap_types=['temperature', 'self_heating'],
    edge_heatmap_types=[ 'temperature', 'self_heating', 'joule_heating',
        'instance_coupling', 'wire_to_wire_coupling'],
    thermal_calculation_settings={'thermal_flow':2},
    options=options)
```

13.3.4. Post Thermal Electromigration Analysis

Post-thermal electromigration check is performed to back-annotate the results into the RedHawk-SC database, and recompute the electromigration limits. A thermally-aware electromigration analysis gives accurate RMS values and peak currents on both power/ground and signal nets of your design.

Specify a `thermal_view` flow option with the `create_electromigration_view` command as shown below:

```
em_settings = {'metal_line_number':9, context_factors=None, temperature_em=None,
               em_ruleset_names=None, em_mission_factor=None}
fit_settings = {calculate_fit=False, temperature_fit=None, fit_life_time=None,
                fit_mode='basic', fit_tech_file=None}
db.create_electromigration_view(analyses_view,
    thermal_view=<thermal_flow_option>,
    delta_t_rms=None, options=option, em_settings=em_settings,
    fit_settings=fit_settings)
```

For example:

- To perform power electromigration analysis with '`thermal_flow':1` settings, use the following command:

```
post_thermal_pem=db.create_electromigration_view(av_dynamic_npv,
    thermal_view=thmv_flow1, **post_thermal_pem_args)
```

Setting up the arguments for post thermal power electromigration analysis:

```
post_thermal_pem_args = dict(
    tag='post_thermal_pem',
    mode='dc',
    options=options,
```

```
calculate_fit=True,
temperature_fit=25.0,
fit_life_time=5e5,
fit_mode='reduction')
```

- To perform signal electromigration analysis with 'thermal_flow':1 settings, use the following command:

```
post_thermal_sem=db.create_electromigration_view(sigem_scv,
    thermal_view=thmv_flow1, **post_thermal_sem_args)
```

Setting up the arguments for post thermal Signal electromigration analysis:

```
post_thermal_sem_args = dict(
    tag='post_thermal_sem',
    mode='dc',
    options=options,
    calculate_fit=True,
    temperature_fit=25.0,
    fit_life_time=5e5,
    fit_mode='reduction')
```

Post RedHawk-SC Electrothermal SelfHeat Analysis

The vertical fins of a FinFET transistor operate at relatively high voltages, thereby generating high current densities and correspondingly high operating temperatures than other regions on the transistor. In a multi-fin structure, the central fins tend to be warmer because they are further away from the contacts. These differences in temperature does not account to the ambient temperature.

To consider more realistic temperatures, you can use the output from electrothermal analysis in RedHawk-SC Electrothermal.

- Generate a chip thermal profile using RedHawk-SC Electrothermal. You can import the `chip_thermal_profile.bin` file, and run the selfheat analysis using the following command:

```
thmv = db.create_thermal_view(emv, emv_sigem, sh_file, trf_file,
    temperature=110, options=options, tag='thmv'
    thermal_config={'thermal_profile':'chip_thermal_profile.bin'})
```

- You can also use the `AnalysisView` and `ExtractView` from the RedHawk-SC Electrothermal output, and run the selfheat analysis with `create_thermal_view_from_files` using the output files. Use the following command:

```
create_thermal_view_from_files(av=None, av_sigem=None, ev=None,
    thermal_profile='', tag='thmv', options=None)
ARGUMENTS
av : AnalysisView object (type=AnalysisView, default_value=None)
av_sigem : SignalNetCurrentView object(type=SignalNetCurrentView,
default_value=None)
ev : A full mode Extractionview or a list of two
Extractionviews of pg and signal (type=object, default_value=None)
thermal_profile : Thermal profile from CTA (type=object, default_value='')
tag : Name of the view (type=str, default_value='thmv')
options : View creation options, typically from
get_default_options() (type=object, default_value=None)
```

Importance of Post Thermal Electromigration Analysis

Electromigration lifetime is exponentially dependent on the inverse temperature. Hence, thermal analysis is an essential prerequisite to electromigration analysis to perform an accurate full-chip reliability analysis. Neglecting the temperature effect in electromigration failure analysis can substantially overestimate the metal lifetime, leading to unacceptable prediction error.

A well-analyzed temperature profile plays an important role in the application of thermal stress evaluation and package design at chip or PCB level. You can refactor the design rules for interconnects by maximum, average and DC current-density limits that depend on temperature, layer and quality goals. In order to prevent excessive self-heating in interconnects, you must consider the root-mean-square and peak currents while limiting the maximum permitted increase in temperature.

The number of electromigration violations might increase significantly in future technology nodes. It is highly important that today's design flows change from the traditional electromigration flows to thermally-aware electromigration analysis, enabling the expected current density rise and ensuring reliable circuits.

Related information

[Creating Electromigration View](#) on page 272

[Creating Thermal View](#) on page 400

13.4. SelfHeat Reports

Following SelfHeat related heatmaps are available in RedHawk-SC:

- Temperature
- Self_heating
- Instance_coupling
- Wire_to_wire_coupling
- Joule_heating

By default, RedHawk-SC reports the `temperature` heatmap only. To override the default behaviour, you can enable below heatmap types with the `create_thermal_view` command:

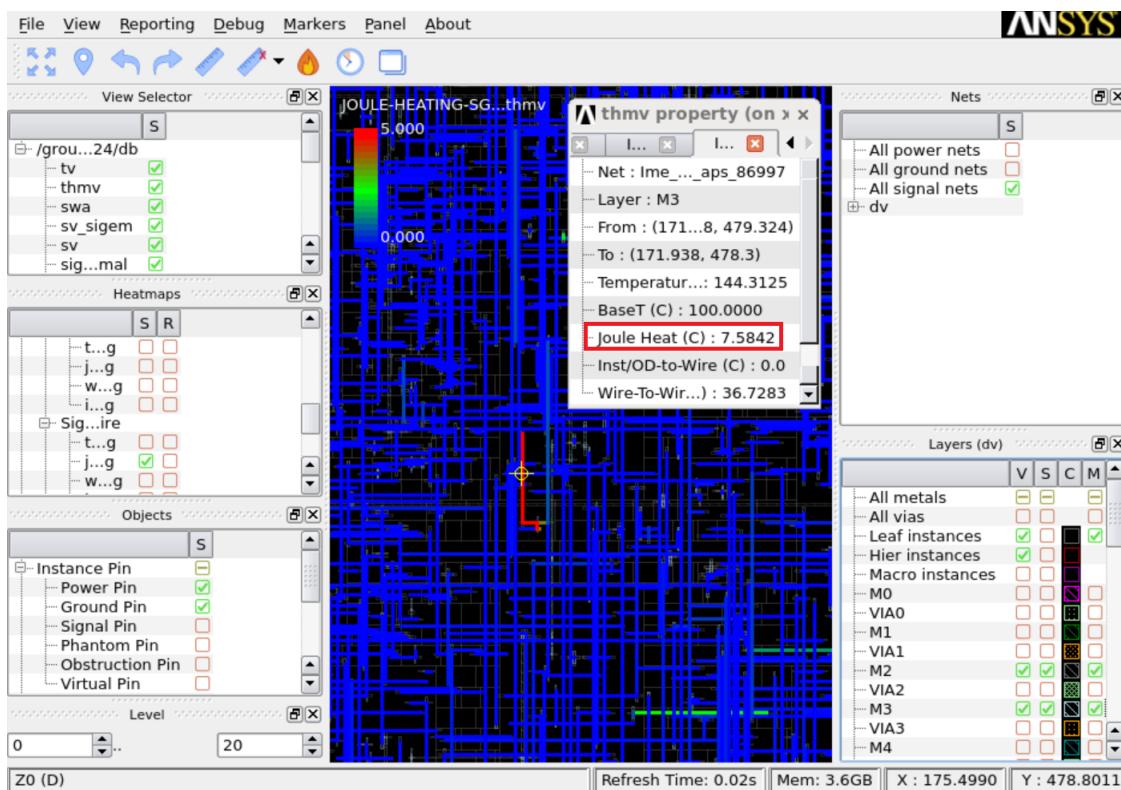
- To generate instance temperature report:

```
instance_heatmap_types=['temperature', 'self_heating']
```

- To generate edge temperature report:

```
edge_heatmap_types=['temperature', 'self_heating',
                    'joule_heating', 'instance_coupling',
                    'wire_to_wire_coupling']
```

Following figure shows Temperature Heatmap inside RedHawk-SC GUI.



Several APIs are available to generate edge temperature and instance temperature reports.

The following sections describe the different methods to report SelfHeat analysis results:

- [Reporting Instance Temperature](#) on page 404
- [Reporting Edge Temperature](#) on page 405

13.4.1. Reporting Instance Temperature

Instance temperature report includes path to report file and details of temperature and self_heating heatmap. Use the following API command to report instance temperatures:

```
ThermalView.report_instance_temperatures(report_file=None, columns=['instance',
    'loc_x', 'loc_y', 'temperature_ambient', 'temperature', 'self_heating'],
    sort=True, sort_order='descending', sort_columns=['self_heating'],
    formats=None, header=None, footer=None, max_lines=10000,
    data_type='temperature', data_range=None, regions=None)
```

By default, the report is automatically compressed as `instance_temperature_all.rpt.gz`. If `max_lines` are specified to sort the report, the default report is stored as `instance_temperature_max_lines.rpt.gz`.

The following figure is a snippet from an instance temperature report:

# instance	loc_x	loc_y	temperature_ambient	temperature	self_heating
cts_inv_590761458	262.9	237.3	125	137.8	12.8449001312
cts_inv_589961450	266.9	426.3	125	137.3	12.3017997742
cts_inv_590461455	227	298.9	125	136.8	11.8360004425
cts_inv_526460815	766	1198	125	136.4	11.3878002167
cts_inv_589261443	167.1	343.7	125	136.3	11.2544002533
cts_inv_558361134	145.1	805.7	125	136.2	11.2033004761
cts_inv_556561116	37.34	889.7	125	136.1	11.13560009
cts_inv_588661437	599.9	282.1	125	136	10.975399971
cts_inv_557461125	66.59	842.1	125	136	10.9715003967
cts_inv_555361104	269.7	556.5	125	135.9	10.9307003021
cts_inv_586261413	135.8	438.9	125	135.9	10.8695001602
cts_inv_587261423	233.6	38.5	125	135.9	10.8501996994
cts_inv_554061091	227.9	1198	125	135.8	10.8437995911
cts_inv_555461105	414.1	1009	125	135.8	10.7988004684
cts_inv_558861139	293.3	836.5	125	135.8	10.7748003006
cts_inv_551761068	34.3	584.5	125	135.7	10.6898002625
cts_inv_559061141	200.4	833.7	125	135.7	10.6610002518
cts_inv_530060851	748.5	1013	125	135.5	10.5207996368
cts_inv_557861129	340.8	937.3	125	135.4	10.4089002609
cts_inv_587661427	411.1	32.9	125	135.4	10.3877000809

Use the below help function to view updated syntax and functions:

```
help(ThermalView.report_instance_temperatures, detailed=True)
```

13.4.2. Reporting Edge Temperature

Edge temperature report includes path to report file and details of temperature, self_heating, joule_heating, instance_coupling and wire_to_wire_coupling heatmap. Use the following API command to report edge temperatures:

```
ThermalView.report_edge_temperatures(report_file=None, columns=['net', 'layer',
    'loc_x',
    'loc_y', 'temperature_ambient', 'temperature', 'self_heating',
    'joule_heating', 'instance_coupling', 'wire_to_wire_coupling'],
    sort=True, sort_order='descending', sort_columns=['self_heating'],
    formats=None, header=None, footer=None, max_lines=10000, layers=None,
    net_type='pg', nets=None, data_type='temperature', data_range=None,
    regions=None)
```

By default, the report is automatically compressed as edge_temperature_all.rpt.gz. If max lines are specified to sort the report, the default report is stored as edge_temperature_max_lines.rpt.gz.

The following figure is a snippet from edge temperature report:

```
# net layer loc_x loc_y temperature_ambient temperature self_heating joule_heating instance_coupling wire_to_wire_coupling sliver
core3/VDD_INT metall1 675 343 125 178.5 53.51 0.0298 2.432 51.0429992676 True
core3/VDD_INT metall1 674.5 343.2 125 178.5 53.51 0.04161 2.432 51.0311927795 True
core3/VDD_INT metall1 675.4 345.8 125 178.5 53.51 0.04992 2.432 51.0228805542 True
core3/VDD_INT metall1 676.7 345.8 125 178.5 53.51 0.05005 2.432 51.0227470398 True
core3/VDD_INT metall1 677.7 345.8 125 178.5 53.51 0.04967 2.432 51.0231285095 True
core3/VDD_INT metall1 677.4 346 125 178.5 53.51 0.04745 2.432 51.0253486633 True
core3/VDD_INT metall1 674.5 348.8 125 178.5 53.51 0.0888 2.432 50.9840049744 True
core3/VDD_INT metall1 677.8 348.8 125 178.5 53.51 0.04967 2.432 51.0231323242 True
core3/VDD_INT metall1 678.6 351.6 125 178.5 53.51 0.0575 2.432 51.0153045654 True
core3/VDD_INT metall1 674.9 354.4 125 178.5 53.51 0.02992 2.432 51.0428771973 True
core3/VDD_INT metall1 675.4 362.6 125 178.5 53.51 0.1979 2.432 50.8749198914 True
core3/VDD_INT metall1 673.8 362.8 125 178.5 53.51 0.1013 2.432 50.9715270996 True
core3/VDD_INT metall1 673.7 365.4 125 178.5 53.51 0.02602 2.432 51.0467834473 True
core3/VDD_INT metall1 674.2 365.6 125 178.5 53.51 0.02464 2.432 51.0481567383 True
core3/VDD_INT metall1 674.8 368.2 125 178.5 53.51 0.04935 2.432 51.0234489441 True
core3/VDD_INT metall1 674.2 368.4 125 178.5 53.51 0.02832 2.432 51.0444831848 True
core3/VDD_INT metall1 675.3 368.4 125 178.5 53.51 0.02115 2.432 51.0516471863 True
core3/VDD_INT metall1 676.4 371 125 178.5 53.51 0.04255 2.432 51.030254364 True
core3/VDD_INT metall1 675.1 371.2 125 178.5 53.51 0.06214 2.432 51.0106582642 True
core3/VDD_INT metall1 676.2 376.6 125 178.5 53.51 0.07753 2.432 50.9952697754 True
core3/VDD_INT metall1 676.5 376.8 125 178.5 53.51 0.07347 2.432 50.999332428 True
core3/VDD_INT metall1 675.5 379.4 125 178.5 53.51 0.02837 2.432 51.0444335938 True
core3/VDD_INT metall1 674.7 382.2 125 178.5 53.51 0.08937 2.432 50.9834327698 True
core3/VDD_INT metall1 674.5 382.4 125 178.5 53.51 0.04527 2.432 51.0275306702 True
core3/VDD_INT metall1 674.3 385 125 178.5 53.51 0.04228 2.432 51.0305175781 True
core3/VDD_INT metall1 673.1 385.2 125 178.5 53.51 0.03399 2.432 51.0388069153 True
core3/VDD_INT metall1 677.7 385.2 125 178.5 53.51 0.027 2.432 51.0457992554 True
"pg edge temperatures.gz" [noeo1] 211L, 70164C
1,1
```

Use the below help function to view updated syntax and APIs:

```
help(ThermalView.report_instance_temperatures, detailed=True)
```

Reporting Custom Edge Temperatures

To generate a report for any specific edge temperature, use the following API command:

```
ThermalView.get_edge_temperature(layer, coord, net)
```

Table 21: Valid Arguments for Edge Temperatures

Argument	Description
layer	Layer object (type=Layer, required=True)
coord	Location to query in microns (type=RealCoord, required=True, constraint="RealCoord in microns")
net	Net object (type=Net, required=True)

For example, to report the edge temperatures for metall1 layer with VDD_INT net type, use the following API command:

```
thmv_flow1.get_edge_temperature(Layer('metall1'),
RealCoord(1069.545000,544.670000),
Net('core3/VDD_INT'))
```

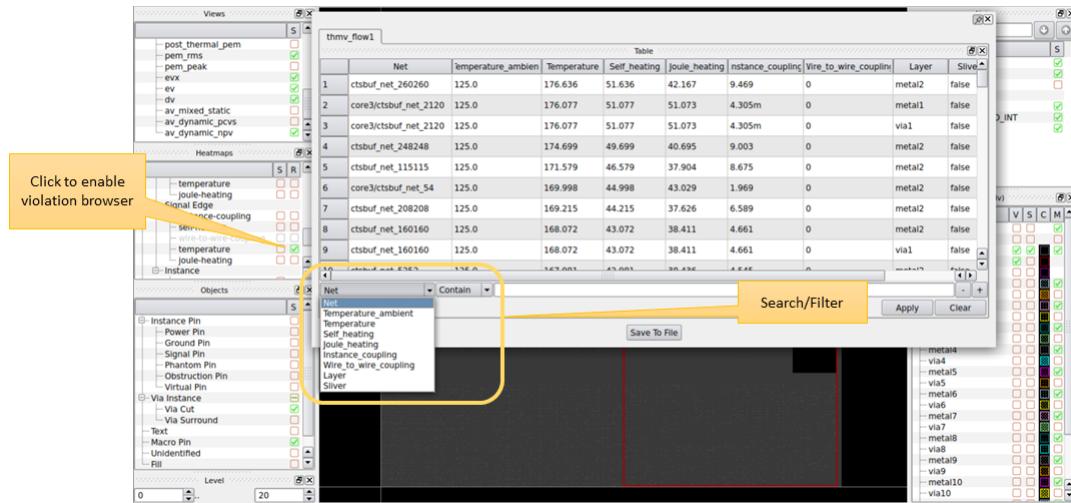
The command gives the following edge temperature report:

```
[{'from': RealCoord(1069.435000,544.645000),
'temperature': 131.19273376464844,
'temperature_ambient': 125.0,
'to': RealCoord(1069.815000,544.645000),
'self_heating': 6.192729949951172,
'instance_coupling': 1.9693700075149536,
'sliver': True,
'joule_heating': 4.223360061645508}]
```

13.5. Reviewing Results in GUI

RedHawk-SC GUI report displays top 1000 violations based on different parameters, such as `temperature`, `joule_heating`, `instance_coupling`, `wire-to-wire_coupling`.

The following figure shows violations in RedHawk-SC GUI:

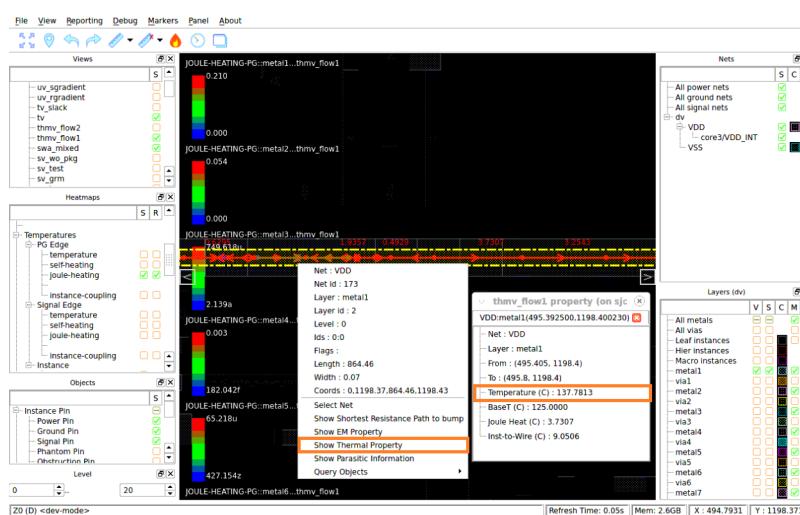


The **Heatmaps** window in the left pane of the GUI has different categories to segregate the violations. You can select options under **Instance**, **PG Edge** or **Signal Edge** to view violations for respective categories.

Locating an Issue

By default, the GUI shows violations with respect to the **temperature** heatmap. To enable violations for other heatmaps, select the suitable option from the left pane.

1. In violations report, double-click the thermal violation to locate the wire or via location.
2. Right-click the edge, and select **Show Thermal Property** option from right-click menu.
3. The following figure shows thermal property of the selected edge.



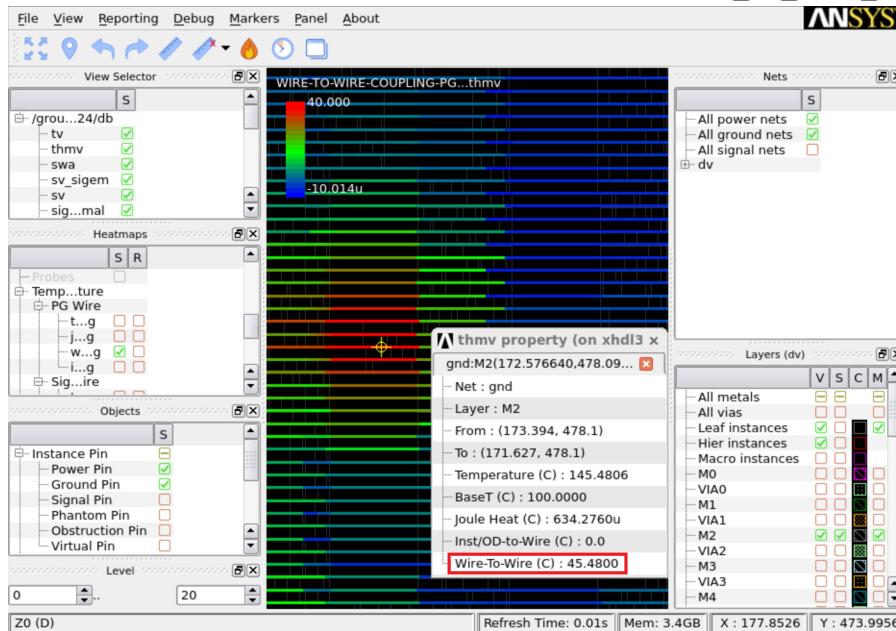
Fixing a wire_to_wire_coupling Issue

wire_to_wire_coupling violations are caused when neighboring wire segments have higher joule_heating temperatures.

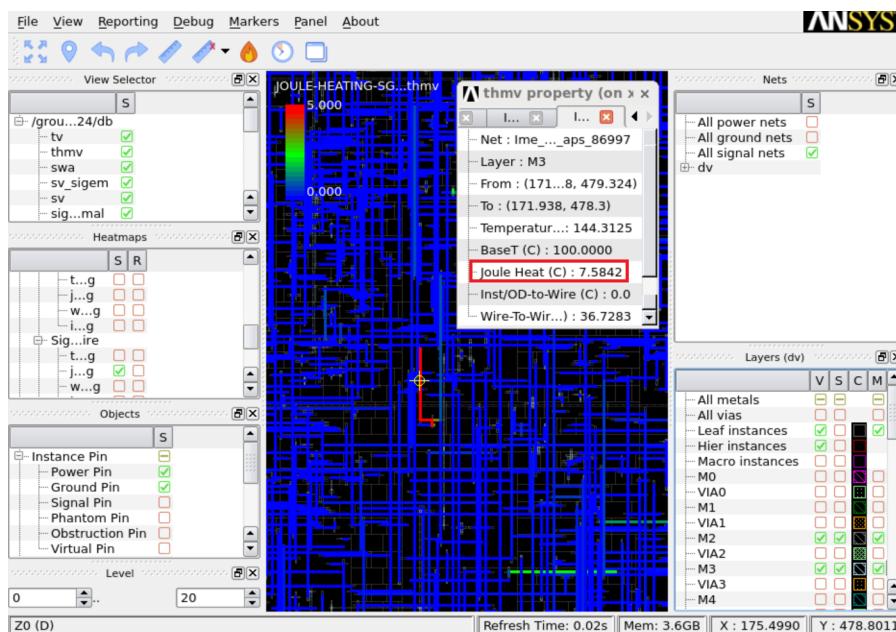
To debug such issues, follow these steps:

1. Invoke two RedHawk-SC GUI windows for a single database.
2. Enable wire_to_wire_coupling in one layout window and joule_heating in another layout window.
3. Consider the following example:

- a. The following figure shows a wire segment with 45.48C wire_to_wire_coupling.



- b. Find the nearest wire segment with higher joule_heating as shown in the following figure.

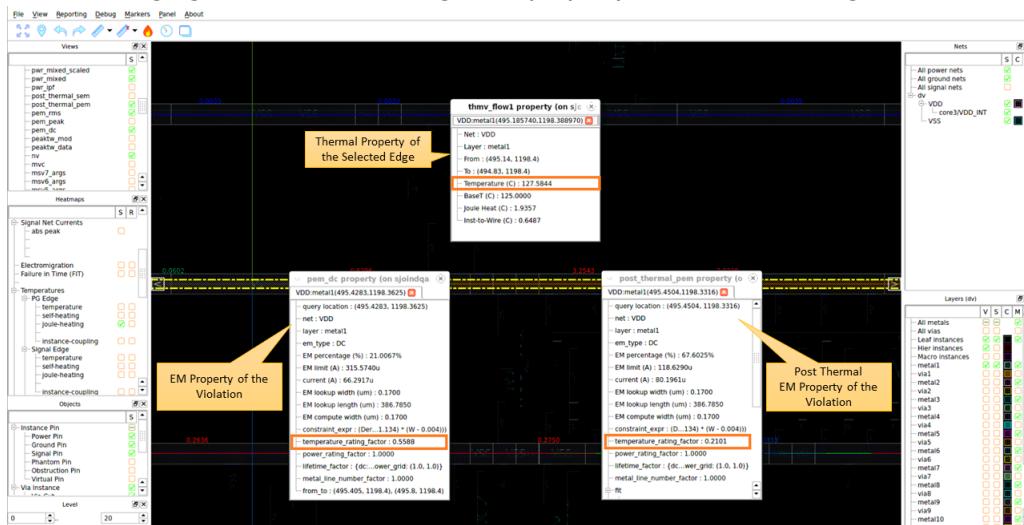


4. Fix the thermal issue by comparing the results, and carrying out appropriate design changes.

Reviewing Post Thermal Electromigration Results

You can use the output from thermal analysis to back-annotate the results and re-compute Electromigration limits. Use the following procedure to review the results in GUI:

1. From the left pane, configure Views to review `pem_dc` and `post_thermal_pem` properties.
2. In violations report, double-click the thermal violation to locate the wire/via location.
3. Right-click the edge, and select **Show EM Property** option from right-click menu. A file browser appears.
 - a. From the dropdown menu, select `pem_dc` to review pre-thermal electromigration properties.
 - b. From the dropdown menu, select `post_thermal_pem` to review post-thermal electromigration properties.
4. The following figure shows electromigration property of the selected edge.



5. Review the results. Observe that the post-thermal electromigration property shows that temperature rating factor has decreased.

14: Power Profiling Across Long Vectors

Power calculation across long duration leads to high runtime and memory consumption. To overcome this challenge, PowerProfileView (PPV) enables you to perform power profiling across long vectors. PPV helps in analyzing and identifying high peak power, di/dt , and moving average windows to perform detailed power integrity analysis. With a flexible user interface, PPV takes RTL or gate level VCDs, where events from sequential points are propagated to all the instances in the design. This also applies to FSDB files.

PPV supports tile or region based profiling where power is calculated for each tile. An integrated command, `perform_vector_selection`, is available for vector window selection. There are some utilities to find out windows of high power and high change in power.

The following topics are discussed in this chapter:

- [Power Profile Inputs](#) on page 411
 - [Power Profile Outputs](#) on page 416
 - [Utilities for Vector Window Selection](#) on page 424
 - [Integrated Command for Vector Selection](#) on page 425
-

14.1. Power Profile Inputs

PPV has multiple settings to control the accuracy, modes of calculations, and the amount of data to be stored. To create PPV, use `<SeaScapeDB>.create_power_profile_view` command. For more details, use `help(SeaScapeDB.create_power_profile_view)` command at the tool command prompt.

TimingView

TimingView stores imported STA file and SDC constraints. Timing window file is a mandatory input to PPV.

Signal Parasitics

ExtractView imports SPEF file, which is an input to PPV. Additionally, you can pass parasitics information of un-annotated nets in SPEF file in the PG only ExtractView.

Vector Information

ValueChangeView (VCV) reads VCD or FSDB files to get events information. File name, preamble, top or block settings, and time slices with start and stop time are given as additional inputs. You can pass `value_change_data` to PPV. `slice_name` key plays an important role in the flow as top or block level FSDB, and the right vector duration to do analysis are setup using the slice names.

Voltage Levels

An important input that stores voltage for each power net in the design.

```
voltage_levels= {'VDD1':0.88, 'VDD2':0.75, 'VSS':0}
```

VCD Processing Controls per Scope

PPV allows scope level control for certain settings at the design (top instance) level, block-level, master-cell level or at instance level.

```
object_settings = {
    'design_values': {'vcd_mode': 'non_true_time'},
    'block_values': [{{'pattern': 'hier3/block1', 'vcd_mode': 'true_time'},
                     {'pattern': 'hier3/block2', 'vcd_mode': 'non_true_time'}}, ]}
```

Interval Size and Scenario Duration

The `time_interval_length` is a settings key, which depicts sampling interval for power. Setting low time interval results in high resolution of power profiling but increased runtime and memory.

`scenario_duration` is the total time duration for which PPV is created.

The following example shows how to use these settings:

```
settings={'time_interval_length':100e-09}
ppv_args = dict(...,
               scenario_duration=5e-06,
               settings=settings,...)
ppv = db.create_power_profile_view(...,**ppv_args)
```

The `create_power_profile_view` and `perform_vector_selection` commands automatically consider `scenario_duration` from the input VCD data. If `scenario_duration` is not set, the tool uses the maximum duration of the slice in `value_change_data` as `scenario_duration`.

PPV removes partial time interval when `scenario_duration` is not an integer multiple of `time_interval_length` in the `create_power_profile_view` and `perform_vector_selection` commands. The tool drops the partial interval with the following warning message:

```
WARNING<PPV.107> scenario_duration (6.50000000000022e-07) is not an integer
multiple of time_interval_length (7e-08), the last interval will be dropped
for analysis.
```

Calculating Power

To control power calculations across various power components, `calculation` key is added under `settings`. The first level keys such as `leakage`, `internal`, `switching` have multiple sub-keys. You can control the accuracy level by setting the accuracy key to `zero`, `low`, `medium`, or `high`.

To calculate switching power, there are two ways to consider power associated with rise and fall transitions:

- Full switching energy of `[load_cap * voltage * voltage]` for each rise transition and none for fall transitions. This is the default behaviour.
- Half energy for each rise transition and half energy for each fall transition.

The `transition_count` key helps in controlling this behaviour with `rise_only` and `rise_plus_fall_div_2` values.

The internal power is calculated from energy tables in the Liberty file. The energy tables are 2D tables with transition time and load capacitance values. If these values exceed the table limits, then extrapolation is not done by default. However, you can set `nlpm_extrapolation` to `True` for doing extrapolation. This is a less used setting and only needed for specific needs.

The `assign_cell_leakage_to_primary_power` key is used only when the leakage power in the Liberty file does not have pin-based values. By default, the leakage power is applied only on primary power pin. You can set this key to `False` to divide leakage power equally among all power pins. This is a less used setting. The default value is best for most cases.

For more details about calculation settings, use

`help(SeaScapeDB.create_power_profile_view)` command at the tool command prompt.

```
calculation_settings = { 'calculation': {
    'internal': {'nlpm_extrapolation': False, 'accuracy': 'medium'},
    'leakage': {'assign_cell_leakage_to_primary_power': False, 'accuracy': 'medium'},
    'switching': {'transition_count': 'rise_only', 'accuracy': 'medium'},
    'limit_net_cap_to_pin_max_cap': False}}
ppv = db.create_power_profile_view(..., settings=calculation_settings, **ppv_args)
```

VCD Settings

In the vector selection flow, `vcd` is a dict to control vector-based flow settings. It has the following keys:

- `clock_frequency_precedence`: a list `['vcd', 'sta']`, describing precedence for frequency source between VCD or FSDB and STA file. Default is `['vcd']`.
- `overrides_lso`: if `True`, VCD based signal gets higher precedence over LSO based signal on any instance pin. Default is `False`.
- `apply_on_sequential_launch_only`: if `True`, annotate VCD based signals only on sequential instances. Default is `False`.

```
settings = {'vcd': {'clock_frequency_precedence': ['vcd', 'sta'],
    'overrides_lso': True, 'apply_on_sequential_launch_only': True}}
```

Storing Detailed Analysis Results

By default, PPV stores interval based power data of the top block per power domain. There are multiple `object_settings` to store data for design, block, and leaf level instances:

- `results`: a dict describing the results that should be generated for this scope. A results dict includes the following keys:
 - `interval_based`: it creates power results over time interval when set to `True` and it creates average power results over the `scenario_duration` when set to `False`.
 - `save_policy`: a list of strings describing how the power data is saved for the scope. Valid values are `['supply', 'clock_domain', 'cell_category']`. If not specified, default value is `['supply']`.
- `leaf_instance_results`: a dict describing the results that should be generated for leaf instances in this scope. This key is not valid in `leaf_instance_values`.

```
object_settings_ppv = {
    'design_values' : {
        'results' : {'interval_based': True, 'save_policy': ['supply', 'cell_category']},
        'calculate_power': True,
        'leaf_instances_results' : {'interval_based': False}},
    'block_values' : [{ 'pattern': 'blockA1', 'results' : {'interval_based': True}}]}
```

Tile Based Power

PPV supports tile-based profiling that allows power numbers per interval to be stored for each tile. Tile-based profiling helps in voltage drop analysis by calculating the highest power across regions to uncover local hotspots. The tool generates tile results when reporting is enabled for both block level and tile-based power.

`tile_results` is a dict describing the results that should be generated for physical tiles. This key is only valid within `design_values`. Within `tile_results`, `tile_width` and `tile_height` are mandatory inputs (keys) which determine the dimensions of tile. The `tile_width` is input in microns.

```
object_settings = {
    'design_values': {'calculate_power': True,
                      'leaf_instance_results': {'interval_based': False},
                      'results': {'interval_based': True,
                                  'save_policy': ['supply']},
                      'tile_results': {'interval_based': True,
                                      'save_policy': ['supply',
                                                      'clock domain',
                                                      'cell category'],
                                      'tile_height': 5.0,
                                      'tile_width': 5.0}}}
```

Usage Examples

When delay annotated VCD is given for a top level design and a block. Data is stored for a specific block.

```
vcd_files_vcv = [
    {'file_name': design_data_path + '/fsdb/block1_sdf_vcd.fsdb',
     'instances': ['block1'],
     'time_slices': [{slice_name: 'fsdb_block1', 'start_time': 188e-06,
                      'stop_time': 193e-06}],
     'preamble': 'testbench1/sys_ch01/blockC_inst'},
    {'file_name': design_data_path + '/fsdb7/top_Galaxy_vcd.fsdb',
     'instances': [''],
     'time_slices': [{slice_name: 'fsdb_top', 'start_time': 48e-06,
                      'stop_time': 52e-06}],
     'preamble': 'top'},
],
vcd_args = dict(
    vcd_files=vcd_files_vcv,
    options=options,
    tag='vcv')
vcv = db.create_value_change_view(dv, **vcd_args)

value_change_data = [
    {'view': vcv, 'slice_name': 'fsdb_block1'},
    {'view': vcv, 'slice_name': 'fsdb_top'}]
object_settings = {
    'design_values': {
        'vcd_mode': 'true_time',
        'block_values': [
            {'pattern': 'block1',
             'results': {
                 'interval_based': True}}]}}

ppv_settings = {
    'object_settings': object_settings,
    'time_interval_length': 20e-09,
    'pvt': {'voltage_levels': {'VSS': 0.0, 'VDD2': 0.91, 'VDD1': 0.91, }}},
```

```

        }

ppv_args = dict (
    scenario_duration=5e-06,
    value_change_data=value_change_data,
    settings=ppv_settings
    options=options,
    tag='ppv')
ppv = db.create_power_profile_view(timing_view=tv, external_parasitics=evx,
extract_view=ev, **ppv_args)

```

When a zero delay top-level VCD and a delay annotated block VCD are inputs. Data is stored for all the blocks. Extra cell category wise power is stored for design level.

```

value_change_data = [
{'view' : vcv, 'slice_name' : 'top_zero_delay'},
{'view' : vcv, 'slice_name' : 'hier3_sdf'}]
object_settings = {
'design_values' : {
'vcd_mode' : 'non_true_time',
'results': {
'interval_based' : True,
'save_policy': ['supply', 'cell_category']},},
'block_values' : [
{'pattern' : 'hier3',
'vcd_mode' : 'true_time'},
{'pattern' : '.*',
'results' : {
'interval_based' : True}}],}

ppv_settings = {
'object_settings':object_settings,
'time_interval_length':10e-09,
'pvt': {'voltage_levels': {'VSS' : 0.00, 'VDD' : 0.8}},
}
ppv_args = dict (
    scenario_duration=3e-06,
    value_change_data=value_change_data,
    settings=ppv_settings,
    options=options,
    tag='ppv')
ppv = db.create_power_profile_view(timing_view=tv, external_parasitics=evx,
extract_view=ev, **ppv_args)

```

When a zero delay top-level VCD is given. Data is stored for design level, and average power for all leaf instances.

```

value_change_data = [
{'view' : vcv, 'slice_name' : 'top_zero_delay'},]
object_settings = {
'design_values' : {
'vcd_mode' : 'non_true_time',
'results': {
'interval_based' : True,
'save_policy': ['supply', 'clock_domain']},
'leaf_instance_results' : {
'interval_based': False}}}
ppv_settings = {
'object_settings':object_settings,

```

```

'time_interval_length':2e-09,
'pvt': {'voltage_levels': {'VSS' : 0.00, 'VDD' : 0.8}}},

ppv_args = dict (
    scenario_duration=1.5e-06,
    value_change_data=value_change_data,
    settings=ppv_settings
    options=options,
    tag='ppv')
ppv = db.create_power_profile_view(timing_view=tv, external_parasitics=evx,
extract_view=ev, **ppv_args)

```

A delay annotated top level VCD. Stores tile-based power and `cell_category` power for all blocks.

```

value_change_data = [
{'view' : vcv, 'slice_name' : 'top_sdf'},]
object_settings = {
'design_values' : {
'vcd_mode' : 'true_time',
'results': {
'interval_based' : True,},,
'tile_results' : {
'tile_height': 50,
'tile_width' : 100,
'interval_based': True},,,
'block_values': [
{'pattern' : '.*',
'results' : {
'interval_based' : True,
'save_policy' : ['supply', 'cell_category']}]},}
}

ppv_settings = {
'object_settings':object_settings,
'time_interval_length':3.5e-09,
'pvt': {'voltage_levels': {'VSS': 0.00, 'VDD' : 0.8}},}
}

ppv_args = dict (
    scenario_duration=7e-06,
    value_change_data=value_change_data,
    settings=ppv_settings,
    options=options,
    tag='ppv')
ppv = db.create_power_profile_view(timing_view=tv, external_parasitics=evx,
extract_view=ev, **ppv_args)

```

14.2. Power Profile Outputs

Like other views in SeaScape, PPV stores all the data in disk and enables access through functions. The data is stored as an advanced data structure, which is fast to access. There are also some built-in scripts that process PPV data. You can use your own scripts to extract and process data.

Design and Block Level Power

The `<ppv_name>.get_results()` function allows you to retrieve power results such as total duration of PPV run, domain wise power, power for block or leaf instances, and tile indices. For more details, use `help(ppv_name.get_results)` at the tool command prompt.

In the following example, if `intervals` is passed as a key of `<ppv_name>.get_results()` function, then it returns 3640 intervals.

```
>>> len(ppv.get_results()['intervals'])
3640
>>> ppv.get_results()['intervals'][0]
(0.0, 1.999999987845058e-08)
>>> ppv.get_results()['intervals'][1]
(1.999999987845058e-08, 3.999999975690116e-08)
```

The `top` key returns domain wise power for the whole design. The index of list matches with the index of the intervals. If the first interval is 0-20 ns and second interval is 20-40 ns, then

`ppv.get_results()['top'][Net('VDD1')][0].get_switching_power()` returns switching power of first interval and `ppv.get_results()['top'][Net('VDD1')][1].get_internal_power()` returns internal power of second interval.

```
>>> ppv.get_results()['top'].keys()
[Net('VDD1'), Net('VDD2')]
>>> type(ppv.get_results()['top'][Net('VDD1')])
<class 'gp_export_py.EEDetailedPowerVector'>
>>> print ppv.get_results()['top'][Net('VDD1')][0]
{'total' : 0.0138827, 'switching' : 0.00499244, 'internal' : 0.00793883,
 'leakage' : 0.000951392, 'clock_pin' : 0, 'glitch' : 0, }
>>> print ppv.get_results()['top'][Net('VDD1')][1]
{'total' : 0.0129075, 'switching' : 0.00455984, 'internal' : 0.00739621,
 'leakage' : 0.000951392, 'clock_pin' : 0, 'glitch' : 0, }
>>> ppv.get_results()['top'][Net('VDD1')][1].get_switching_power()
0.004559843335300684
>>> ppv.get_results()['top'][Net('VDD1')][45].get_total_power()
0.012928158044815063
```

The `instance` key also works in the similar way but with extra hierarchy of instances.

```
>>> ppv.get_results()['instance'].keys()
[Instance('inst1'), Instance('inst2')]
>>> ppv2.get_results()['instance'][Instance('inst1')][Net('VDD1')]
[2].get_total_power()
9.940560374843699e-09
```

The following example shows the complete output of `get_results`. PPV has scenario duration of 30 ns with three intervals of 10 ns and power of two specific instances.

```
>>> ppv2.get_results()
{'duration': 2.999999892949745e-08,
 'instance': {Instance('inst1'): {Net('VDDD1V5'):
 CMVector<ee_pwr::DetailedPower>([{'total' : 9.94056e-09, 'switching' : 0,
 'internal' : 0,
 'leakage' : 9.94056e-09, 'clock_pin' : 0, 'glitch' : 0, }, {'total' :
 4.18389e-05,
 'switching' : 1.51515e-05, 'internal' : 2.66775e-05, 'leakage' : 9.94056e-09,
 'clock_pin' : 0,
 'glitch' : 0, }, {'total' : 9.94056e-09, 'switching' : 0, 'internal' : 0,
```

```

'leakage' : 9.94056e-09,
'clock_pin' : 0, 'glitch' : 0, },]},  

Instance('inst2'): {Net('VDDD1V5'):  

CMVector<ee_pwr::DetailedPower>([{'total' : 3.59415e-06, 'switching' : 0,  

'internal' : 3.58748e-06,  

'leakage' : 6.66944e-09, 'clock_pin' : 3.58748e-06, 'glitch' : 0, },{'total'  

: 5.21057e-06,  

'switching' : 0, 'internal' : 5.2039e-06, 'leakage' : 6.66944e-09, 'clock_pin'  

: 2.87949e-06,  

'glitch' : 0, },{'total' : 1.71874e-06, 'switching' : 0, 'internal' :  

1.71207e-06,  

'leakage' : 6.66944e-09, 'clock_pin' : 1.71207e-06, 'glitch' : 0, },]},  

'top': {Net('VDDD3V3'): CMVector<ee_pwr::DetailedPower>([{'total' : 0.014417,  

'switching' : 0,  

'internal' : 0.00333195, 'leakage' : 0.0110851, 'clock_pin' : 0, 'glitch' :  

0, },  

{'total' : 0.0114739, 'switching' : 0.000239432, 'internal' : 0.000149391,  

'leakage' : 0.0110851,  

'clock_pin' : 0, 'glitch' : 0, },{'total' : 0.0145928, 'switching' : 0,  

'internal' : 0.0035077,  

'leakage' : 0.0110851, 'clock_pin' : 0, 'glitch' : 0, }]),  

Net('VDDD1V5'): CMVector<ee_pwr::DetailedPower>([{'total' : 0.0747315,  

'switching' : 0.0382604,  

'internal' : 0.035585, 'leakage' : 0.000954241, 'clock_pin' : 0.0256908,  

'glitch' : 0, },  

{'total' : 1.09851, 'switching' : 0.615335, 'internal' : 0.48228, 'leakage' :  

0.000954241,  

'clock_pin' : 0.0292249, 'glitch' : 0, },{'total' : 0.0770548, 'switching' :  

0.019929,  

'internal' : 0.0561891, 'leakage' : 0.000954241, 'clock_pin' : 0.0496814,  

'glitch' : 0, }]),  

'intervals': [(0.0, 9.9999993922529e-09), (9.9999993922529e-09,  

1.99999987845058e-08),  

(1.99999987845058e-08, 2.999999892949745e-08)]}

```

Leaf Instance Power

The `ppv.get_instance_power_data()` function enables data for all the leaf instances. It takes `instance_name` key and returns output similar to that of `get_results`. For example,

```

>>> ppv.get_instance_power_data(Instance('instance_name'))
{Pin('VDD'): CMVector<ee_pwr::DetailedPower>([{'total' : 9.94056e-09,  

'switching' : 0,  

'internal' : 0, 'leakage' : 9.94056e-09, 'clock_pin' : 0, 'glitch' : 0, },  

{'total' : 4.18389e-05,  

'switching' : 1.51515e-05, 'internal' : 2.66775e-05, 'leakage' : 9.94056e-09,  

'clock_pin' : 0,  

'glitch' : 0, },{'total' : 9.94056e-09, 'switching' : 0, 'internal' : 0,  

'leakage' : 9.94056e-09,  

'clock_pin' : 0, 'glitch' : 0, }])

```

Cell Category and Clock Domain Wise Power

The `ppv.get_results` function returns per supply power for total design as well as for each block with data storage. However, `ppv.get_results` function does not return per clock source or per cell category power. For this purpose, `ppv.get_block_power_data()` function is added. The output of `ppv.get_block_power_data()` function is a python dict with keys as a combination of clock, supply, and cell categories.

For example, `ppv.get_block_power_data` returns per supply power by default.

```
>>> ppv.get_block_power_data(Instance('')).keys()
[Net('VDD_1V2'), Net('VDD_EXT'), Net('Inst3_hier2/VDDI_t3')]
```

The `by_supply=False` does not report power per supply. The `by_cell_category=True` returns power per cell category. These are cell categories for which power is stored.

```
>>>
ppv.get_block_power_data(Instance('hier34/hier56'),by_supply=False,by_cell_category=True).keys()
['macro', 'flip_flop', 'combinational', 'pad', 'unknown', 'latch']
```

-1 refers to the default clock, for which no clock domain is associated. To check clock source IDs and names, use the `<TimingView>.get_clock_attributes()` function.

```
>>>
ppv.get_block_power_data(Instance('block3'),by_supply=False,by_clock_domain=True).keys()
[482, 33, -1]
```

When combination of more than one category is given, the keys become a python tuple of three entries, as (`<power_supply>`, `<clock_domain>`, `<cell_category>`). When one of the keys is missing, it is marked as 'unknown_supply', '-1', and 'unknown' respectively.

```
>>>
ppv.get_block_power_data(Instance('block8'),by_supply=False,by_cell_category=True,
by_clock_domain=True).keys()
[('unknown_supply', -1, 'combinational'), ('unknown_supply', -1, 'macro'),
('unknown_supply', -1, 'flip_flop'), ('unknown_supply', -1, 'latch'),
('unknown_supply', 482, 'unknown'),
('unknown_supply', -1, 'pad')]
```

When a cell category is missing, it is marked as `unknown`. All the other combinations between clock domain and power domain are present.

```
>>> ppv.get_block_power_data(Instance(''),by_supply=True,by_cell_category=False,
by_clock_domain=True).keys()
[(Net('VDD_1V2'), 33, 'unknown'), (Net('VDD_1V2'), 482, 'unknown'),
(Net('VDD_EXT'), 482, 'unknown'),
(Net('VDD_EXT'), 33, 'unknown'), (Net('Inst3_hier2/VDDI_t3'), 482, 'unknown'),
(Net('Inst3_hier2/VDDI_t3'), 33, 'unknown'),]
```

With three power domains, two clock domains and six cell categories, there are 36 combinations.

```
>>> ppv.get_block_power_data(Instance(''),by_supply=True,by_cell_category=False,
by_clock_domain=True).keys()
[(Net('VDD_1V2'), 33, 'macro'), Net('VDD_1V2'), 33, 'flip_flop'),
Net('VDD_1V2'), 33, 'pad'),
Net('VDD_1V2'), 33, 'unknown'), Net('VDD_1V2'), 33, 'combinational'),
Net('VDD_1V2'), 33, 'latch'),
(Net('VDD_1V2'), 482, 'macro'), Net('VDD_1V2'), 482, 'flip_flop'),
Net('VDD_1V2'), 482, 'pad'),
Net('VDD_1V2'), 482, 'unknown'), Net('VDD_1V2'), 482, 'combinational'),
Net('VDD_1V2'), 482, 'latch'), ...]
```

The value of key is the same data type of `EEDetailedPowerVector`, which is mentioned earlier.

```
>>>
ppv.get_block_power_data(Instance(''), by_supply=True, by_cell_category=True, by_clock_domain=True)
[(Net('VDD_EXT'), -1, 'macro')]
<gp_export_py.EEDetailedPowerVector object at 0x2b4a90f05450>
>>> print
ppv.get_block_power_data(Instance(''), by_cell_category=True,
by_clock_domain=True)
[(Net('VDD_EXT'), -1, 'macro')][45]{'total_power' : 5.90611e-05,
'switching_power' : 0,
'internal_power' : 0, 'leakage_power' : 5.90611e-05, 'clock_pin_power' : 0,
'glitch_power' : 0, }
>>> ppv.get_block_power_data(Instance(''), by_cell_category=True,
by_clock_domain=True)
[(Net('VDD_EXT'), -1, 'macro')][45].get_total_power()
5.90611416555e-05
>>> ppv.get_block_power_data(Instance(''))[Net('VDD')][12]
EEDetailedPower(1.43443, 0.496434, 0.487381, 0.451901, 0.270244, 0)
```

Tile-Based Power

The design is divided into rectangular partitions or regions, called tiles. RedHawk-SC calculates power for each tile for each interval. There are functions to access built-in utilities to visualize power observed in each tile. The `ppv.get_results()['tile'].get_all_tile_indices()` and `ppv.get_tile_power_data(2,3,by_clock_domain=True,by_cell_category=True).keys()` functions report tile location and corresponding indices as shown in the following example.

```
>>> ppv.get_results()['tile'].get_all_tile_indices()
[(0, 0), (0, 1), (0, 2), (0, 3), ..... (11, 10), (11, 11)]
>>> ppv.get_results()['tile'].get_tile_box(0,0)
Rect(-1516000000,-1481000000,-1266000000,-1231000000)
>>> ppv.get_results()['tile'].get_tile_box(9,9)
Rect(734000000,769000000,984000000,1019000000)
```

Here, `get_tile_power_data` function stores cell category and clock domain based power.

```
>>>
ppv.get_tile_power_data(2,3,by_clock_domain=True,by_cell_category=True).keys()
[(Net('vbat_3v_filtered'), 53, 'combinational_clock'), ('unknown_supply', 52,
'combinational'),
('unknown_supply', 92, 'unknown'), (Net('vbat_3v_filtered'), 43,
'combinational'),
(Net('vbat_3v_filtered'), 73, 'latch'), ('unknown_supply', 53, 'flip_flop'),
(Net('vbat_3v_filtered'), 73, 'sequential_clock'), (Net('vbat_3v_filtered'),
73, 'combinational'), (Net('vbat_3v_filtered'), 92, 'unknown')]
```

The `ppv.get_tile_power_data()` function is similar to `get_block_power_data()` function, tile indices is input instead of instance object and rest of the inputs remain the same. Output is a dict with keys as a combination of categories and `EEDetailedPowerVector`.

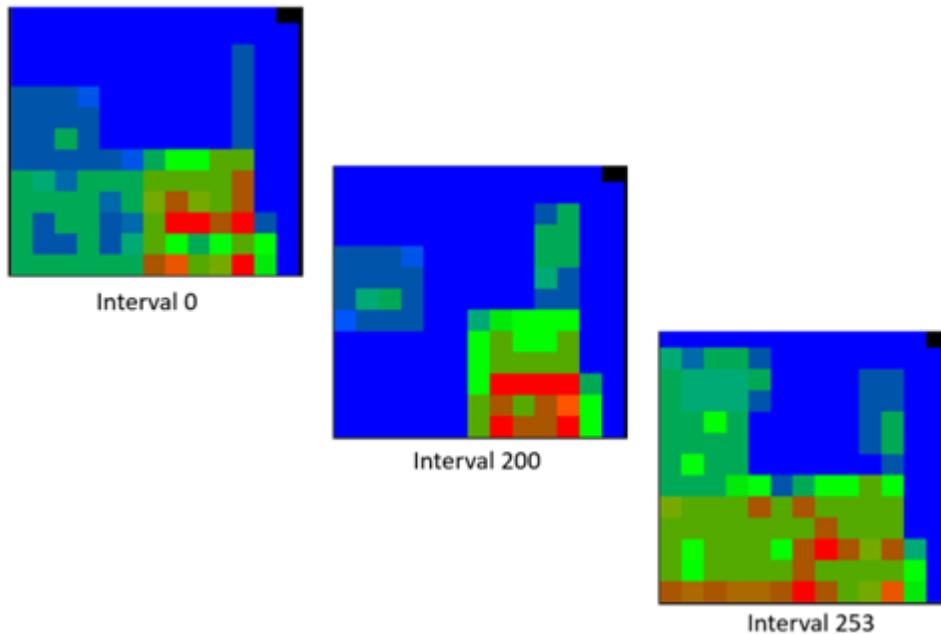
Visualizing Tile-Based Power

Tile-based profiling enables you to see the movie or animation (GIF) of changing power across region. To save the `tile_power` heatmap picture as movie to replay, use the `generate_ppv_movie(ppv, output_file='./ppv.gif', delay=0.3)` command.

The `ppv_utils` module has certain functions to create tile-based heatmaps and movies from the PPV results. In the following example, there is a delay of 300 ms between heatmap of each interval.

```
>>> import ppv_utils
>>> heatmaps = ppv_utils.create_heatmaps_for_intervals(ppv.get_results(),
    ppv.get_results()['tile'], dv)
>>> len(heatmaps)
325
>>> gui.add_layer(heatmaps[100], '100')
>>> gui.add_layer(heatmaps[200], '200')
>>> gui.add_layer(heatmaps[300], '300')
```

Figure 74. Tile-Based Power Heatmaps from Different Intervals



Visualizing Power Over Time

SeaScape utilizes `Waveform` object for storing or viewing waveform related data in piece wise linear (PWL) format. However, power across an interval is constant, therefore, PPV stores and plots waveform using the `PiecewiseConstantWaveform` data type, as shown in the following example for 3 intervals:

```
PiecewiseConstantWaveform(x_values=[0.0, 9.9999993922529e-09,
1.99999987845058e-08, 2.999999892949745e-08], y_values=[0.08914847671985626,
1.1099846363067627, 0.09164756536483765, 0.09164756536483765], title='PPV3 Total
Power', label_x='time (s)', label_y='power (W)')
```

The `ppv.get_power_waveform` function returns plottable data in `PiecewiseConstantWaveform` type. This function takes `instance`, `power_net`, `clock_name`, `power_category`, `title`, and `time_offset` as inputs.

Usage example:

```
ppv_pwc_top_total = ppv.get_power_waveform(instance=Instance(''), title =
'Total Power TOP')
```

Internal power of the hier1 block:

```
ppv_pwc_hier2_internal = ppv.get_power_waveform(instance=Instance('hier2')],  
power_category = 'internal', title='Internal Power HIER1')
```

Power from all instances, when clock source reaches `clock_3C`:

```
ppv_pwc_clock3C = ppv.get_power_waveform(instance=Instance(''), clock_name =  
'clock_3C', title = 'Clock 3C')
```

Power from all instances connected to VDDE power domain:

```
ppv_pwc_VDDE = ppv.get_power_waveform(instance=Instance(''), power_net =  
Net('VDDE'), title = 'Domain VDDE')
```

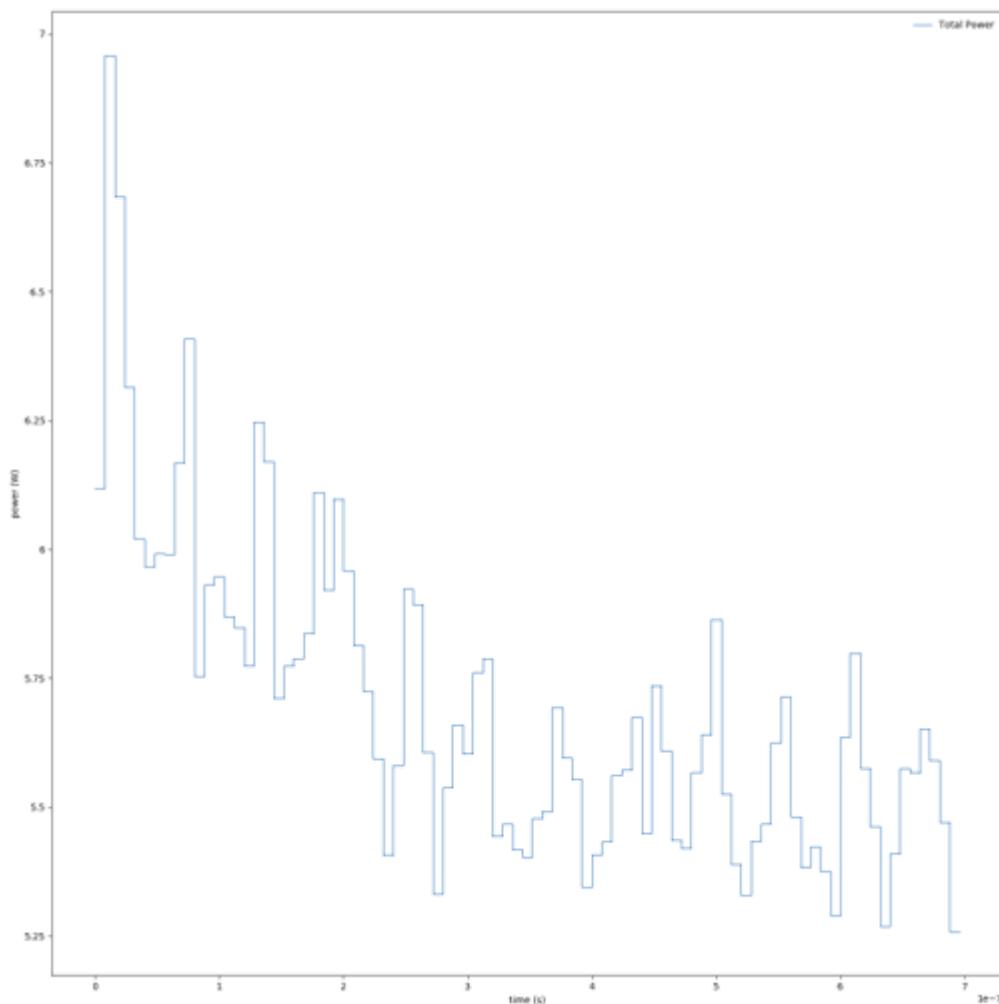
Switching power for a leaf instance:

```
ppv_pwc_inst1_switching = ppv.get_power_waveform(instance=Instance('inst1')),  
power_category = 'switching',  
title = 'Switching Power INST1')
```

Combination or exclusion can be done easily over the `PiecewiseConstantWaveform` object. To plot the power of top design with exclusion of two blocks (`hier1`, `hier2`), use the following commands:

```
>>> ppv_pwc_top_total = ppv.get_power_waveform(instances=Instance(''))  
>>> ppv_pwc_hier1_total = ppv.get_power_waveform(instance= Instance('hier1'))]  
>>> ppv_pwc_hier2_total = ppv.get_power_waveform(instance= Instance('hier2'),)  
>>> plot([ppv_pwc_top_total-ppv_pwc_hier1_total-ppv_pwc_hier2_total])
```

Simple subtraction is done here and it results in the following waveform:

Figure 75. Power Profile of Top Design Excluding Two Blocks

Similarly, additions or subtractions can be done to view power waveforms from two power domains added together, or addition from specific instances. You can also plot the result directly rather than storing to a variable before plotting by using the `plot(ppv.get_power_waveform(instance=Instance(''), power_net=Net('VDD'), power_category='total'))` command.

Time Offset

VCV or PPV always start from 0 irrespective of the time in VCD. It is easy to represent time when more than one VCD inputs are present. For example, events from 1.1 -1.2 μ s in VCD are 0-100 ns in VCV or PPV. When visualizing power over time, the event is from 0-100 ns. To see the waveform in the original format, use `time_offset` key. The `time_offset` of 1.1 μ s, plots the waveform with events corresponding to their actual VCD time stamps.

```
plot(ppv.get_power_waveform(instance=Instance(''), time_offset=1.1e-06))
```

14.3. Utilities for Vector Window Selection

PPV takes extra redhawk_sc_apa licenses, but integrated command performs vector selection without the need of extra license. For flows that use PPV as such, SeaScape has some utilities to find out VCD slices with high power or high power change. See [Integrated Command for Vector Selection](#) on page 425 for more details.

Vector Selection Utilities

There are two utilities for vector selection based on the power domains. Utilities return more than one window with N number of slices:

- `find_n_peak_power_slices`: Selects worst N slices from scope results based on peak power
- `find_n_power_change_slices`: Selects worst N slices from scope results based on power change

Usage example of vector selection:

```
import vector_selection
>>> help(vector_selection)
FUNCTIONS
find_n_peak_power_slices(scope_results, intervals, slice_num_intervals,
max_output_slices=1,
allow_overlapping_slices=True, start_time=0.0)
find_n_power_change_slices(scope_results, intervals, max_output_slices=1,
start_time=0.0)
```

For inputs of `find_n_peak_power_slices()` utility, use `help(vector_selection.find_n_peak_power_slices)` command and for inputs of `find_n_power_change_slices()` utility, use `help(vector_selection.find_n_power_change_slices)` command at the tool command prompt. For more details, see [Overlapping and Non-Overlapping Slices](#) on page 426.

VCD Slices With Highest Change in Power

The process is similar to integrated command with `selection_criteria` as `power_change`, see [Selection Criteria](#) on page 426. PPV creation and post processing script are done separately here. There is flexibility to get more than 10 high power slices. This can be done for block and a subset of time duration given to PPV. Example of utility:

```
>>> import vector_selection
>>> vector_selection.find_n_power_change_slice(scope_results =
ppv.get_results()['top'],
intervals=ppv.get_results()['intervals'], slice_num_intervals=2,
start_time=100e-09)
```

High change in power (`power_change`) is considered by this utility, rather than high power (`peak_power`).

Example of peak window from block level data and output of the function:

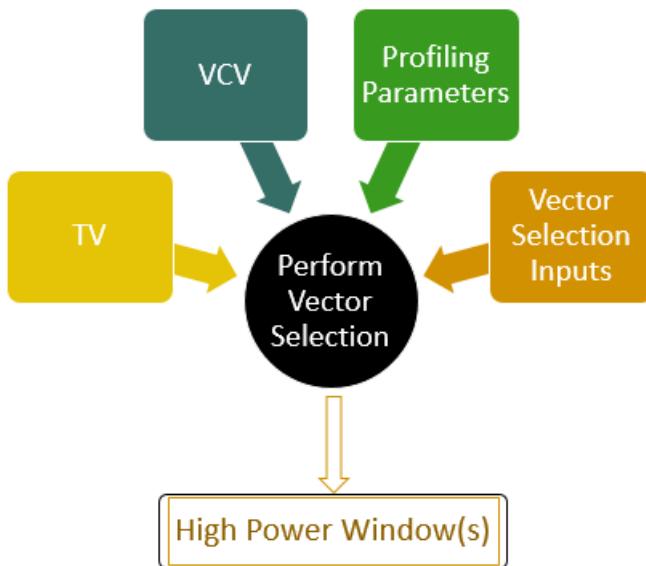
```
>>> ppv_results = ppv.get_results()
>>> import vector_selection
>>> vector_selection.find_n_peak_power_slices(
    scope_results=ppv_results['instance'][Instance('hier2')],
    intervals=ppv_results['intervals'],
    slice_num_intervals=5,
    max_output_slices=3,
    allow_overlapping_slices=False, )
```

```
[{'start_time': 2.055999857475399e-06, 'end_time': 2.0959998892067233e-06, 'average_power': 0.5001076310873032, 'power_per_interval': [0.5140145570039749, 0.5099677741527557, 0.4947694092988968, 0.4823756217956543, 0.49941079318523407]}, {'start_time': 1.7840000055002747e-06, 'end_time': 1.8239999235447613e-06, 'average_power': 0.49586965441703795, 'power_per_interval': [0.5129705518484116, 0.5027259439229965, 0.5000629276037216, 0.4783409535884857, 0.4852478951215744]}, {'start_time': 2.3199999077405664e-07, 'end_time': 2.719999940836715e-07, 'average_power': 0.4937522619962692, 'power_per_interval': [0.4761703610420227, 0.4937312453985214, 0.499926820397377, 0.5105480551719666, 0.48838482797145844]}]
```

Output format is same as that from `perform_vector_selection` command with `selection_criteria` as `power_change`.

14.4. Integrated Command for Vector Selection

SeaScape provides an integrated command, `<SeaScapeDB>.perform_vector_selection()`, which inputs VCD file and outputs only the top N durations with highest power or highest change in power. The command runs PPV under the hood to retrieve these high power durations. The `perform_vector_selection()` command can run with the basic license token used for setting up the RedHawk-SC. It does not return any output such as, block level power, power over time waveform, power data per interval, or per block control.

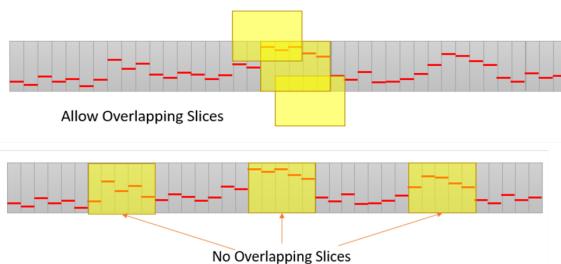


Integrated Command Inputs

`<SeaScapeDB>.perform_vector_selection()` command takes some extra inputs such as `slice_num_intervals`, `max_output_slices`, `allow_overlapping_slices`, `selection_criteria`, and `start_time` for window selection. For more details on the arguments, use `help(SeaScapeDB.perform_vector_selection)` command at the tool command prompt.

Overlapping and Non-Overlapping Slices

The default value of `allow_overlapping_slices` as `True`, allows overlapping of output slices. When it is set to `False`, the slices do not overlap. For example, if highest power window is 20-40 ns and second highest is 24-44 ns, the tool does not report 24-44 ns VCD slice as it overlaps with the highest power slice.



Selection Criteria

You can choose the slices using `peak_power` and `power_change` keys of the `selection_criteria` argument. `peak_power` selects the slices based on the average power and highest average power. The moving average is used to find out the slice with highest power.

Figure 76. Selection Criteria: Peak Power

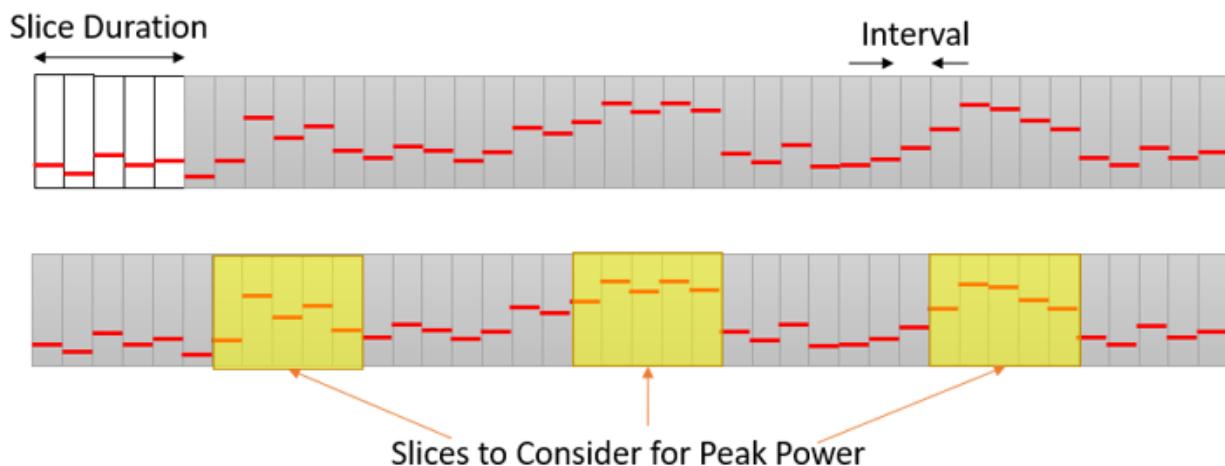
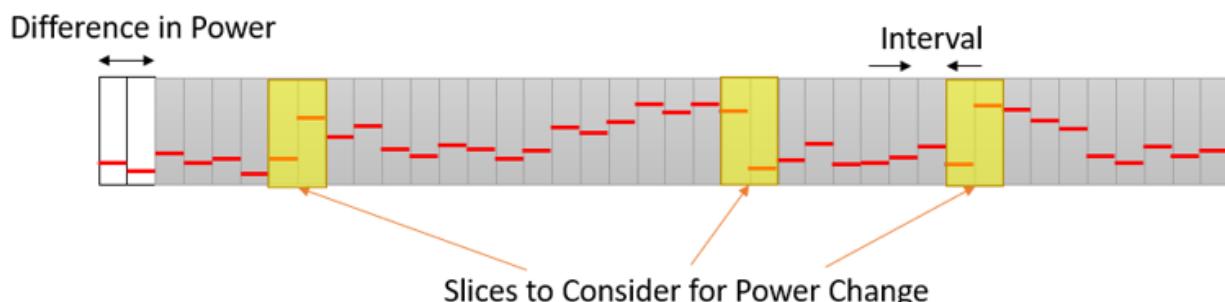


Figure 77. Selection Criteria: Power Change



The `power_change` key helps in finding power differences between adjacent intervals and returns highest increase and decrease in power. The `slice_num_intervals` input is invalid in this mode as `power_change`

always returns a window of length $2 * \text{time_interval_length}$. The slice reported by `power_change` cannot be used directly for dynamic analysis, therefore, perform post processing by keeping adjacent intervals into middle of the duration or end of the duration. For example, if interval length is 4 ns, then VCD slice reported by the command is 108 ns to 116 ns. If desired dynamic analysis duration is 40 ns, then 92 ns to 132 ns is taken for dynamic analysis.

Start Time

Start time is the time at which the output's time is shifted. The behaviour is similar to `time_offset`. The default value is 0. If `start_time` is given as 13e-06, VCD slice of 400-440e-09 becomes 13.400-13.440e-06.

Usage examples:

When `max_output_slices=4`, it returns 4 high power windows.

```
object_settings_ppv = {'design_values' : {'vcf_mode' : 'true_time'}}
pvs_settings={'pvt': {'voltage_levels': {'VSS' : 0.00, 'VDD' : 0.91, }},
    'time_interval_length': 4e-09,
    'object_settings': object_settings_ppv}
perform_vector_selection_args = dict(
    scenario_duration=2e-06,
    slice_num_intervals=5,
    max_output_slices=4,
    allow_non_overlapping_slices=False,
    settings=pvs_settings,
    options=options)
value_change_data = [{'view' : vcv, 'slice_name' : 'gate_top_sdf_full'}]
high_power_windows_sdf_vcd45 = db.perform_vector_selection(timing_view=tv,
    external_parasitics=evx, value_change_data=value_change_data,
    **perform_vector_selection_args)
```

When selection criteria is set to `power_change`.

```
object_settings_ppv={'design_values': {'vcf_mode': 'non_true_time'}}
pvs_settings={'pvt': {'voltage_levels': {'VSS': 0.00, 'VDD': 0.82, 'VDDC': 0.82}},
    'time_interval_length': 500e-12,
    'object_settings': object_settings_ppv}
perform_vector_selection_args = dict(
    scenario_duration=403e-09,
    selection_criteria='power_change',
    max_output_slices=6,
    settings=pvs_settings,
    options=options)
value_change_data = [{"view' : vcv, 'slice_name' : 'gate_top_unit_delay_full'}]

high_dpdt_windows_gate_top_unit_delay_vcd =
db.perform_vector_selection(timing_view=tv,
    external_parasitics=evx, value_change_data=value_change_data,
    **perform_vector_selection_args)
```

Integrated Command Outputs

The output from `<SeaScapeDB>.perform_vector_selection()` command is a list of high power windows. The `peak_power` selection criteria has keys such as, `start_time`, `end_time`, `average_power`, and `power_per_interval`.

When slice_num_intervals=2, max_output_slices=3:

```
[{'average_power': 8.296e-06, 'start_time': 3e-09, 'end_time': 5e-09,
'power_per_interval': [7.774e-06, 8.819e-06],},
{'average_power': 5.512e-06, 'start_time': 1e-09, 'end_time': 3e-09,
'power_per_interval': [1.026e-05, 7.645e-07],},
{'average_power': 8.83e-07, 'start_time': 5e-09, 'end_time': 7e-09,
'power_per_interval': [8.83e-07, 8.83e-07],}]
```

The keys for power_change are different. Each dict has max and min keys which correspond to high increase and decrease in power. Each entry in max and min has keys such as, start_time, end_time, power_change, and power_per_interval.

When slice_num_intervals=2 max_output_slices=2 selection_criteria='power_change':

```
{'max':
[{'start_time': 0.0, 'end_time': 2.0e-09, 'power_change': 9.53e-06,
'power_per_interval': [7.24e-07, 1.02e-05],},
{'start_time': 2.0e-09, 'end_time': 4.0e-09, 'power_change': 7.00e-06,
'power_per_interval': [7.64e-07, 7.77e-06],}],
'min':
[{'start_time': 1.0e-09, 'end_time': 3.0e-09, 'power_change': -9.49e-06,
'power_per_interval': [1.02e-05, 7.64e-07],},
{'start_time': 4.0e-09, 'end_time': 6.0e-09, 'power_change': -7.93e-06,
'power_per_interval': [8.81e-06, 8.82e-07],},
```

To create ScenarioView from vector selection process:

```
vcd_files = [
    {'file_name' : design_data_path + '/fsdb/top_Galaxy_2.5us_gate_vcd.fsdb',
     'instances' : [],
     'time_slices' : [{'slice_name' : 'top_delay_full'}],
     'preamble' : 'top',
   }]
vcv = db.create_value_change_view(design_view=dv, vcd_files=vcd_files,
tag='vcv_full', options=options)
value_change_data_full = [
    {'view' : vcv, 'slice_name' : 'top_delay_full'}]
object_settings = {
    'design_values' : {'gate_vcd_mode' : 'time_based'}}
high_power_windows = db.perform_vector_selection(
    timing_view=tv, external_parasitics=evx,
    value_change_data=value_change_data_full,
    voltage_levels=voltage_levels,
    object_settings=object_settings,
    scenario_duration=2.5e-06,
    time_interval_length=8e-09,
    slice_num_intervals=5,
    max_output_slices=6,
    allow_overlapping_slices=False,
    start_time=vcd_start_time
    options=options)
start_time = high_power_windows[0]['start_time']
end_time = high_power_windows[0]['end_time']

vcd_files_high_power = [
    {'file_name' : design_data_path + '/fsdb/top_Galaxy_2.5us_gate_vcd.fsdb',
```

```
'instances' : [],
'preamble' : 'top',
'time_slices' : [ {'slice_name' : 'high_power_window_vcd',
                  'start_time' : start_time, 'stop_time' : end_time} ] })
vcv_high_power= db.create_value_change_view(design_view=dv,
                                             vcd_files=vcd_files_high_power,
                                             tag='vcv', options=options)
object_settings = {
    'design_values' : { 'gate_vcd_mode' : 'time_based' } }
value_change_data_high_power_window = [
    { 'view' : vcv_high_power, 'slice_name' : 'high_power_window_vcd' }]
scn_npv_vcd = db.create_no_prop_scenario_view(
    timing_view=tv, external_parasitics=evx, extract_view=ev,
    value_change_data=value_change_data_high_power_window,
    voltage_levels=voltage_levels,
    scenario_duration=40e-09,
    object_settings=object_settings_npv_vcd,
    default_clock={'policy':'custom', 'period':8e-9},
    tag='scn_npv_vcd',
    options=options)
```

15: Using the RedHawk-SC Infrastructure

The RedHawk-SC tool uses the SeaScape framework or infrastructure to perform various tasks. The RedHawk-SC infrastructure broadly constitutes the following topics:

- [Managing Workload](#) on page 431
 - [RedHawk-SC Scheduler GUI](#) on page 436
 - [Managing Disk Space](#) on page 451
 - [Resiliency](#) on page 459
-

15.1. Managing Workload

The RedHawk-SC tool automatically distributes complex tasks for import, extraction, logic propagation, and simulation of complex integrated circuit designs. To access this distributed analysis system, you need to provide the syntax to request a CPU resource with enough memory. The tool parses this syntax to launch processes called workers, in the compute farm.

The tool uses the workers for distributed processing. Both the master process (that you directly invoke) and the worker processes use the same RedHawk-SC executable. The tool supports creating launchers for most of the popular compute grids. It can also create launchers based on SSH protocol. In addition, you can define the launcher process for any custom grid.

The following topic describes methods to create and launch workers:

- [Creating and Launching Workers](#) on page 431
-

15.1.1. Creating and Launching Workers

The following topics describe the various methods to create and launch workers.

- [Creating Launcher Objects](#) on page 431
- [Defining Custom Launchers](#) on page 432
- [Automatically Launching Workers](#) on page 433
- [Specifying Number of Workers to Launch](#) on page 434
- [Registering Multiple Default Launchers](#) on page 434
- [Customizing Launchers for Specific Jobs](#) on page 435
- [Reusing Default Setting to Create Launchers](#) on page 436

Creating Launcher Objects

The `create_grid_launcher` command is a general command to define launchers for any custom grid depending on the argument that you provide.

The following example creates a launcher for an LSF farm by using the `create_grid_launcher` command. Processors are allocated on the same host with each job consuming 16GB and using the RHEL7 platform. The `user_label` argument is used to provide an identification string for the launcher.

```
bsub_command = 'bsub -q queue_name -R rhel7
                -R "rusage[mem=16000]" -R "span[hosts=1]" -n 1'
ll = create_grid_launcher('user_label', bsub_command)
```

The following example creates a launcher for a UGE farm by using the `create_grid_launcher` command.

```
qsub_command = "qsub -V -cwd -j y -b y -o run.out  
              -q all_hosts -l mfree=16G -l platform='linux64e7'"  
ll = create_grid_launcher('user_label', qsub_command)
```

Similarly, the `create_grid_launcher` command can create launchers for the RTDA NC farm.

You can list the number of jobs to be launched per worker by specifying the `num_workers_per_launch` argument. If you do not explicitly define `num_workers_per_launch`, the tool automatically detects the number of workers per launch from the submit command specified with the `create_grid_launcher` command.

You can also create SSH-based launchers. In an SSH-based launcher, the workers are directly launched on the execution hosts with SSH protocol instead of relying on the compute cluster to manage and launch jobs.

```
ll = create_ssh_launcher('my_ssh_launcher',  
                       ['host1', 'host2', 'host3', 'host3'])
```

You can also launch workers on the same host as the master with the `create_local_launcher` command. You can use local launchers to create workers required for the jobs when you have access to machines with large storage capacities.

For a function to execute at the beginning while creating a launcher, specify the setting by using the `initial_exec_function` argument. For example,

```
def worker_startup_func():  
    #script-commands to be executed at the start of workers  
    gp_print('New worker launched')  
  
ll_special = create_local_launcher("local",  
                                   initial_exec_function=worker_startup_func)  
ll_special.launch(10)
```

Defining Custom Launchers

The RedHawk-SC tool can create workers in different Unix environments. Each worker is a stand-alone process. You can use the following launcher commands to define how to create workers:

- `create_local_launcher`
- `create_ssh_launcher`
- `create_grid_launcher`

For example,

```
ll = create_grid_launcher('my_launcher', 'my_launch_command -my_options')  
register_default_launcher(ll)
```

If multiple CPUs are required, use the `num_workers_per_launcher` argument as shown in the following example.

```
ll = create_grid_launcher('my_launcher', 'my_launch_command -my_options',  
                        num_workers_per_launcher=2)  
register_default_launcher(ll)
```

You must be sure that multiple CPUs will be consumed in the jobs launched with this launcher.

Automatically Launching Workers

It might be difficult to manually determine the number of workers to be launched to process a design. The tool optimally partitions allocated jobs and can automatically decide the number of workers.

The `register_default_launcher` command informs the system of the launcher object to be used when a new worker is needed. By registering a created launcher as a default launcher, the tool can start workers on demand. In a design, each view creation process requires an optimal number of workers.

You have the flexibility to limit the total number of workers launched, start with a minimum number of workers, delay the view creation until a certain percentage of optimal workers are available.

The following example creates a UGE launcher object, `ll`, and uses it as the default launcher. When any view creation requires some workers, it uses the default launcher to submit a request to the grid to make the workers available. Though the tool can begin the analysis as soon as at least one worker is online, to have a minimum of 16 workers before the run can begin, the `min_num_workers` argument is set to 16.

The system is also capped at a maximum of 600 workers by using the `max_num_workers=600`, that is, the tool can execute a maximum of 600 jobs at any given time. When a greater number of jobs need to be run, the tool waits until one of the jobs finishes to begin executing its next scheduled jobs.

```
qsub_command = "qsub -V -cwd -j y -b y -o run.out
                -q all_hosts -l mfree=16G -l platform='linux64e7'"
ll = create_grid_launcher('user_label', qsub_command)
register_default_launcher(ll, min_num_workers=16, max_num_workers=600)
```

By default, the tool waits for at least one worker to come online. To abort a run when no workers become available even after waiting for a specified time, use the `time_out` argument. In the following example, the run aborts if no workers are available even after the specified time out period of 3600 seconds (one hour).

```
qsub_command = "qsub -V -cwd -j y -b y -o run.out
                -q all_hosts -l mfree=16G -l platform='linux64e7'"
ll = create_grid_launcher('user_label', qsub_command)
register_default_launcher(ll, min_num_workers=16,
                        max_num_workers=100, time_out=3600)
```

In the following example, the tool waits only for 300 seconds for the minimum number of workers to be available before proceeding with the available number of workers.

```
qsub_command = "qsub -V -cwd -j y -b y -o run.out
                -q all_hosts -l mfree=16G -l platform='linux64e7'"
ll = create_grid_launcher('user_label', qsub_command)
register_default_launcher(ll, min_num_workers=16, max_num_workers=100,
                        wait_for_workers_time_out=300)
```

To wait for a percentage of requested workers for a view to be available before view creation, use the `wait_for_workers` argument. This improves performance and peak worker memory consumption. The argument takes in a dictionary of the view class and the corresponding percentage of workers to be available. In the following example, the tool waits for at least 70% of the requested number of workers to be available before starting `AnalysisView` creation.

```
qsub_command = "qsub -V -cwd -j y -b y -o run.out
                -q all_hosts -l mfree=16G -l platform='linux64e7'"
ll = create_grid_launcher('user_label', qsub_command)
register_default_launcher(ll, min_num_workers=16, max_num_workers=100,
                        wait_for_workers={AnalysisView:0.7})
```

In the following example, the tool waits for the availability of 50% of optimal workers to start `ScenarioView` creation, and 70% of optimal workers to start `AnalysisView` creation. However, the tool waits only for 360 seconds before proceeding with the available number of workers for both the views.

```
qsub_command = "qsub -V -cwd -j y -b y -o run.out
                -q all_hosts -l mfree=16G -l platform='linux64e7'"
ll = create_grid_launcher('user_label', qsub_command)
register_default_launcher(ll, min_num_workers=16, max_num_workers=100,
                           wait_for_workers={AnalysisView:0.7, ScenarioView:0.5},
                           wait_for_workers_time_out=360)
```

Specifying Number of Workers to Launch

To specify the number of workers to launch, use the `launch` command. You can use this command with `register_default_launcher` as shown in the following example:

```
qsub_command = "qsub -V -cwd -j y -b y -o run.out
                -q all_hosts -l mfree=16G -l platform='linux64e7'"
ll = create_grid_launcher('user_label', qsub_command)
ll.launch(16)
register_default_launcher(ll, max_num_workers=100)
```

Multithreading Support

The RedHawk-SC transient simulation solve uses multiple threads. To specify the number of threads, use the `max_num_threads_per_worker` argument with the `create_grid_launcher` command as follows:

```
ll = create_grid_launcher('name', command, max_num_threads_per_worker=N)
register_default_launcher(ll)
```

Registering Multiple Default Launchers

The tool uses default launchers to access a new worker. You might need to have multiple default launchers, for example, you might need both an SSH launcher and a farm launcher and the system should fill up the SSH launcher before moving to the farm launcher. You can enable this through the `launcher_func` argument of the `register_default_launcher` command. This argument takes in a function that can accept the name (`tag`) of the launcher last used to launch workers and an optional number of workers being requested to launch. The function should return the launcher object to be used for launching the next worker. When the tool requires a new worker, it uses the `launcher_func` argument to resolve the launcher object to be used.

The following script example defines the function to use when a new launcher is required. Three launcher objects are created. `launcher1` is a local launcher and both `launcher2` and `launcher3` are LSF Launchers. The specified `launcher_func` argument checks the name of the launcher used to launch the last worker. Then, it ensures that the workers are launched in the ascending order of `launcher1`, `launcher2`, and `launcher3`.

```
launcher1 = gp.create_local_launcher('local1')
launcher2 = gp.create_grid_launcher('l2', 'bsub ....')
launcher3 = gp.create_grid_launcher('l3', 'bsub .... -q big')

def my_launcher_func(last_launcher_name):
    if last_launcher_name == 'local1':
        return launcher2
    elif last_launcher_name == 'l2':
```

```

        return launcher3
    else:
        return launcher1

register_default_launcher(launcher_func=my_launcher_func)

```

The following script example defines three SGE launchers and specifies the maximum number of workers that can be launched with each launcher. The `launcher_func` argument collects all the available launchers that be used based on the number of workers already launched by each launcher. It then returns the first launcher that is different from the previously used launcher, falling back to `launcher3` in case of any errors.

```

launcher1 = gp.create_grid_launcher('sge1', 'qsub -q queue1')
launcher2 = gp.create_grid_launcher('sge2', 'qsub -q queue2')
launcher3 = gp.create_grid_launcher('sge3', 'qsub -q queue3')
launchers_with_limits = [(launcher1, 20), (launcher2, 10), (launcher3, 60)]

def my_launcher_func(launcher_name):
    available_launchers = [ll for ll, limit in launchers_with_limits if
                           ll.get_num_launched() < limit]

    ll = None
    for ll in available_launchers:
        if ll.get_name() != launcher_name:
            break
    if ll is None: # fail condition, in case max_num_workers is not set right
        gp_assert(0, 'fall back launcher used')
        ll = launcher3
    return ll

gp.register_default_launcher(launcher_func=my_launcher_func)

```

Customizing Launchers for Specific Jobs

Jobs with large memory consumption require specific workers to be launched with large memory resources. Regular workers can be used for execution of other jobs. A typical example is the chip power model (CPM) creation flow where the process of creating the CPM consumes large memory.

The following script example creates two launchers, one for regular jobs and another for jobs with large memory requirements. To direct the tool to use a particular launcher (including corresponding workers) to perform specific jobs, use the `set_jobs` command with the created launcher object. The `set_jobs` command takes in a list or regular expression patterns of assigned jobs.

Though workers from both `launcher` and `big_launcher` are available, the tool uses only the workers created with `big_launcher` for CPM-specific jobs (`cpm.write_spice_deck` and `cpm.run_asim_power_model*`). For all other scheduled jobs, all the available workers are used.

```

launcher = create_grid_launcher('launcher', 'qsub -V -b -cwd -j y
                                -q queue -l mfree=16G -o uge.log')
big_launcher = create_grid_launcher('big_launcher', 'qsub -V -b y -cwd -j y
                                    -q queue_perf -l mfree=250G -o uge_perf.log')
big_launcher.set_jobs(['cpm.write_spice_deck*', 'cpm.run_asim_power_model*'])
big_launcher.launch(1)
register_default_launcher(launcher)

```

To ensure that a launcher and corresponding workers perform only exclusive jobs and no other jobs, use the `set_exclusive_jobs` command. In the following example, workers created with the `ll_big` launcher are only used to perform jobs for the patterns specified with the `set_exclusive_jobs` command.

```
ll_big = create_grid_launcher('big_launcher', 'bsub -R <large_mem>')
ll_small = create_grid_launcher('small_launcher', bsub_command)
ll_big.set_exclusive_jobs(CMRegex('.*\d+abc')) ## regex interface
ll_big.launch(4)
register_default_launcher(ll_small)
```

Reusing Default Setting to Create Launchers

Instead of specifying the launcher type for every analysis, you can create a configuration file for the tool to use the default launcher configuration. For example, you can create a file called `~/.seascape_rc/launcher.config` with the following contents. The tool uses the specified UGE launcher each time it needs a new worker and when no default launcher is registered.

```
[{'launcher_name': 'uge_launcher',
 'launcher_command': 'qsub -l mfree=16G',
 'launcher_type': 'uge',
 'num_workers_per_launch': 1}]
```

The tool expects the following optional and mandatory keys for `~/.seascape_rc/launcher.config`

- Required keys:

```
(['launcher_name', 'launcher_type'])
```

- Optional keys:

```
(['min_number_workers', 'max_number_workers', 'worker_multiplier',
 'hosts', 'launcher_command', 'num_workers_per_launch'])
```

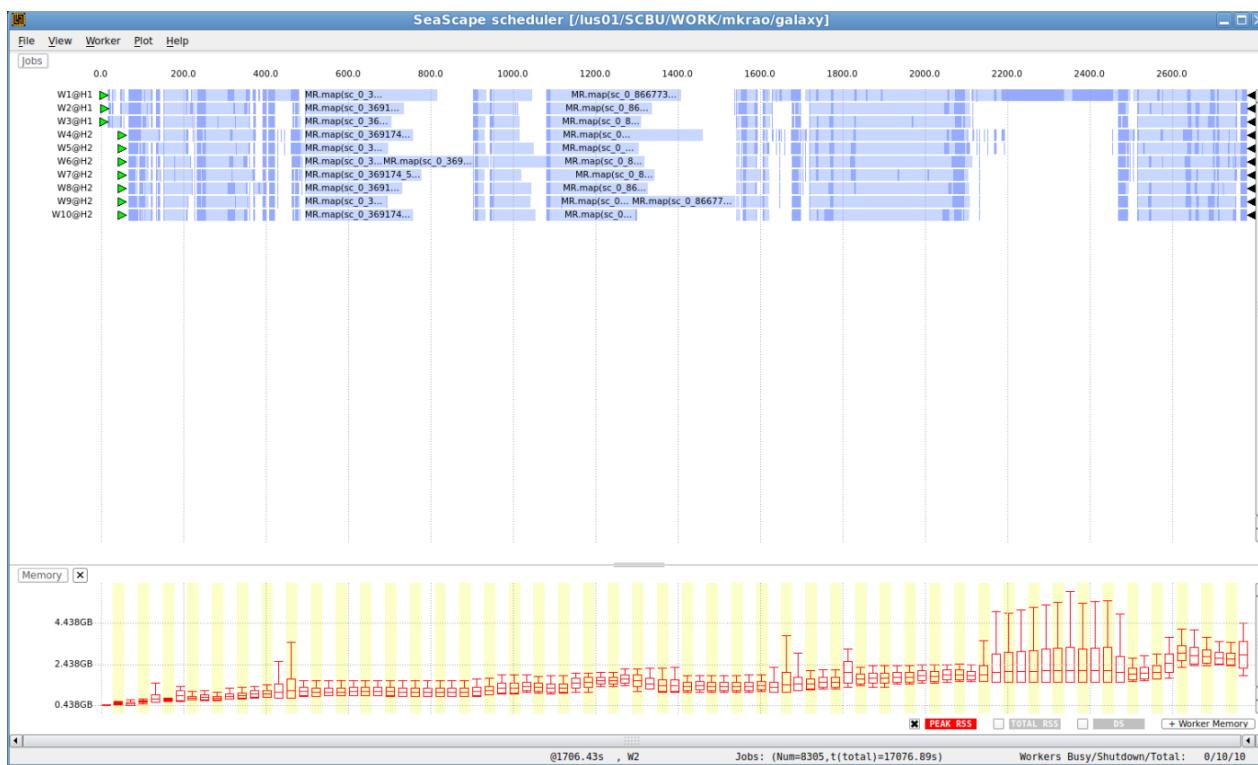
15.2. RedHawk-SC Scheduler GUI

The RedHawk-SC Scheduler GUI provides real-time visualization of workers and their jobs. It is useful to visualize and track the progress of jobs when multiple jobs run in parallel across multiple systems in a farm. The following topics describe how to work with the Scheduler-GUI.

- [Overview](#) on page 436
- [Invoking the Scheduler GUI](#) on page 441
- [Features of the Scheduler GUI](#) on page 442
- [Debugging With Scheduler](#) on page 448

15.2.1. Overview

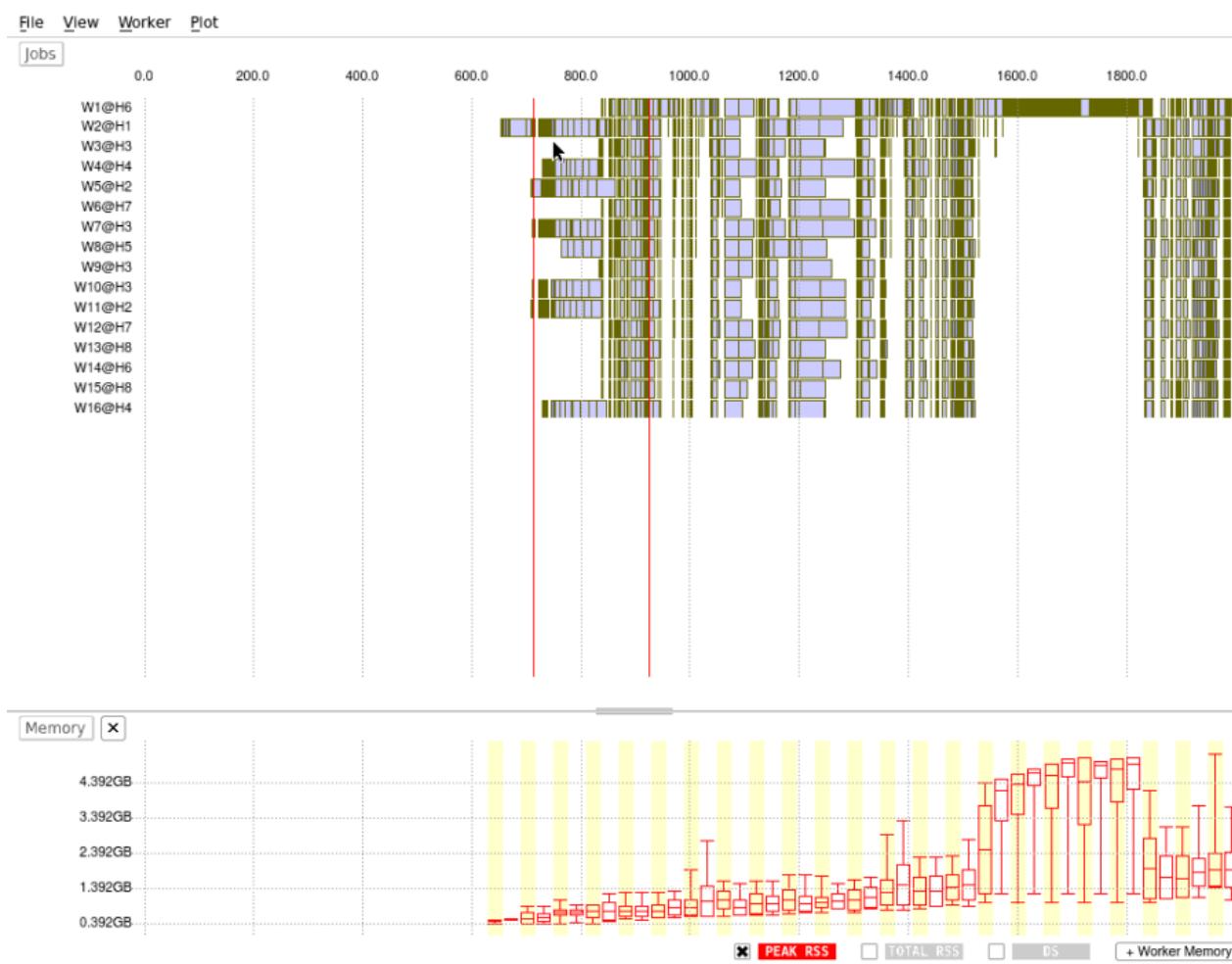
The Scheduler-GUI displays the jobs executed per worker on the y-axis and the time (in seconds) on the x-axis.



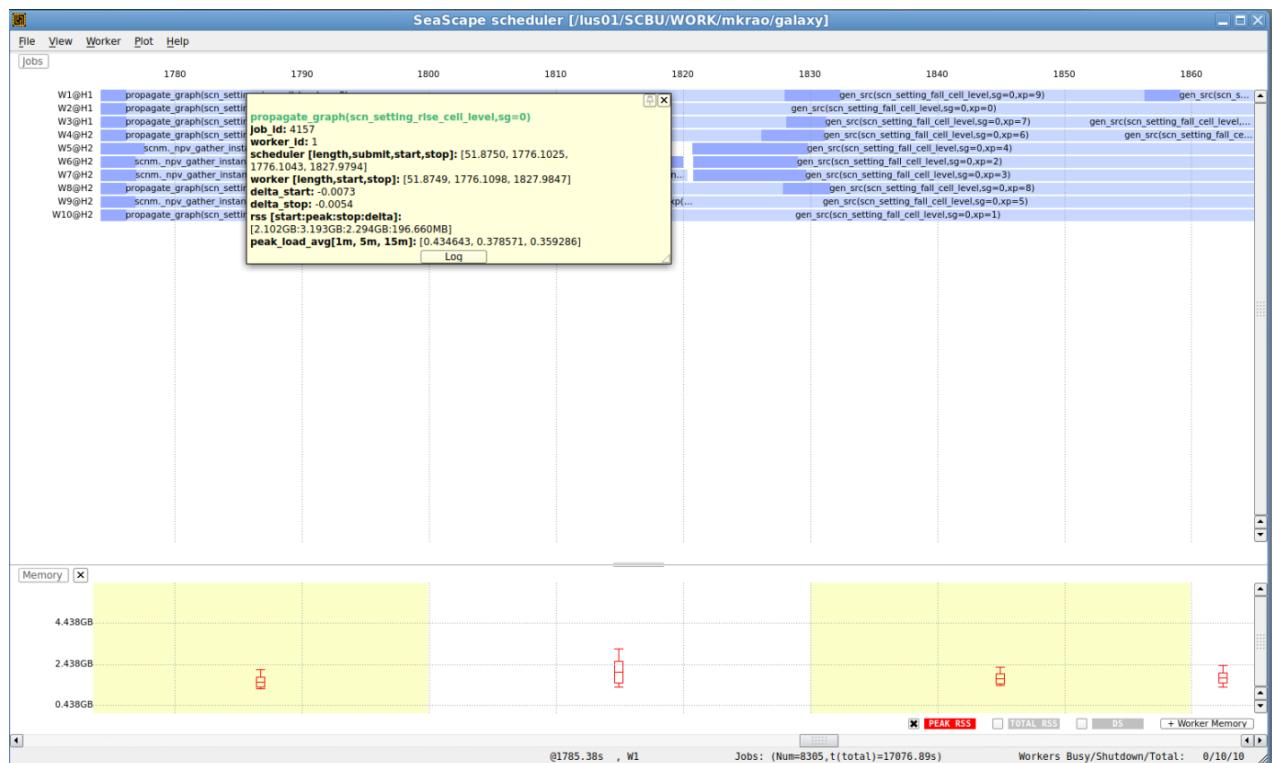
Each horizontal bar of the figure represents a worker launched by the tool. On the y-axis, each worker is marked with its worker id and host id, such as W4@H2.

To zoom in and zoom out the Scheduler display, use the **z** and **Z** keys respectively. To get the original **fit** view, use the **f** shortcut key.

You can also avail these options from the **View** menu. To zoom in using a mouse, right-click and horizontally drag the mouse on the Scheduler display. In the magnified view, use the horizontal bars to move across different time intervals. To scroll horizontally, use the left and the right arrow keys.

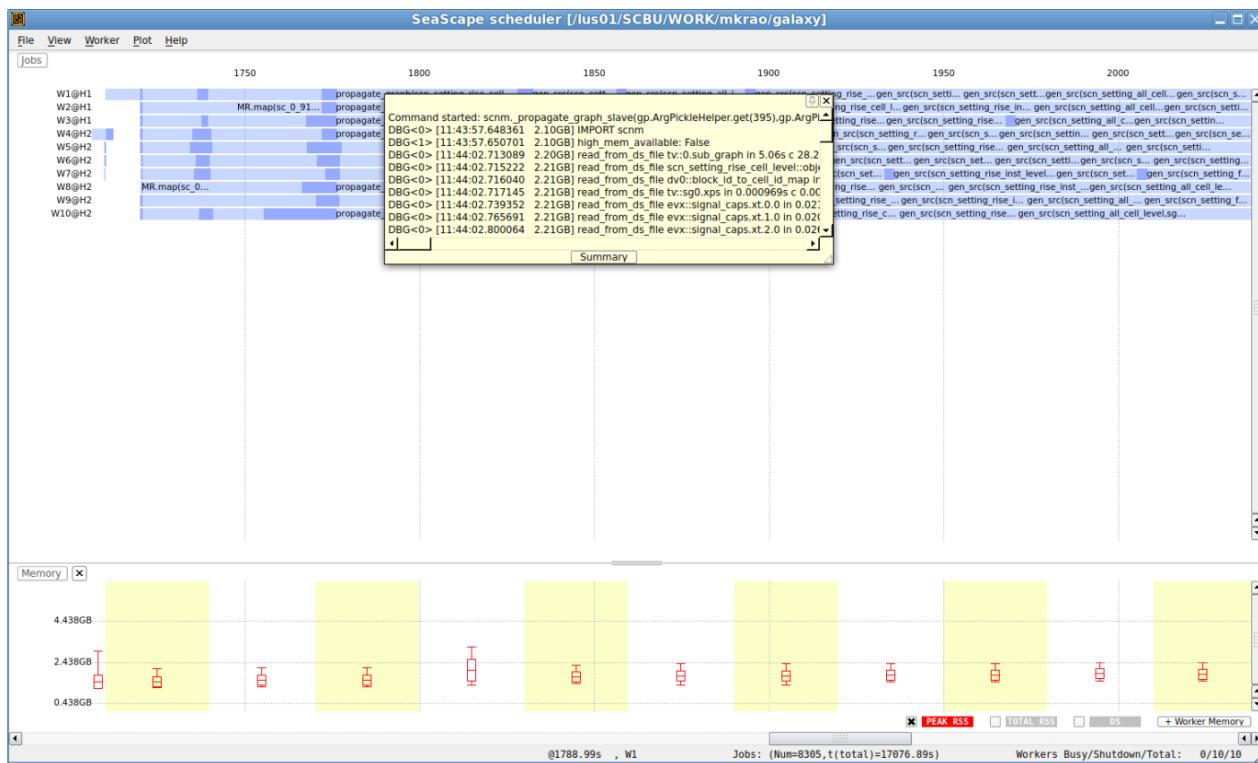


For more details of a job being executed, click the corresponding horizontal bar.



A yellow popup window appears with the following information:

- Job name (highlighted in green)
- Job Id
- Worker Id
- Job start time and stop time
- Total time taken for the job
- Job memory consumption details
- Average load of the execution host for the job duration in one, five, and fifteen minute intervals



By default, the Scheduler shows the job names. To view only the job outlines, use the **O** shortcut key. The following table shows the list of shortcut keys available for use in the Scheduler GUI.

Table 22: Shortcut Keys of the Scheduler GUI

Shortcut Key	Action
F	Zoom Fit
z	Zoom In
Z	Zoom Out
j	Show Job Name
s	Show Job Detail
o	Show Job Outline
i	Show Idle Jobs
t	Show Time Bar
h	Highlight Jobs
d	Remove Highlight of Jobs
Ctrl + Shift + s	Stop Updating Job Information

You can invoke and close the Scheduler GUI irrespective of the analysis session. This means that you can first open the Scheduler GUI and then connect to an ongoing analysis to gauge the progress, or scrutinize a completed run from its working directory.

15.2.2. Invoking the Scheduler GUI

In RedHawk-SC, every job to be executed is scheduled. The jobs are executed as and when the pre-requisite conditions for the jobs are satisfied. You can open multiple Scheduler GUI windows. Closing one or more Scheduler-GUI windows do not effect ongoing runs.

You can invoke the Scheduler-GUI in multiple ways.

From the RedHawk-SC Shell

To invoke the Scheduler GUI from the RedHawk-SC shell, use the following command:

```
gp.open_scheduler_window()
```

This command opens a Scheduler GUI window if the `DISPLAY` variable is correctly initialized.

By default, the title of the Scheduler GUI window has the analysis working directory name. You can change the default name. See `help(gp.open_scheduler_window)`.

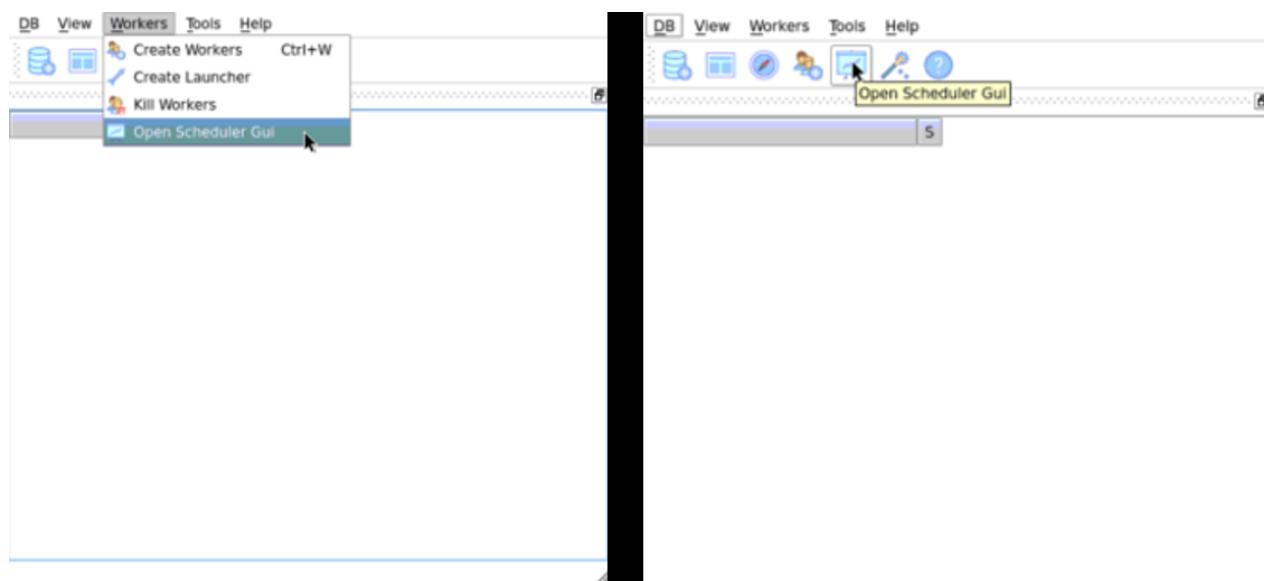
Similarly, you can configure the horizontal bars corresponding to each job to display the job name that it is currently executing .

To force open multiple Scheduler GUI windows, use the `force` argument with the `gp.open_scheduler_window()` command.

Opening the Scheduler GUI From Console

You can open the Scheduler GUI from the console window in one of the following ways:

- From the menu bar, click **Workers>Open Scheduler Gui**.
- OR
- From the tool bar, click the **Open Scheduler Gui** icon.



From a Completed or Live Session

To open the Scheduler GUI for an ongoing or completed run, use the `-g` option with the `redhawk_sc` command:

```
redhawk_sc -r /runs/redhawk_sc/galaxy/gp/ -g -l
```

The `-r` option specifies the path to the remote working directory to connect to. The tool uses the working directory (typically named gp) of a complete or ongoing run to visualize jobs.

The optional `-l` argument disables the creation of a new run directory.

This enables you to monitor a batch run, watch a live run from a different terminal, or profile completed runs.

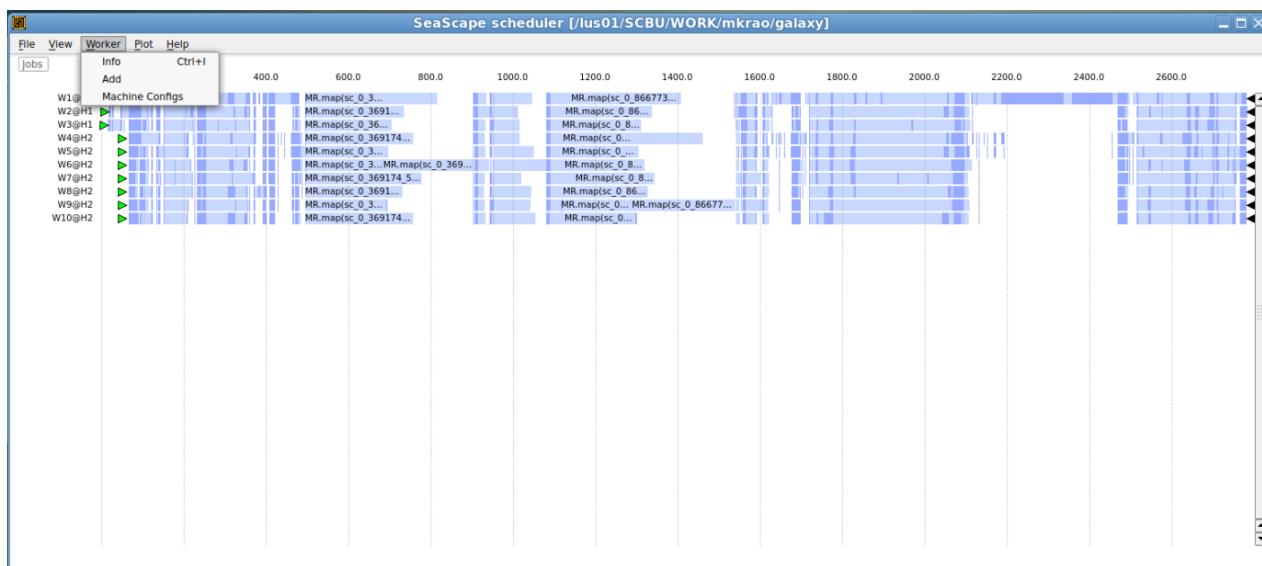
15.2.3. Features of the Scheduler GUI

The following features of the Scheduler GUI enable you to determine the status of the analysis and the machines that run the analysis.

Viewing Worker Information

The Scheduler can provide you with a quick summary of all the workers that execute RedHawk-SC jobs.

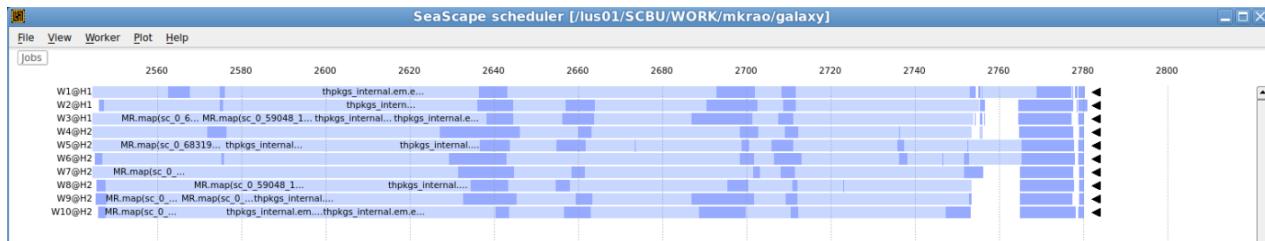
To view worker information summary, select **Worker>Info** from the menu bar or press **Ctrl + I** shortcut keys.



This opens a **Worker Info** window that reports a summary of the workers, execution hosts and some ancillary information.

Worker Info								
Name	Host	NOW RSS	NOW VIRT	NOW DS	PID	PEAK RSS		
W8@H2	cdc011m042 [H2]	2.156GB	6.200GB	1.175GB	192330	3.566GB		
W7@H2	cdc011m042 [H2]	2.008GB	6.101GB	966.000MB	192329	3.114GB		
W6@H2	cdc011m042 [H2]	2.272GB	6.209GB	1.086GB	192333	4.450GB		
W10@H2	cdc011m042 [H2]	2.144GB	7.448GB	1.060GB	192332	3.336GB		
W3@H1	cdc011m045 [H1]	1.962GB	6.743GB	998.000MB	84698	2.862GB		
W1@H1	cdc011m045 [H1]	3.043GB	8.954GB	1.810GB	84696	5.958GB		
W2@H1	cdc011m045 [H1]	2.244GB	5.883GB	1.122GB	84697	3.334GB		
W9@H2	cdc011m042 [H2]	1.921GB	6.059GB	959.000MB	192331	3.043GB		
W5@H2	cdc011m042 [H2]	2.240GB	7.098GB	1.109GB	192325	4.440GB		
W4@H2	cdc011m042 [H2]	2.203GB	6.022GB	1.076GB	192334	3.315GB		

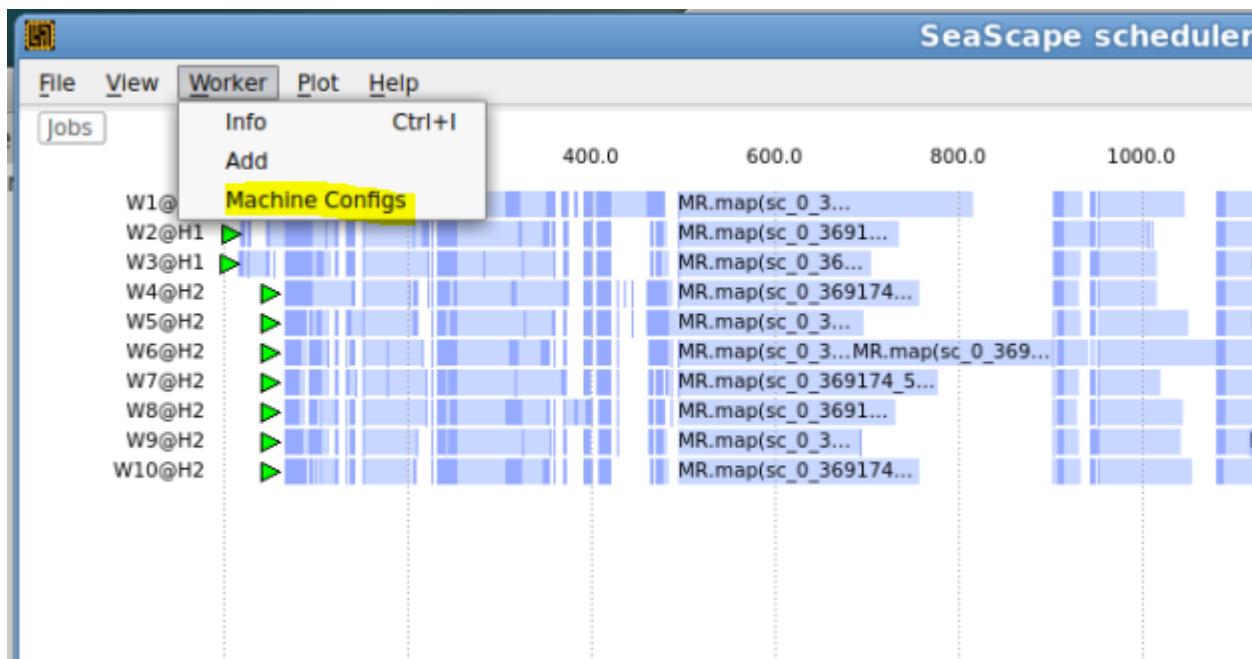
The Scheduler GUI displays visual indicators for the health of workers. When a run is completed or some of the workers are killed during the run (such as due to using the `shutdown_workers` command), the termination is indicated by a black left-facing triangle adjacent to the worker bar.



Viewing Host Information

To view host information summary, select **Worker>Machine Configs** from the menu bar.

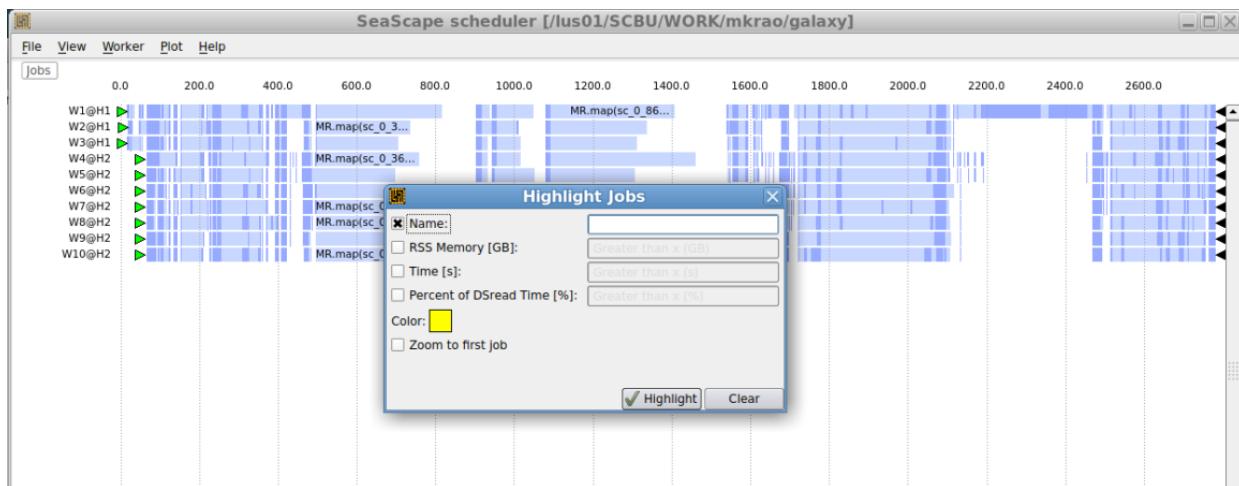
This opens a **Host Info** window that reports a tabular summary of processing parameters for each execution host.



Highlighting Jobs

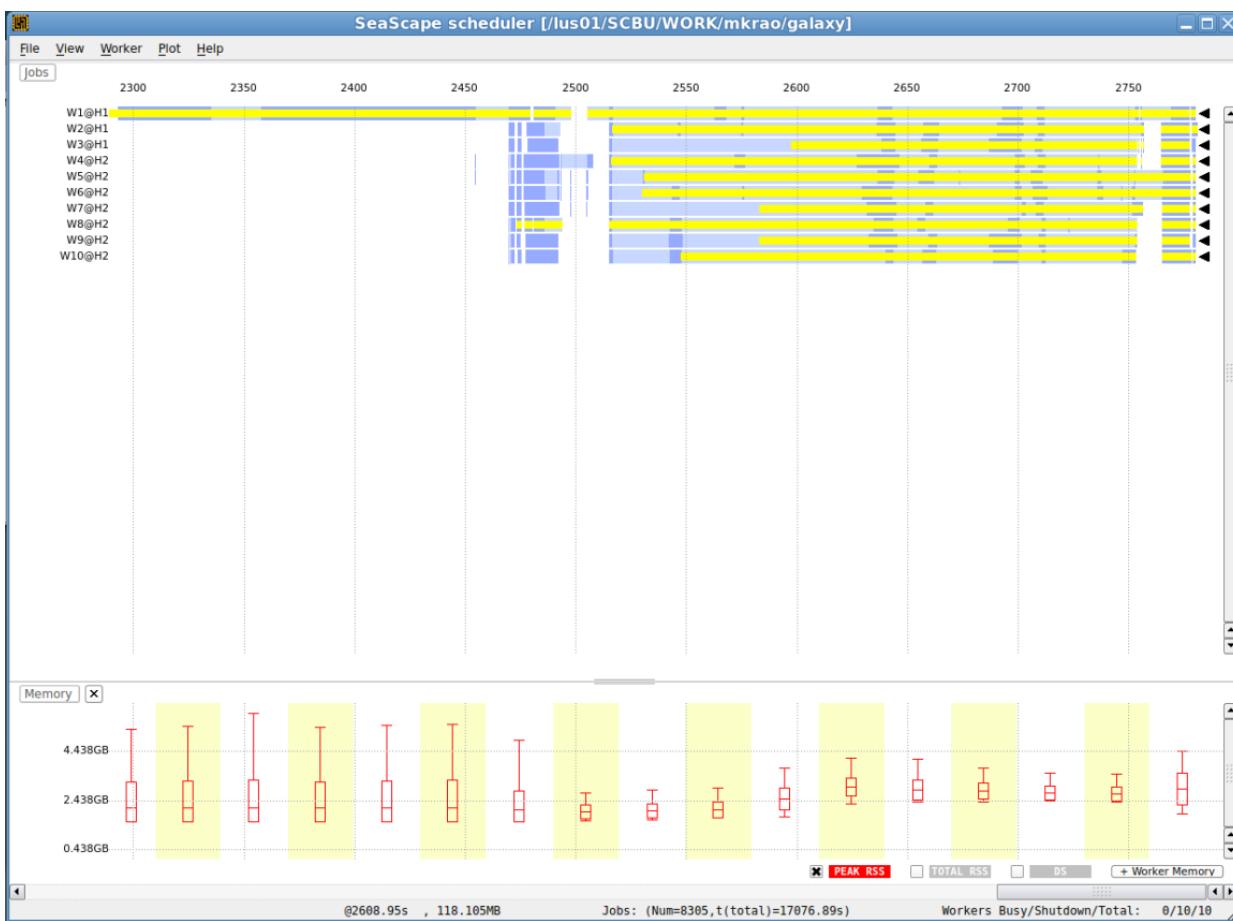
For easy identification, you can highlight jobs in the Scheduler GUI based on certain constraints.

- To highlight jobs, press the **h** shortcut key. This opens the **Highlight Jobs** window.



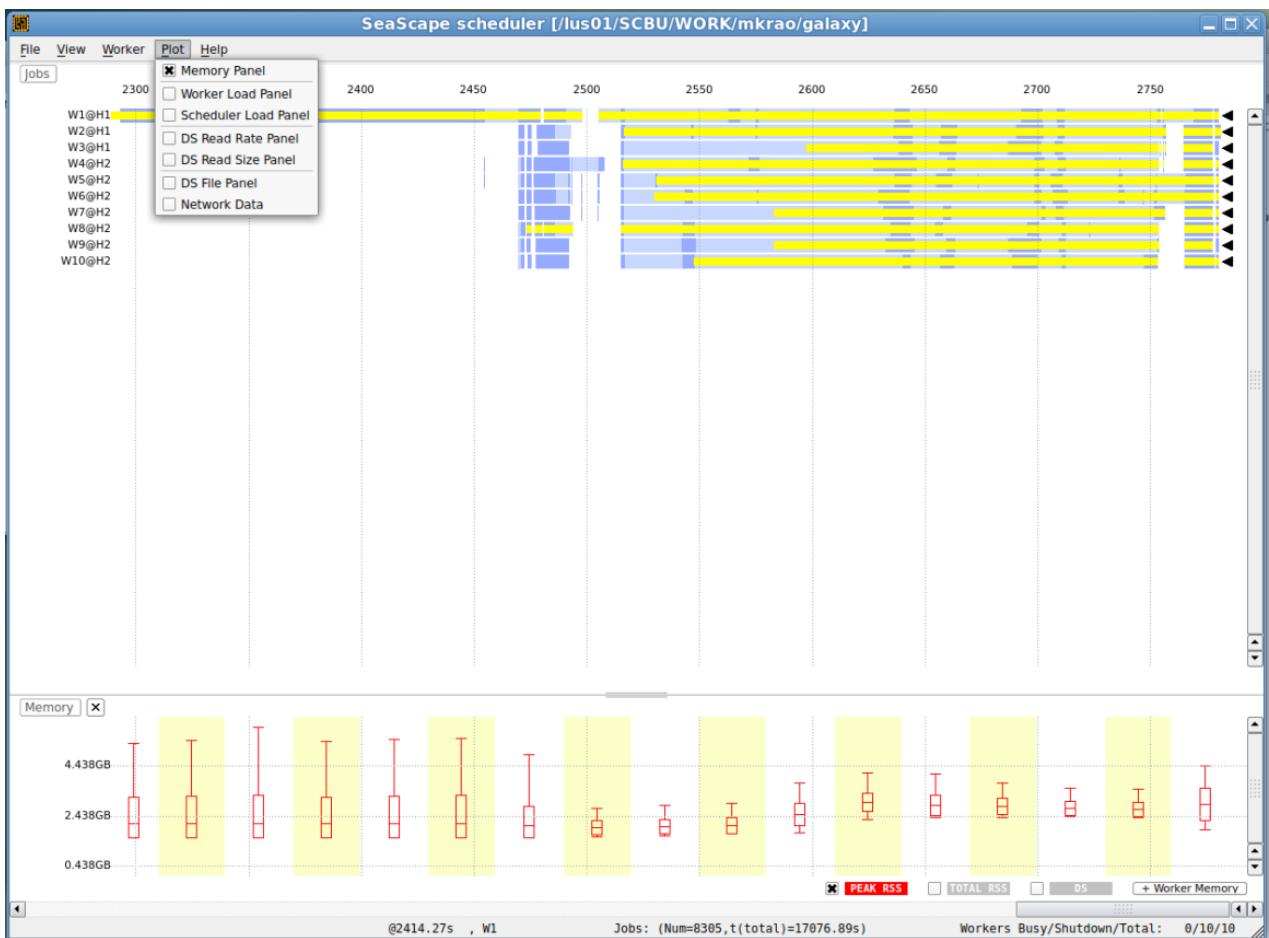
You can highlight jobs by name, threshold memory consumption, or the time when the job occurs. As the x-axis denotes time, the worker bar denotes the time point at which the job starts or stops. You can select the color to highlight jobs.

- To zoom into the first highlighted job when multiple jobs are highlighted, check the **Zoom to first job** box.



Worker Statistics and Plots

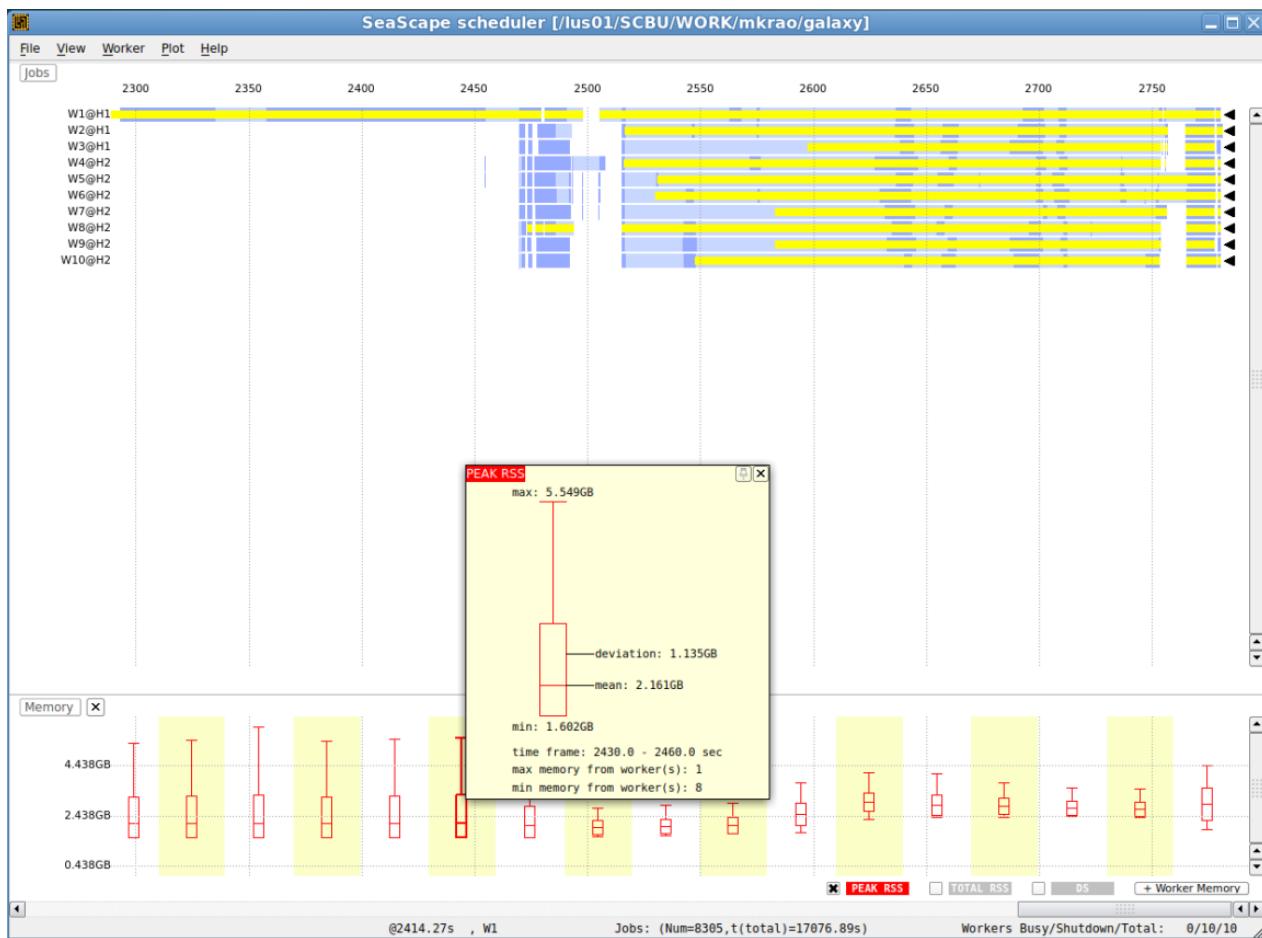
The Scheduler-GUI can display various worker statistics about the progress of the runs and performance of the hosts.



By default, the Scheduler-GUI opens with the **Memory** Panel. You can add any of the available panels to the display. At a time, you can only display two panels in addition to the **Worker** Panel.

A panel displays information as vertical error bars. For example, the default **Memory** panel displays the maximum, minimum, and mean memory consumption of the workers in 30-second intervals.

You can click each of these intervals to view the worker with the maximum and the minimum memory consumption.



This opens a popup window that displays the following information:

- Start time and end time of 30-second interval
- Maximum and minimum memory consumed by any worker during this interval
- Average memory consumed by the workers in this period
- Worker that consumed the maximum and the minimum memory
- Mean memory consumed
- Standard deviation

The standard deviation, with the mean memory consumed, enables you to determine the memory spread and how well-distributed the jobs are.

Other available panels also display the information that is captured every 30 seconds. The following table shows the other available panels.

Panel Name	Function
Worker Load Panel	Captures the system load on the workers
Scheduler Load Panel	The load on the host which is running the Scheduler GUI
DS Read Rate Panel	Read Rate of the internal view files of RedHawk-SC
DS Read Size Panel	Read size of the internal view files of RedHawk-SC
DS File Panel	Size of the view related files in the memory or the disk

Panel Name	Function
Network Data	The amount of network data transacted

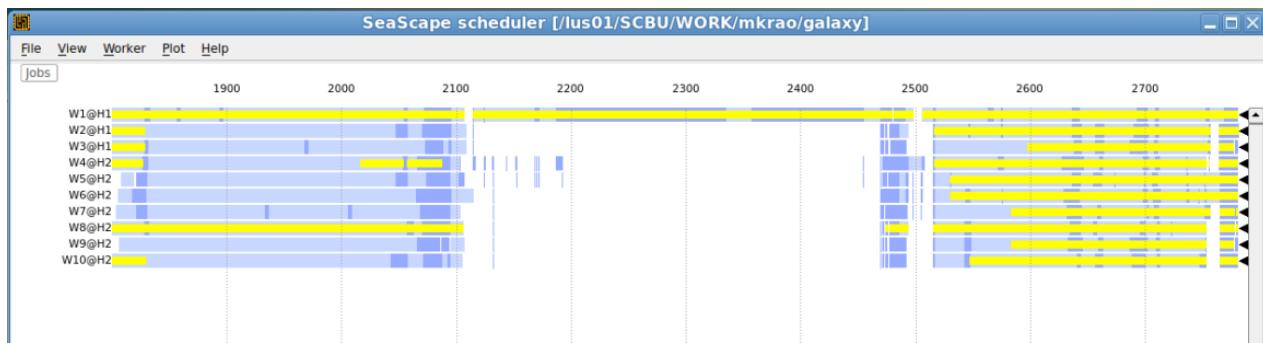
The plotted worker loads are normalized load averages, that is, the load information from the hosts is divided by the number of CPU cores. As each worker uses a single core, the normalized load average indicates whether the host is loaded.

15.2.4. Debugging With Scheduler

The Scheduler GUI is an effective tool to debug and profile analysis, and can display multiple indicators of sluggish performance.

Understanding Performance Sticks

Typically, the RedHawk-SC tool processes most jobs in parallel. To start executing, some jobs depend on the results of specific jobs. The tool holds the execution of the dependent jobs till the completion of these specific jobs, and marks these specific jobs as performance sticks in the Scheduler-GUI display.



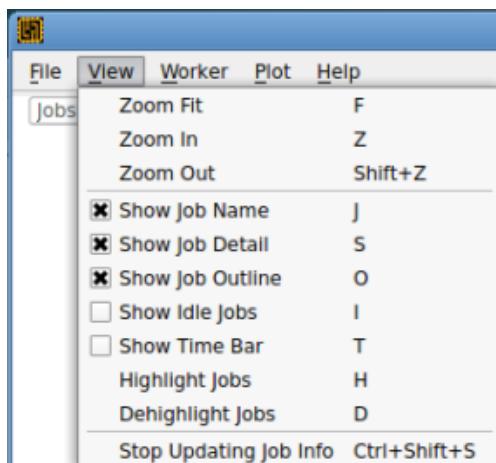
As shown in the figure, the stick job holds the rest of the dependent jobs in the flow. This increases the runtime and reduces scalability (runtime versus number of CPUs).

For information about job duration and job memory consumption, click the displayed stick job.

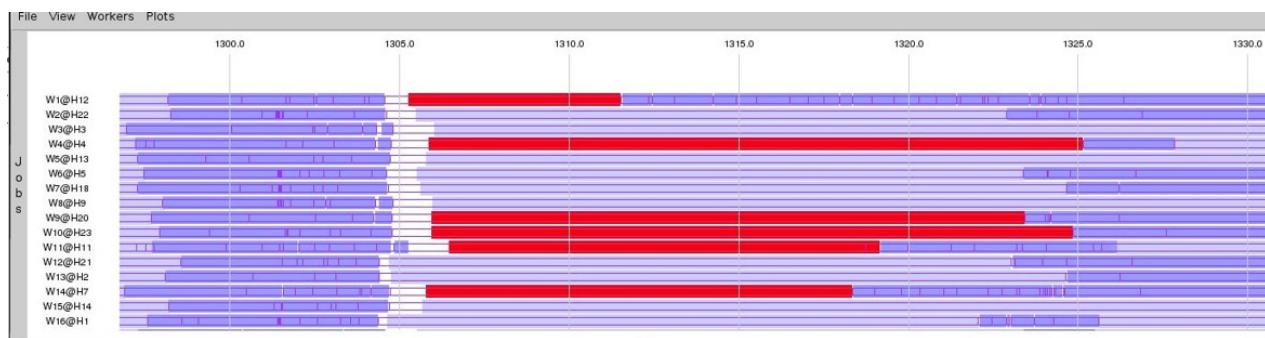
You must report such jobs for potential performance improvements. Some solutions are to partition a large job into smaller jobs or to change dependencies of the scheduled jobs so that they run in parallel to the stick job.

Network or Scheduler slowness

Any network or Scheduler Slowdown is indicated on the Scheduler-GUI by using **View > Show Job Detail** (or the keyboard shortcut 's').



This is best done when zoomed into a particular time area in the GUI as drawing the details for the full jobs might take some time. A specimen of the zoomed-in GUI with the Scheduler Details highlighted is shown below.



The purple outlined boxes indicate the Scheduler's job submission and finish times. The filled blue boxes indicate the worker's job start and finished times. The light blue boxes are for jobs whose delta between the scheduler job finished time and worker job finished time are greater than 5 seconds. The red boxes are the difference in delta between scheduler's finish and start times and the worker's finish and start times which are greater than 0.5 seconds or 5% of the delta between scheduler's job finish and submission times. To put it in an equation,

$$\Delta_{Scheduler} = t_{SFinish} - t_{SSubmit}$$

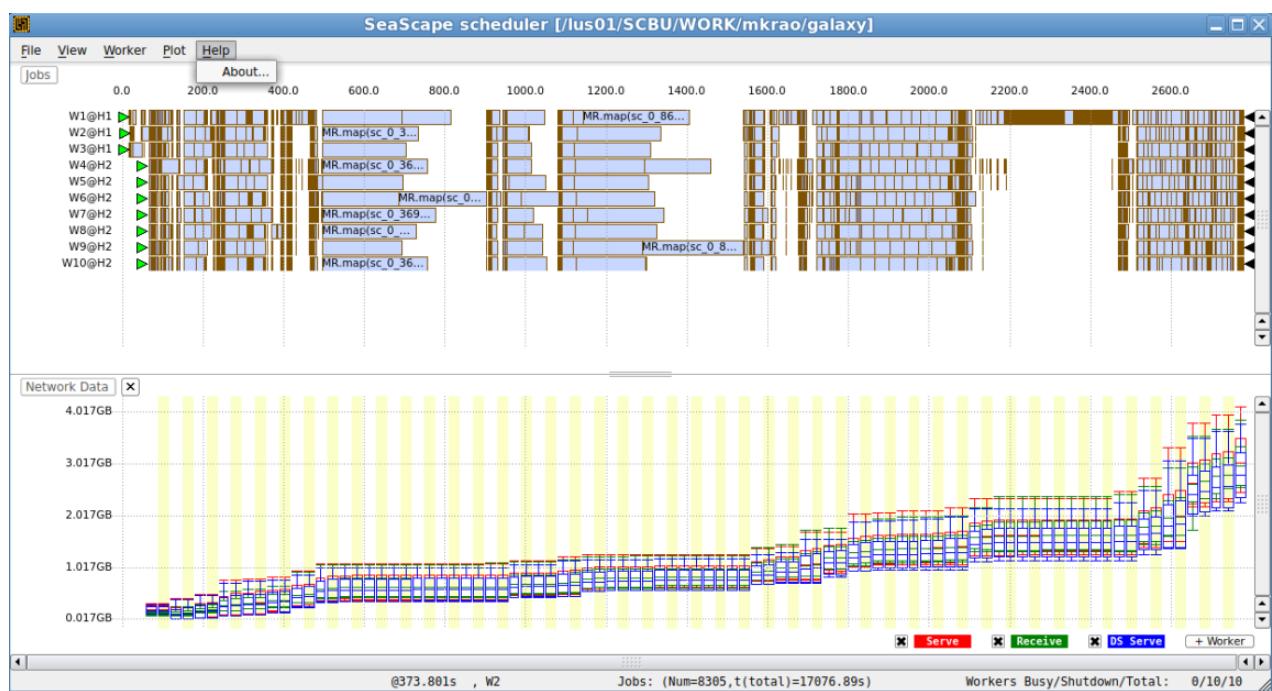
$$\Delta_{Worker} = t_{WFinish} - t_{WSubmit}$$

$$t_{RedBox} = \Delta_{Scheduler} - \Delta_{Worker} > 0.5 \text{ s or } 5\% \text{ of } \Delta_{Scheduler}$$

The red boxes indicate network problems and/or scheduler slowdown. Scheduler slowdown is when the scheduler is running on a host with very high loads.

In RedHawk-SC, Scheduler is a completely independent process and can be opened irrespective of the session. The Scheduler Host Loads are available at every 30s either from the verbose log file (run_v.log) or by displaying the Scheduler Load Panel.

We can also use the **Network Data** Panel to get error bars for the data sent and received by each worker along with DS File Served for each of the workers.



Highly Loaded Machines

Turn on Worker **Load** Panel, to understand the load on the execution hosts where the workers are running.



The figure shows that the run was completed on relatively lightly loaded machines. At any point, no worker has more than a normalized load of 0.35. To normalize the load from execution hosts, divide the load reported by the machine by the number of the CPUs in the machine.

Slow Disk Access

To determine slow-disk access in the analysis, display the DS Read Rate and DS Read Size panels.

Relatively Slower CPUs

To display configurations for each worker, use **Worker > Machine Configs** menu. You can then compare the hardware specifications of workers to make sure that the analysis is run on machines of similar capability.

15.3. Managing Disk Space

In its simplest form, the on-disk representation of a SeaScape database is a regular directory with multiple sub-directories and files. The following topics describe disk space management methodologies of the SeaScape database for the RedHawk-SC tool.

- [Overview](#) on page 452
- [Methods of Disk Storage](#) on page 453
- [Merging Distributed Databases](#) on page 457

- [Profiling Databases](#) on page 458

15.3.1. Overview

The following figure shows a typical SeaScape directory structure on a GNU or Linux system. Each subdirectory represents an available ‘view’.

```
tree -d ./galaxy_database --noreport
./galaxy_database
├── av_dynamic
│   └── ds_files
├── av_static
│   └── ds_files
├── dv
│   └── ds_files
├── dv0
│   └── ds_files
├── emv
│   └── ds_files
│       └── staged
├── ev
│   └── ds_files
├── evx
│   └── ds_files
├── lv
│   └── ds_files
├── nv
│   └── ds_files
├── pwr
│   └── ds_files
├── reff
│   ├── ds_files
│   └── staged
├── scn_dynamic
│   └── ds_files
├── scn_static
│   └── ds_files
├── sv
│   └── ds_files
├── swa
│   └── ds_files
└── tv
    └── ds_files
```

A view is a collection of common data types that are grouped together for ease of generation, organization, query, and reuse. For example, the DesignView contains geometries, netlists, and names based on the input DEF and GDS data. An ExtractView contains circuits generated by parasitic extraction of a DesignView. An AnalysisView contains node and instance voltages from a power grid noise simulation. The database can also contain views that you generate, such as a UserView.

Each view further contains a subdirectory called ‘ds_files’ that stores the view-specific files. The tool reads these files when the view is loaded onto the RedHawk-SC GUI.

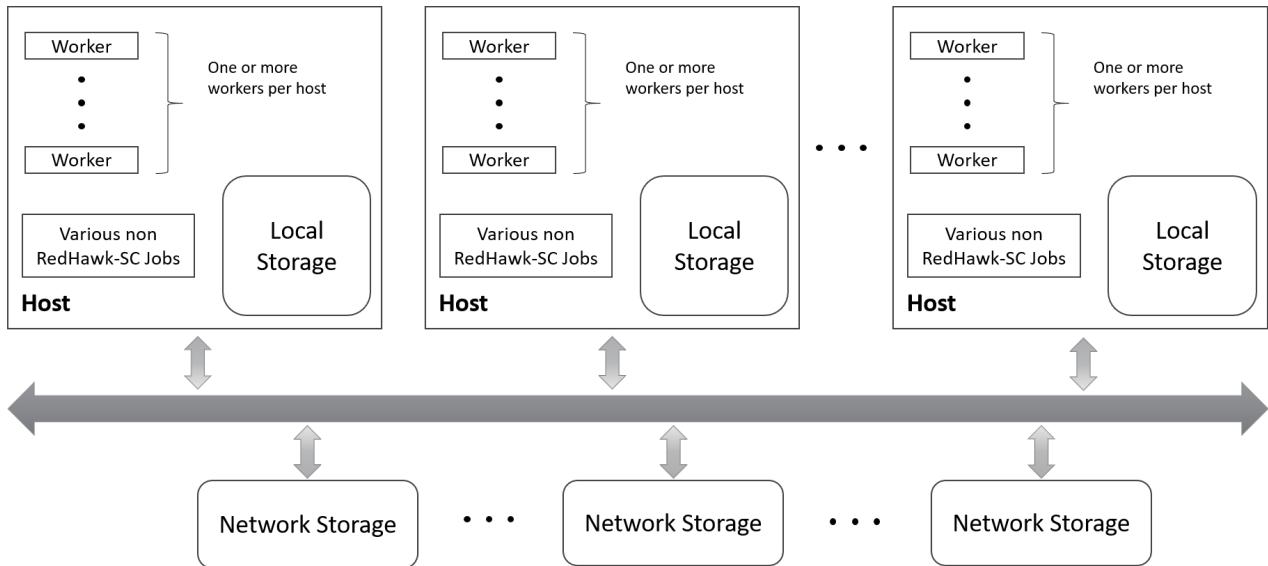
You can create and access multiple SeaScape databases in a single RedHawk-SC session. All SeaScape databases are write-once read-many. Once data is written into the database, it cannot be modified or deleted. Therefore, when multiple views are created in the database, corresponding older views remain intact.

To distinguish between different versions of the same view (if you do not explicitly tag each view), the tool automatically appends a version number to the default view name. For example, if a DesignView tagged dv is already present in the RedHawk-SC database and the same database is used to create another DesignView with the same tag, the tool creates the dv#v1 directory.

The data generated for the views are first kept in the memory of each worker, and then written in parallel to the disk. The following section describes the available methods of storing data in a disk.

15.3.2. Methods of Disk Storage

Consider the following representative diagram of a compute cluster.

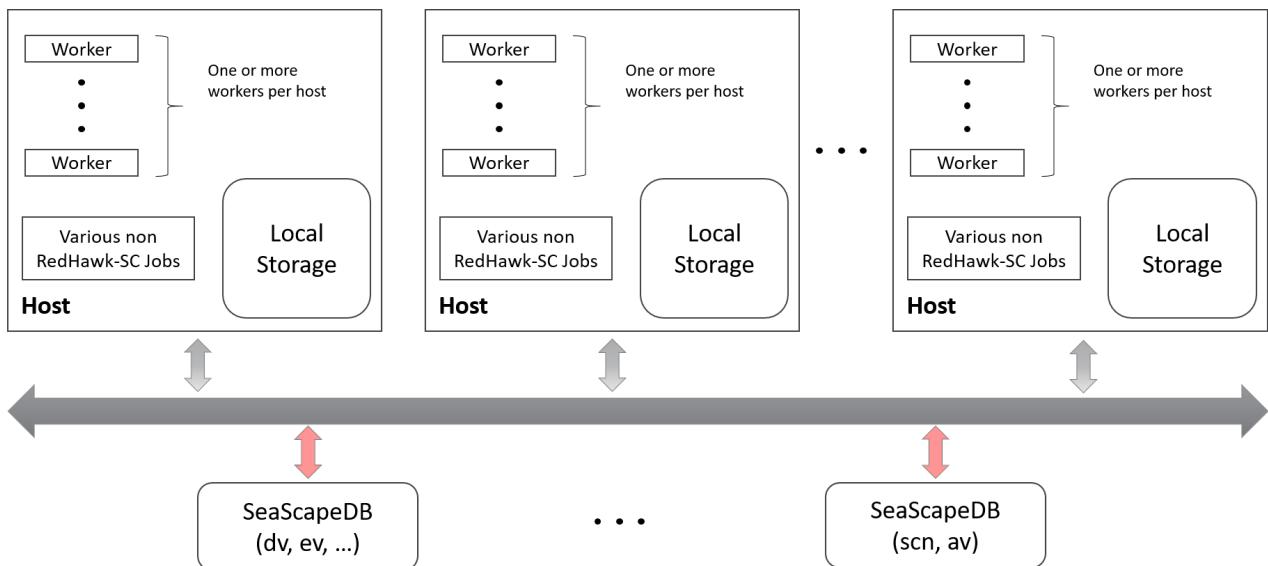


The following storage options are available for use with the RedHawk-SC tool:

- A shared disk location is accessible from all the worker host machines. See [Creating a SeaScapeDB with Multiple Storage Locations](#) on page 453.
- A central disk is a single shared disk accessible by all workers. See [Central Disk](#) on page 454.
- A distributed disk location is a local disk space in the worker host machine that is not accessible by workers running in other host machines. See [Creating a Distributed SeaScapeDB](#) on page 456.

Creating a SeaScapeDB with Multiple Storage Locations

The method of saving a SeaScapeDB is used when one of the disks might not be enough to hold the database. Multiple storage locations can be provided to the RedHawk-SC tool to create the database. You do not have control over the locations where each view is saved. The tool automatically distributes the storage to the given locations. While saving to a location, the tool first uses the largest disks. At the start of saving each view, the tool checks the available storage space of each of the provided storage locations.



This method is better than using a single central location. With the views split over multiple storage locations, data is stored in a distributed manner. However, accessing individual storage locations and all jobs competing for common network storage remain issues.

To provide multiple locations to store the database,

```
storage_locations = [{"location":'/tmp/db1', 'type':'shrd'},
                     {"location":'/tmp/db2', 'type':'shrd'}]
db = open_db('/work/design/galaxy/galaxy_db',
             storage_locations=storage_locations)
```

This creates the `galaxy_db` directory at the `/work/design/galaxy` location. This directory holds only the meta data with pointers to the actual storage locations. Therefore, to open such a database once created, you need not provide the `storage_locations` used.

```
db = open_db('/work/design/galaxy/galaxy_db')
```

To delete the SeaScapeDB stored on remote disks, set the `force_delete` argument to `True` with the `open_db` command.

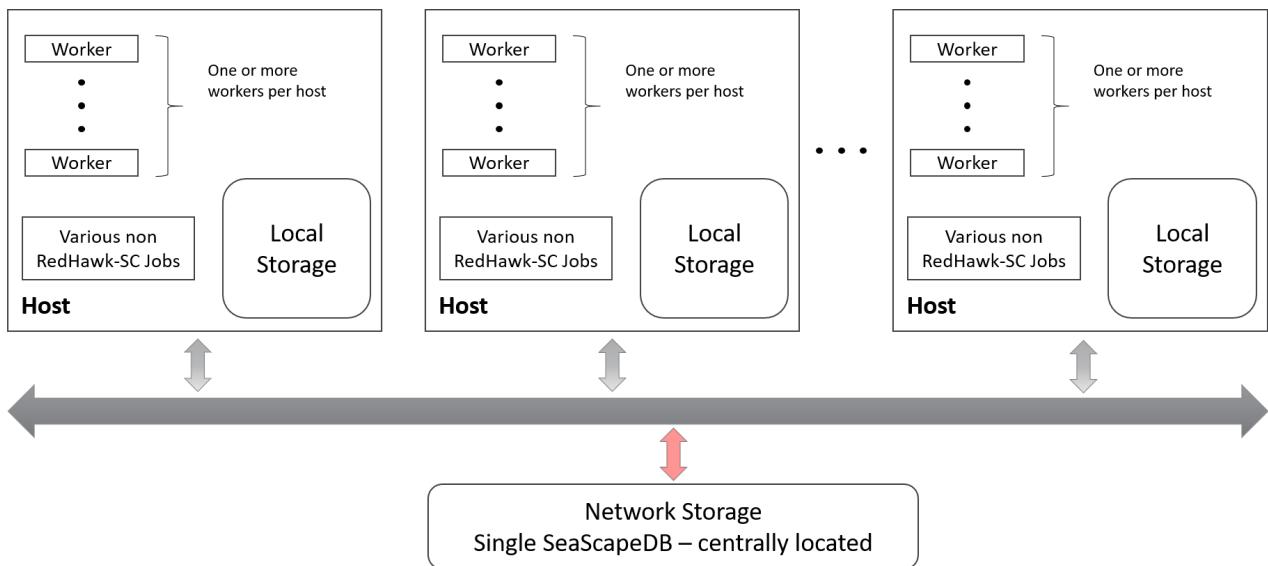
```
db = open_db('/work/design/galaxy/galaxy_db', force_delete=True)
```

Central Disk

The central disk methodology refers to using a single central directory as the SeaScape database. The location is shared in the network and all hosts in the compute farm executing jobs have access to this disk. It is convenient for you to have all the analysis-related data in a single location.

In a compute farm, a large number of jobs run on a typical host. There might be times when all these jobs try to access the storage at the same time causing network and bandwidth issues.

Further, the disk should have enough space to accommodate the SeaScapeDB and should be available to the network.



To create a central database, input the location and name of the directory to the `open_db` command. For example,

```
db = gp.open_db('/work/design/galaxy/galaxy_db')
```

To completely overwrite the database,

```
db = open_db('/work/design/galaxy/galaxy_db', force_delete=True)
```

Similarly, to open the created database,

```
db = open_db('/work/design/galaxy/galaxy_db')
```

Any new view created in this database does not overwrite any existing views. To not create unwanted views, open the SeaScapeDB in read-only mode.

```
db = open_db('/work/design/galaxy/galaxy_db', mode='r')
```

A RedHawk-SC session can work with multiple databases. You have granular control over the locations used for each view. For example, you can run AnalysisView without saving it. After the analysis, the required data is saved into UserViews that you can archive.

In the following example, `db_tmp` is stored at a location routinely cleaned by the system administrator.

```
db_save = open_db('/work/design/galaxy/scn_db')
db_tmp = open_db('/tmp/av_db')
db_archive = open_db('/storage/archive/galaxy_results_db')
scn = db_save.create_scenario_view(...)
av = db_tmp.create_analysis_view(...)
uv = db_archive.create_user_view(tag='results')
uv['instance_dvd_heatmap'] = av.get_instance_voltage_heatmap(type='eff_dvd')
```

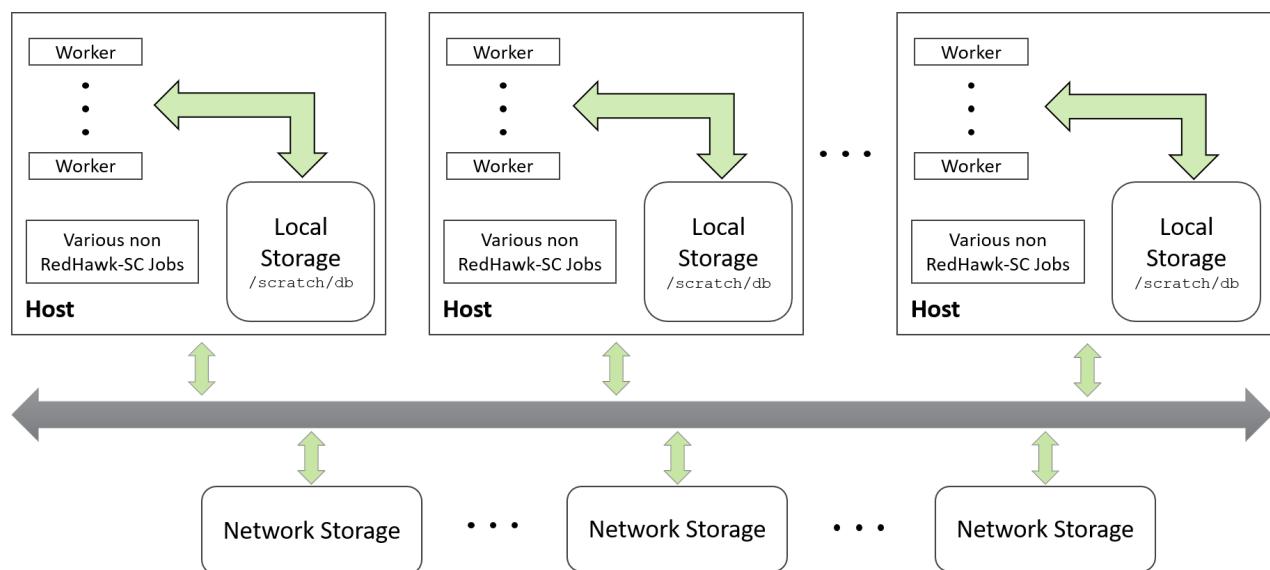
Another option is to create a temporary, in-memory database. The tool discards such a database when the RedHawk-SC session ends. The following snippet uses a temporary database to store AnalysisView results.

With the `enable_save` argument set to `False`, the created database is stored only in memory and is discarded at the end of the session. The provided location is not used.

```
db_save = open_db('/work/design/galaxy/scn_db')
db_tmp = open_db('/tmp/av_db', enable_save=False)
db_archive = open_db('/storage/archive/galaxy_results_db')
scn = db_save.create_scenario_view(...)
av = db_tmp.create_analysis_view(...)
uv = db_archive.create_user_view(tag='results')
uv['instance_dvd_heatmap'] = av.get_instance_voltage_heatmap(type='eff_dvd')
```

Creating a Distributed SeaScapeDB

When a distributed database is created, data is saved to the local disk of the worker hosts. This is different from saving the database at multiple locations because each local disk holds a chunk of the same view. The RedHawk-SC tool tracks the meta data and where the "chunks" of the views are stored.



With data stored in local disks, read and write access to local storage has no network impact. The worker distribution across the farm ensures random distribution of data. This ensures that no one host is hit significantly with IO requests to access the view data. This improves the network and IO impact of the run and makes the performance better.

To create a distributed SeaScapeDB,

```
storage_locations = [{'location': '/tmp/local_dbs/galaxy/galaxy_db',
                      'type': 'dist'}]
db = gp.open_db('/work/design/galaxy/galaxy_db',
                storage_locations=storage_locations)
```

With `storage_locations` specified, the tool stores only a small metadata under `/work/design/galaxy/galaxy_db`. The bulk of the files will be saved on each worker's host at `/tmp/local_dbs/galaxy/galaxy_db`. To open the database for reading,

```
db = gp.open_db('/work/design/galaxy/galaxy_db')
```

You need not specify `storage_locations` again to read the database because the tool stores metadata for the location of the view related-files. While opening a distributed database, the tool queries the metadata

to generate a list of hosts and automatically launches file-server daemons on those hosts using SSH. When opening a distributed database, it is customary to see messages such as the following:

```
INFO<PYG.117>    Detected distributed database, file-service daemons will be
                  launched on these hosts: host123, host345, host567.
INFO<GRD.006>    SSH Launcher created, file-service daemons will be launched
                  on hosts
                  host123, host345, host567 [1 workers per launch].
INFO<GRD.009>    Launching 3 file-service daemons with launcher
                  _ssh_lr_file_service_only439
```

File-server daemons are light-weight processes used to serve only the files residing on the host on which they are launched. They are not used for job execution. To correctly read the distributed database, you need to have `ssh` access setup without password.

If `ssh` access to hosts are disabled, you can use `rsh` (remote shell) to create file-service daemons. You can also specify `lsf`, `uge`, or any generic grid command that supports host naming specification.

To replace `ssh` with `rsh`,

```
db = open_db('./galaxy_db', launch_workers_if_needed={'type':'rsh'})
```

If you prefer to use LSF or UGE,

```
# to open a distributed DB, with LSF access
# (workers as file-service daemons and job execution)
db = open_db('./galaxy_db',
             launch_workers_if_needed={'type':'lsf', 'command':'bsub -q sh_shortqa -m
{host}'})

# to open a distributed DB, with UGE access (file-service daemons only)
db = open_db('./galaxy_db', launch_workers_if_needed=
{'type':'uge', 'command':'qsub -V -b y -cwd -j y -l h="{host}" -o {},'
'file_service_only':True})
```

In the first example, LSF is used to open the daemons to both serve files and execute jobs. The option for the daemon to execute jobs is controlled via the `file_service_only` key. In the example where UGE is used, explicitly set the daemons to be just file-service daemons. Both methods require that the compute environment support the specification to address hosts individually. In the examples, the tool replaces `host` with the host names when the daemons are launched.

15.3.3. Merging Distributed Databases

The RedHawk-SC installation kit includes utilities to merge the created database into a single location. The script can be used stand-alone from the installation folder or as part of the RedHawk-SC command file.

The stand-alone script is located in the installation directory:

```
<installation_directory>/gps/scripts/mergeDB -h

** ANSYS Inc. 2020. All data confidential **

usage: mergeDB [options]

optional arguments:
  -h, --help            show this help message and exit
  --master_db distributed_db_location [distributed_db_location ...],
                        -m distributed_db_location [distributed_db_location ...]
```

```

The db directory created successfully by earlier Redhawk-SC
run
--new_db NEW_DB, -n NEW_DB
    The location to store merged db
--list_hosts, -l      Find the list of hosts for the db.
    Outputs the py lines for launchers needed for use in a db
read script.
    No duplication of the db is performed.
--clean_new_db, -c    Clean new db directory if exists
--num_parallel NUM_PARALLEL, -j NUM_PARALLEL
    Num of parallel DB copy
--use_scp, -s          Use 'scp' over default 'rsync' to copy ds_files
directory
--report_storage, -r   Report total storage per view
--selected_views view_name [view_name ...]
    Perform merge on these views only

```

To merge a distributed database located at /work/distributed_db/galaxy_db, use

```
mergeDB -m /work/distributed_db/galaxy_db -n ./galaxy_db -c -j 10
```

This merges the views from the distributed database located across different hosts or disks to the central location, ./galaxy_db.

The merge operation can also be done from the RedHawk-SC shell or RedHawk-SC command file. The operation syntax is:

```

import db_utils

db = gp.open_db('/work/distributed_db/galaxy_db')
db_utils.merge_db(dbs=[db], central_db_path='./galaxy_db', num_parallel=-1)

```

You should first open the saved distributed database before calling db_utils.merge_db. Selected views across multiple databases can be merged with this utility. For example,

```

import db_utils

db_scn = gp.open_db('/work/distributed_db/galaxy_scn_db')
db_av = gp.open_db('/work/distributed_db/galaxy_av_db')
db_utils.merge_db(dbs=[db_scn, db_av], central_db_path='./galaxy_db',
                  selected_views=[dv, ev, scn, av], num_parallel=-1)

```

This merges only the views specified with the selected_views argument (spread across multiple databases that are specified with the dbs argument) to the central location './galaxy_db'.

15.3.4. Profiling Databases

To get the size of a database and the different ds-files that contribute to the size, use the analyze_db_size command from the db_utils package.

For example,

```

import db_utils
db_utils.analyze_db_size('./db', percent=0.002)

```

This lists all the ds_file patterns that contribute to 0.002% of the total database size.

```
>>> db_utils.analyze_db_size('./db', percent=0.002)
{'db_size': '6.358 GB',
 'views': {'av_bqm': {'ds_files': [('edge_currents_stats_smxt_', '172.592 MB')], 'total_size': '697.275 MB'},
            'av_dynamic': {'ds_files': [('edge_currents_stats_smxt_', '233.379 MB')], 'total_size': '1.107 GB'},
            'av_static': {'ds_files': [('edge_currents_stats_smxt_', '184.742 MB')], 'total_size': '833.650 MB'},
            'bqm_data_weak_kv': {'ds_files': [('stats_VDD_data_', '5.182 KB')], 'total_size': '20.917 KB'},
            'dv': {'ds_files': [('design_flat_', '3.606 MB')], 'total_size': '3.846 MB'},
            'dvo': {'ds_files': [('part_signals_', '86.959 MB')], 'total_size': '270.629 MB'},
            'ev': {'ds_files': [('part_power_nets_', '147.840 MB')], 'total_size': '673.855 MB'},
            'ev_bqm': {'ds_files': [('part_power_nets_', '156.780 MB')], 'total_size': '709.582 MB'},
            'evx': {'ds_files': [('net_ckts_nc_', '160.321 MB')], 'total_size': '174.884 MB'},
            'lv': {'ds_files': ['/home/gear/CustomerData/GENERIC/Galaxy/design_data/libs/NangateOpenCellLibrary_typical_ccs_lib_'],
                    'total_size': '9.274 MB'},
            'nv': {'ds_files': [('nrc_stackup_ ', '60.888 KB')], 'total_size': '71.211 KB'},
            'peakTW_data': {'ds_files': [('peak_current_heatmap_TW_overlap_VSS_data_ ', '2.021 KB')], 'total_size': '4.037 KB'},
            'pem_dc': {'ds_files': [('Longest_emViolation_chip_id_region_image_xt_ ', '91.559 MB')], 'total_size': '204.079 MB'},
            'pem_peak': {'ds_files': [('Longest_emViolation_chip_id_region_image_xt_ ', '94.078 MB')], 'total_size': '231.183 MB'},
            'pem_rms': {'ds_files': [('Longest_emViolation_chip_id_region_image_xt_ ', '93.195 MB')], 'total_size': '208.478 MB'},
            'pwr': {'ds_files': [('xp_power_data_ ', '23.509 MB')], 'total_size': '29.709 MB'},
            'scn_bqm': {'ds_files': [('instance_power_heatmapheatmap_obj_ ', '1.996 KB')], 'total_size': '12.241 KB'},
            'scn_no_prop_vectorless': {'ds_files': [('npv_xp_instance_data_ ', '207.391 MB')], 'total_size': '327.391 MB'},
            'scn_static': {'ds_files': [('instance_power_heatmapregion_image_xt_ ', '24.148 MB')], 'total_size': '73.441 MB'},
            'sv': {'ds_files': [('part_subxt_power_nets_ ', '132.427 MB')], 'total_size': '398.067 MB'},
            'sv_bqm': {'ds_files': [('part_subxt_power_nets_ ', '141.373 MB')], 'total_size': '421.890 MB'},
            'swa': {'ds_files': [('sa_data_ ', '35.980 MB')], 'total_size': '44.064 MB'},
            'tv': {'ds_files': [('sub_graph_ ', '28.172 MB')], 'total_size': '54.117 MB'},
            'weak_kv': {'ds_files': [('design_flat_ ', '3.605 MB')], 'total_size': '3.839 MB'}}}
```

You can use this information to reduce disk usage.

15.3.5. Controlling Memory Consumption Versus Runtime

When you need to run large designs on machines with small memory, you can use memory control modes by specifying the `set_config` command as follows:

```
set_config('global_memory_control', 'Low' | 'Medium' | 'High' | 'Ultra')
```

Note: You can add this setting at the beginning of your script. All `set_config` commands must be issued before any workers are launched.

You can specify one of the following four modes:

- **Low:** Default. Consumes highest worker peak memory and shortest runtime among the available modes.
- **Medium:** Worker peak memory consumption is less than, while runtime is more than the `Low` mode. View creation is serialized causing lesser parallelization of jobs.
- **High:** Worker peak memory consumption is less than, while runtime is more than the `Medium` mode. Before beginning the next view creation, all ds files (view-specific files) of the previously completed view are moved to disk.
- **Ultra:** Consumes least worker peak memory and longest runtime among the available modes. Most of the ds files are directly written to disk (no memory footprints).

15.4. Resiliency

Resiliency enables the tool to be reliable without intervention in unstable and loaded networks. The tool reuses saved-in-disk views before attempting to create new views. Each restart creates a corresponding work directory that enables you to troubleshoot failed runs.

A RedHawk-SC session might fail due to network glitches or unexpected system failures. Such events delay the turn around time of analysis. The SeaScape framework has the capability to automatically restart a run when the tool fails. The master process and the workers ping each other at specified intervals to ensure that the processes are active. If a ping is not responded to within a specified time or if a communication delay is detected, the tool issues warnings, such as "Slow connection...".

If socket connections between the master and the workers are broken or if any workers are killed (by the operating system or externally), the tool detects this (independent of the ping system) and kills all the workers. In such cases, resiliency enables the run to restart and complete the analysis. Peer-to-peer socket connections (between workers) automatically reconnect if broken. The tool tries to reconnect up to 30 times within 10 minutes and kills the session if peer to peer sockets cannot be established.

The following sections describe these resiliency features:

- [View Level Resiliency](#) on page 460
- [Disabling Resiliency](#) on page 461
- [Modifying Number of Resiliency Attempts](#) on page 461
- [Micro Resiliency](#) on page 461

View Level Resiliency

The RedHawk-SC analysis flow is divided into multiple SeaScape Resilient Views (SRVs). The commonly-used views, DesignView, LibertyView, TimingView, ExtractView, ScenarioView, and AnalysisView, are examples of SRVs. Each SRV is automatically saved to disk on completion except when the run is diskless. The save-to-disk operation occurs in parallel with, and does not interfere with other ongoing view creation operation.

If a run fails prematurely, the tool restarts itself and the saved SRVs are restored. The run then continues, creating the views that were not successfully completed in the failed run. The database location of these runs remains the same. However, multiple work directories, one for each attempted run are created. For example, if a folder has directories gp, gp.1, and gp.2, you can assume that there were two trials of the run in addition to the original run.

Resiliency applies to all failures except syntax related crashes. This means that where the run fails due to disk getting full, the tool exits immediately and does not attempt a rerun. Similarly, if the analysis is run with a "disk-less database", the completed views cannot be reused and resiliency is disabled.

When resiliency is applied, the tool changes its runtime behavior to maximize the success of subsequent runs. For restarted runs, once a view is completed, the tool waits until the view is successfully saved to disk before creating the next view. Therefore, the run time of a resilient run might be slower compared to a run without any failures. The aim of resiliency is to complete the run successfully on unstable networks.

When a run fails and the tool attempts to restart the run, you see the following messages in the shell.

You can see that the original run failed after the completion of DesignView. The tool attempted to revive the run from the point of failure.

```

INFO<SCH.167> View /nfs/Users/user/work/run/gp/db/lv_ss save completed.
INFO<SCH.167> View /nfs/Users/user/work/run/gp/db/lv save completed.
INFO<DVC.128> Top cell detected: Cell('train').
INFO<SCH.134> Step 'Input Design end' ['dv'] @ '2020-Mar-31 11:25:52.114684'.
INFO<SCH.167> View /nfs/Users/user/work/run/gp/db/dv save completed.
INFO<SCH.167> View /nfs/Users/user/work/run/gp/db/dv#1 save completed.
WARNING<PYG.119> View 'dv' is already used in /nfs/Users/user/work/run/gp/db. Using 'dv#2' as the name of the new view. Use the tag argument if you want to name this view explicitly.
ERROR<D15.189> Worker4 died or was killed at 2020-Mar-31 11:25:54.413501. Check Worker4.log for diagnosis.
FATAL<D15.999> Worker4 died.
Raising SIGKILL by gp_terminate (post_fatal)
/tools/redhawk_sc/bin/seascape: line 87: 7968 Killed $prefix_command $SHKEXE -c "set_config('gp.skip_completed_views',$skip_completed_views); set_config('gp.max_num_resiliency_restarts', $max_num_resiliency_restarts) " $run_settings_file_cmd "$0"
INFO<APP.411>
INFO<APP.411> Copyright(c) 2013-2020 ANSYS, Inc., All Rights Reserved.
INFO<APP.411> Use is authorized only through product licensing with ANSYS, Inc.
INFO<APP.411> A complete list of copyrights can be printed by the 'show_copyrights()' command.
INFO<APP.411>
INFO<APP.411> Version: 2020_R2.0.ENG [Optimized code built on Mar 30 2020 @ 22:52:22 (b7097123c)].
INFO<APP.411> Command Line: /tools/redhawk_sc/bin/seascape_main.Optimized -o set_config('gp.auto_read_latest_views', 1); set_config('ds.force_save_every_view', 1); set_config('gp.previous_run_work_dir', '/nfs/Users/user/work/run/gp'); set_config('gp.skip_completed_views',@); set_config('gp.max_num_resiliency_restarts', 2) -c gp_write_run_settings('/home/user/work/run/ss.7958') -c set_debug_level(1) resiliency.py.
INFO<APP.411> Work Directory = /nfs/Users/user/work/run/gp.1, Log File = /nfs/Users/user/work/run/gp.1/run.log.
INFO<APP.411> PID: 9198.
INFO<APP.411> Run started: 2020-Mar-31 11:25:56.287423 on host: sjovmuser.ansys.com.
INFO<LIC.100> Connecting to license server.
INFO<LIC.104> This software license is valid for 311 day(s).
INFO<SCH.133> Remote connection opened to '10.53.5.204':58025 for 'IdToNameConversionWorker'.
INFO<APP.106> Recovering from the failed run #1.

```

Disabling Resiliency

By default, the RedHawk-SC tool runs with resiliency enabled. To run the tool without resiliency, start the tool with the following command line switch:

```
redhawk_sc run.py --no_resiliency
```

Modifying Number of Resiliency Attempts

By default, the tool is configured to run with two resiliency repeats. If the run fails for the third time, the tool does not attempt a restart and aborts. To change the number of trials to restart the tool, use the following syntax:

```
redhawk_sc run.py --max_resiliency_restarts 5
```

This enables the tool to attempt a restart five times before aborting.

Micro Resiliency

In addition to the view level resiliency, job level resiliency, also known as micro resiliency, is available as a setting. To enable micro resiliency, use the `set_config('micro_resiliency', 1)` command at the beginning of your input scripts.

When micro resiliency is enabled and workers are killed during a job execution, the RedHawk-SC session continues with the existing workers instead of restarting from the last saved view. All jobs that were being executed by killed workers, are rerun by other available workers.

For a given job, the tool attempts the micro-resiliency restart only twice and exits when it fails for the third consecutive attempt. If all the workers executing a view are killed, tool restarts from the last saved view.

This feature has the following limitations:

- Micro resiliency does not work for the solve stage in AnalysisView.
- Micro resiliency cannot be used with the `perform_vector_selection` command. The command runs errors out in that case.
- Micro resiliency only works with shared database and not with distributed database. This affects performance and disk usage as the workers store temporary data for revival in case of failures.

16: RedHawk-SC GUI

The RedHawk-SC console provides you with an interactive interface to view results of analysis performed by the RedHawk-SC tool, such as heatmaps, and debug these results. You can open SeaScape databases of your designs, and their available views from the RedHawk-SC console. Each view stores data from different stages of power integrity analysis. You can invoke the layout window for a graphical representation of the design data and analysis results.

Further, you can invoke launchers and launch or kill workers, create custom scripts and run them, generate and view text reports, and use the built-in Help to access RedHawk-SC commands help, application notes, and other available documentation.

This chapter describes how to perform the following tasks with the RedHawk-SC console:

- [Launching the RedHawk-SC Console](#) on page 463
 - [Opening Databases](#) on page 465
 - [Viewing the Layout Window](#) on page 468
 - [Managing Workers](#) on page 487
 - [Using the Script Creation Wizard](#) on page 489
 - [Using RedHawk-SC Help](#) on page 491
-

16.1. Launching the RedHawk-SC Console

To open the console from the installation directory, use the following command. It is optional to specify the Python script.

```
<path_to_rhsc_installation>/bin/redhawk_sc <python_script> --console
```

To open the console from the RedHawk-SC interactive shell, use the following command:

```
open_console_window()
```

The RedHawk-SC console opens. The menu items and icons are described in the following tables:

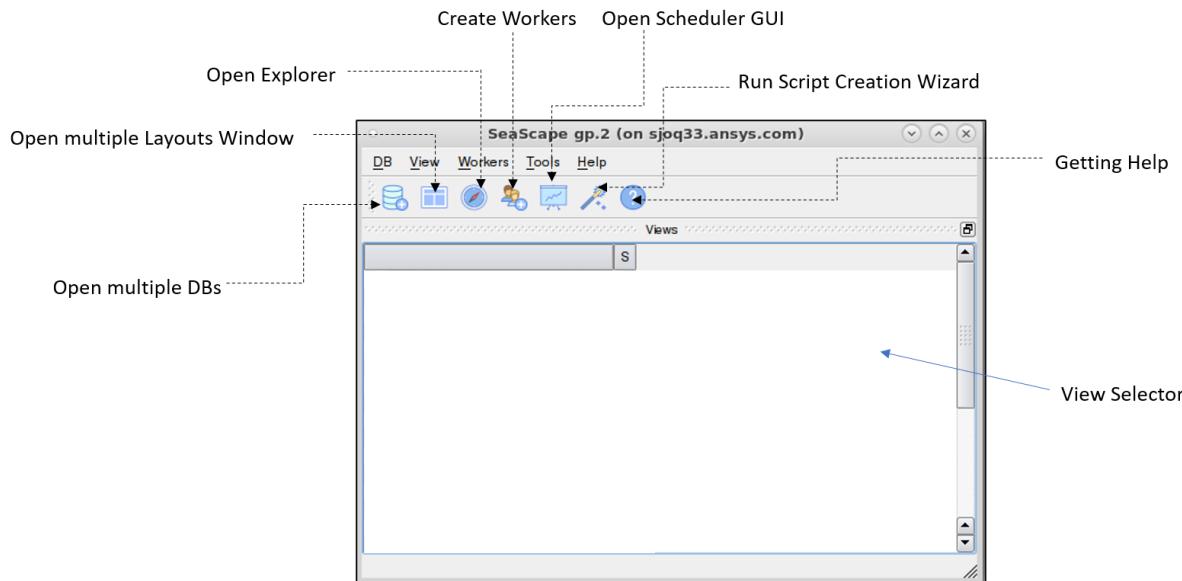


Table 23: Items on RedHawk-SC Console Main Menu

Item	Description
DB	Includes options to open and delete DBs, and delete views of a DB.
View	Includes options to open the Layout window and toggle Detailed selectors .
Workers	Includes options to invoke launchers and workers, and track and manage workers.
Tools	Invokes the Script creation wizard .
Help	Include options to invoke different help types.

Table 24: Icons Available from the RedHawk-SC Console Main Menu

Icon	Name and Description
	Open DB Opens multiple DBs and displays all the views of the DB.
	Layout Window Opens the design Layout window .
	Create Workers Launches workers interactively.
	Open Scheduler GUI Opens the Scheduler GUI.
	Script creation wizard Assists you to create scripts with commonly used Python code.

Icon	Name and Description
	Help Gets tool help.

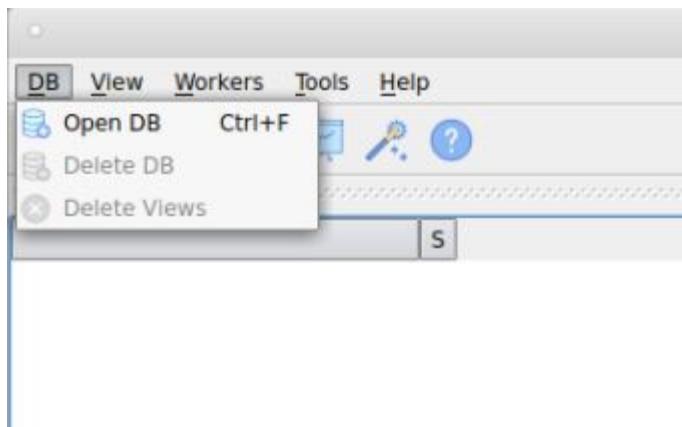
16.2. Opening Databases

To open a database,

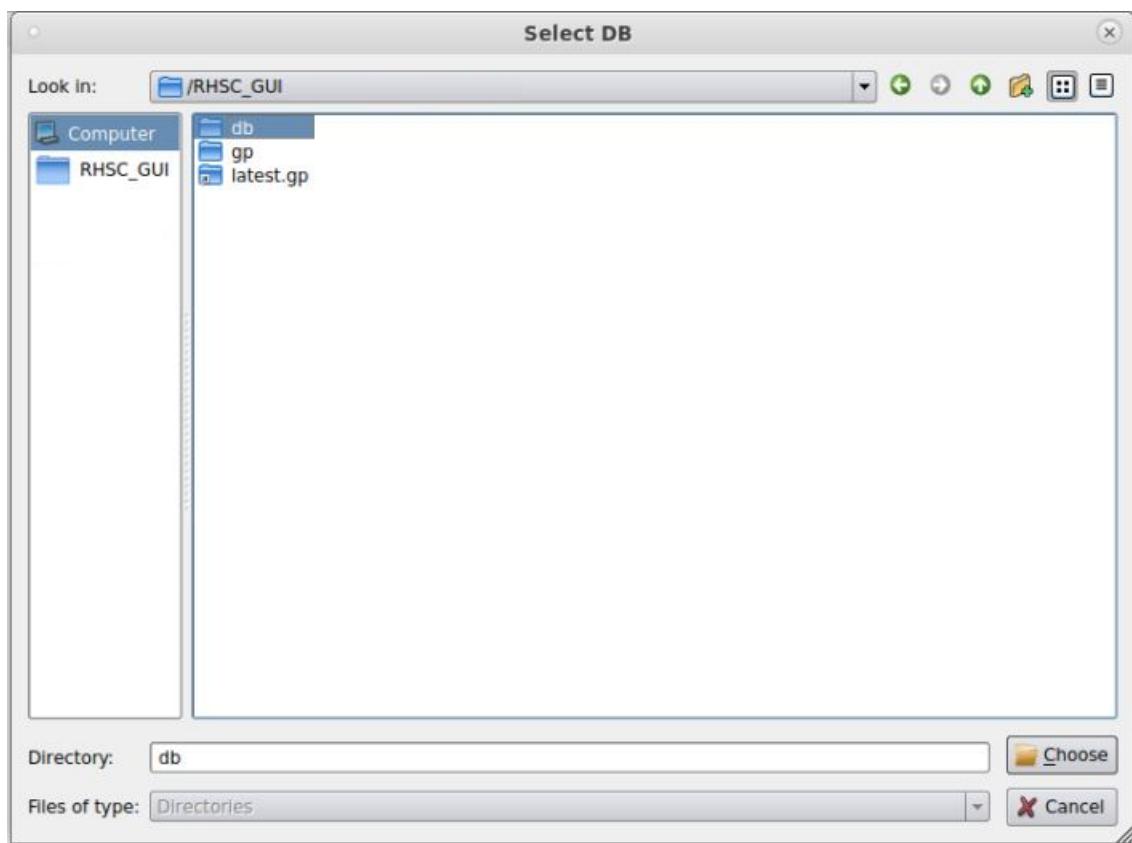
1. Select **DB>Open DB** or click the Open DB



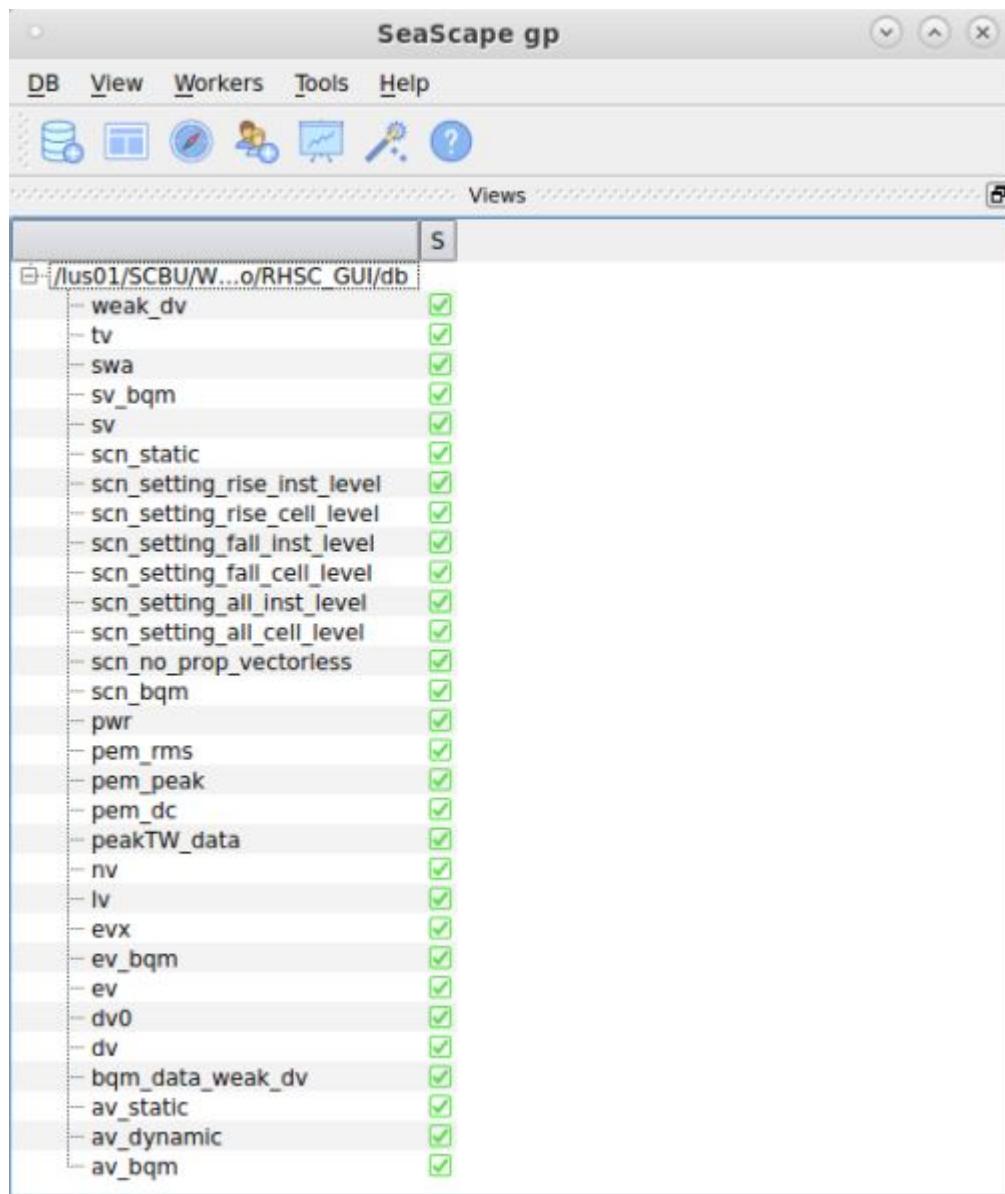
icon from the Console main menu.



This opens the **Select DB** dialog box to select and open available databases.



2. Select the directory to open. This loads the database with all the views in the console.



You can open multiple databases in parallel. The console then displays the list of open databases as shown in the following figure. You can expand a database and see the respective views. You can also select the database and open the corresponding Layout window.



16.3. Viewing the Layout Window

Overview

The layout window is a thin client for the graphical representation of the design data and analysis results. You can see the detailed layout of the design in the layout window immediately after the DesignView is complete. You can view the leaf instances, hierarchical instances, and all metals and vias in the layout with their net (PG and signal) associations.

The layout window enables you to overlay various analysis results, such as, shorts check, instance peak current distribution, layer-wise currents and voltages, on the design layout for comparison.

For easy comparison, you can invoke multiple layout windows (that are linked by default) for a single database to view different data sets in each layout with same zoom levels.

The following sections describe how to use the layout window:

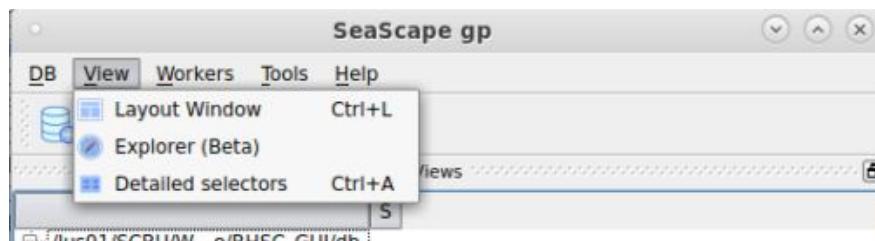
- [Invoking the Layout Window](#) on page 468
- [Using Layout Window View Options](#) on page 470
- [Identifying Small Hotspot Regions](#) on page 470
- [Setting Layout Window Preferences](#) on page 471
- [Querying and Displaying Attributes](#) on page 472
- [Viewing Current and Logic Waveforms of Instances](#) on page 473
- [Viewing Selector Panels](#) on page 473
- [Viewing Analysis Results](#) on page 486

Invoking the Layout Window

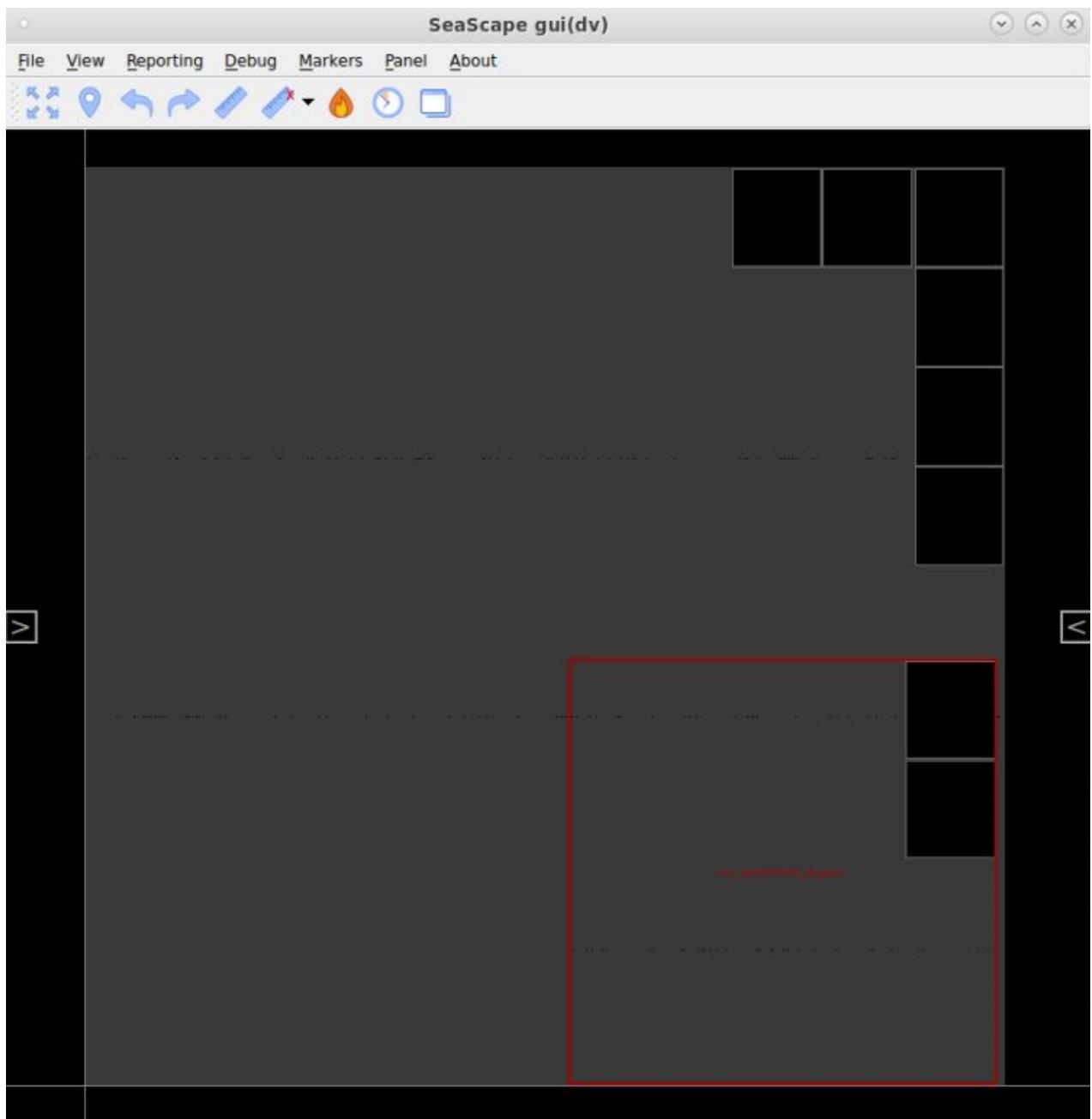
To invoke the layout window, select **View>Layout Window** or click the layout window



icon from the console menu, or press the **Ctrl+L** keys.



This opens the layout window as shown in the following figure:



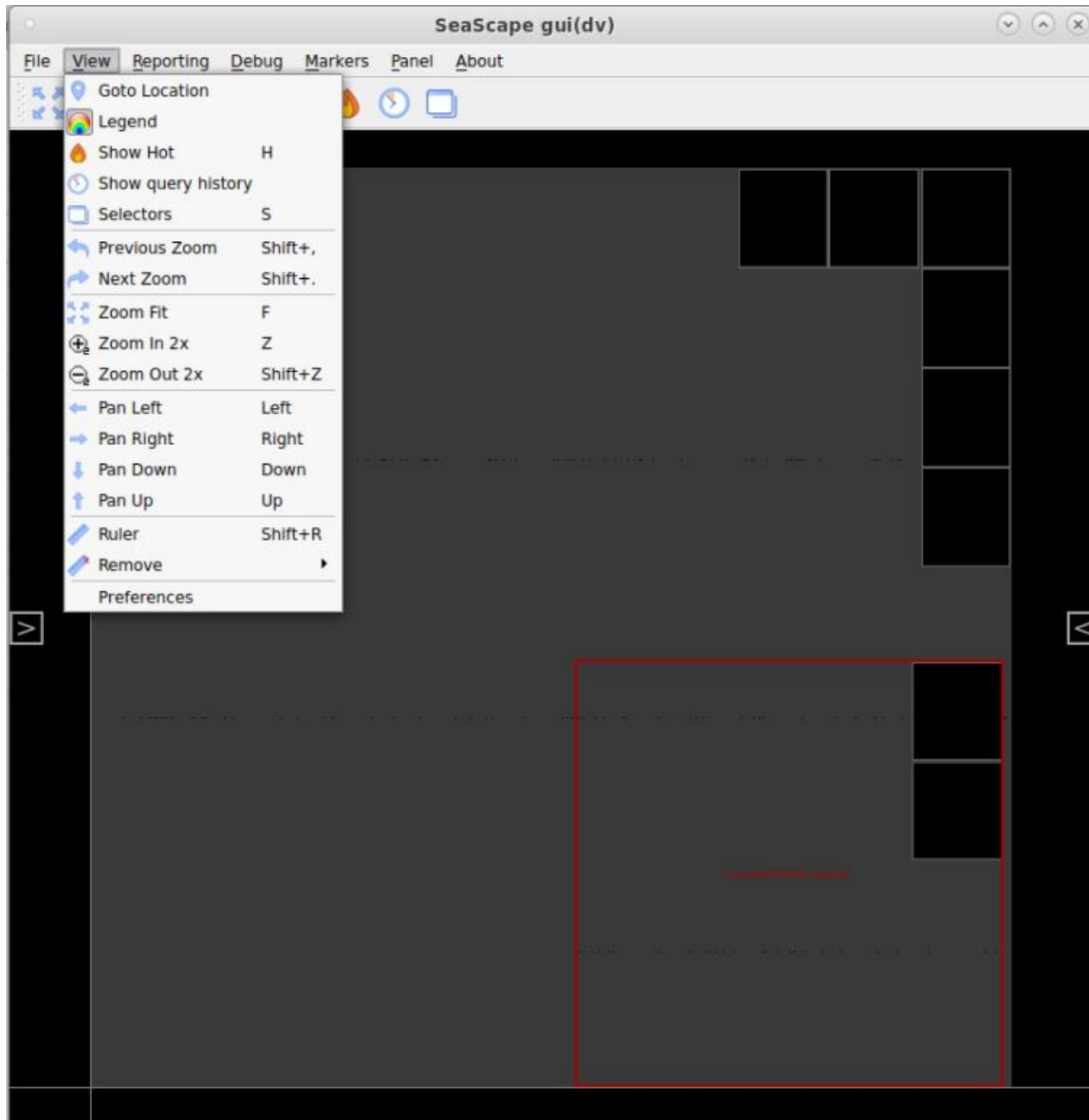
You can open and view multiple layout windows in parallel. By default, all the layout windows are linked to each other. Therefore, actions, such as zoom and pan, are consistent and have the same effect across different layout windows. This enables you to easily compare heatmaps.

In case your database has multiple DesignViews, the **Select View** dialog box opens and prompts you to select the DesignView to open from the drop-down list:



Using Layout Window View Options

The layout window has various navigation options available, as shown in the following figure with keyboard shortcuts.



The following sections describe some of the important options.

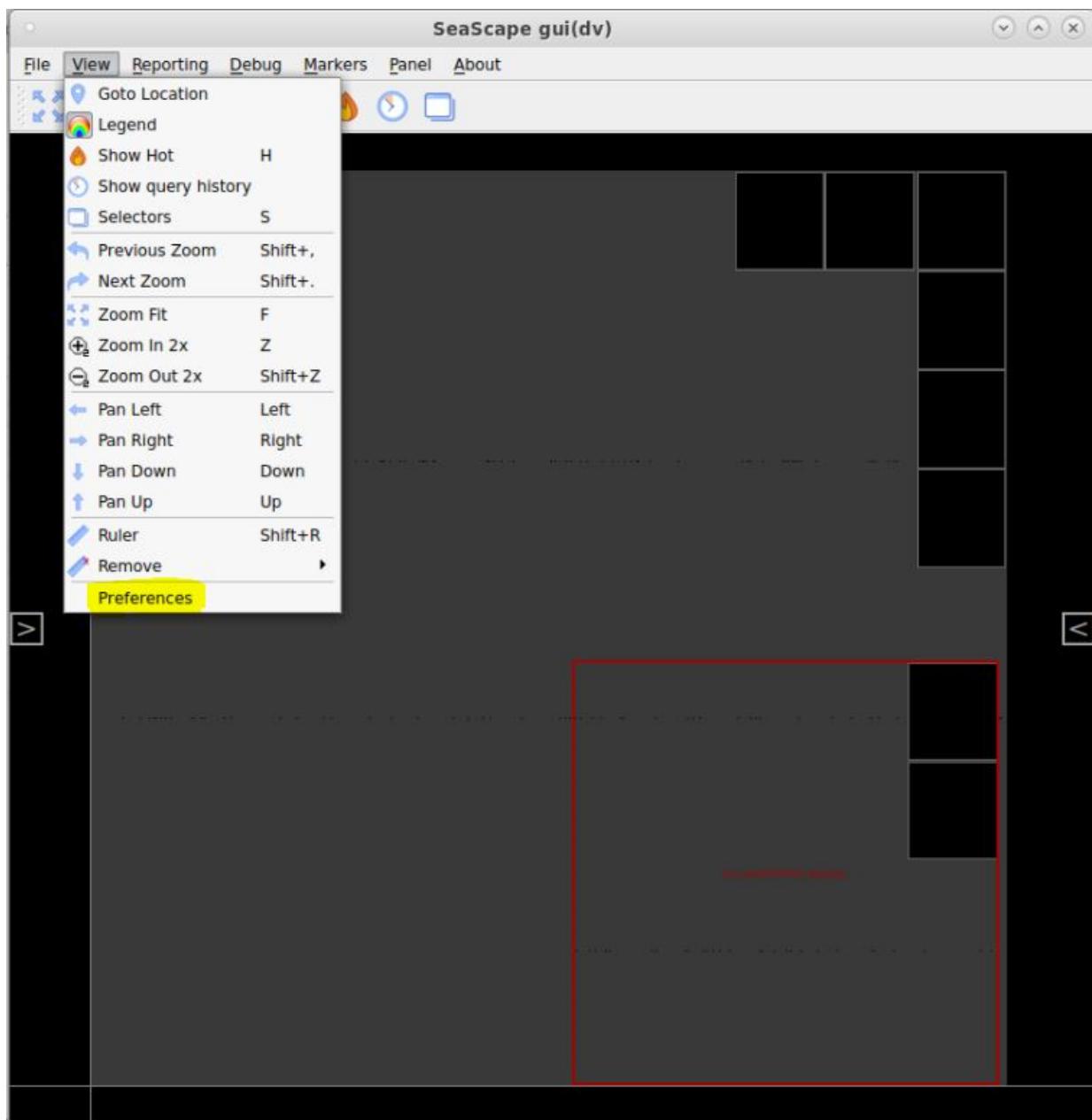
Identifying Small Hotspot Regions

It might be difficult to locate hotspots in a heatmap with few and distant hotspots having small areas.

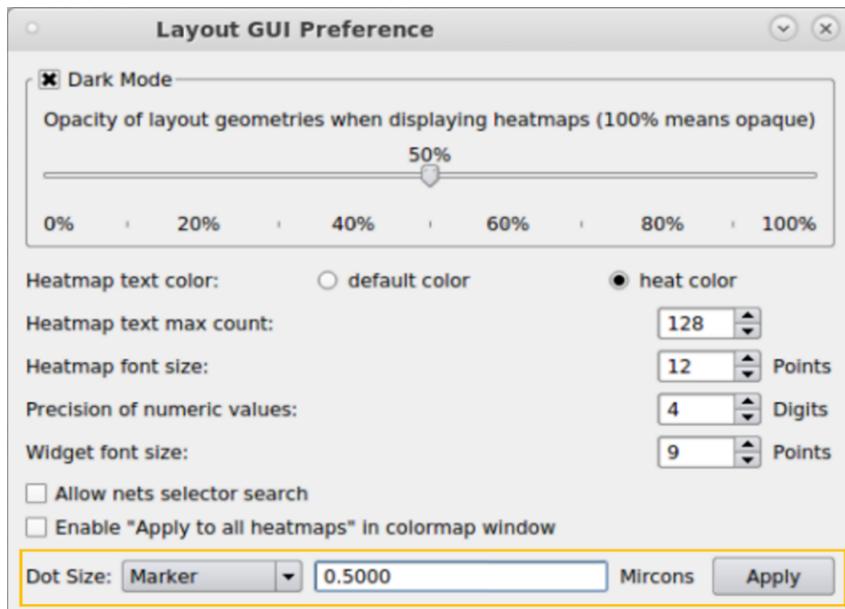
To easily identify such hotspots, select **View>Show Hot** from the **Layout Window** menu.

Setting Layout Window Preferences

1. To set your visual preferences for the Layout Window, select **View>Preferences**.



This opens the following Preference window.



2. Set the following preferences.

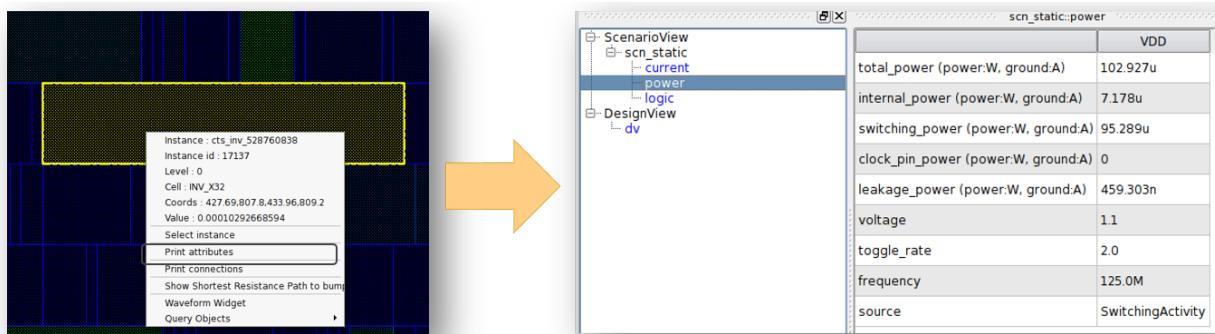
- The default viewing mode is the dark mode. To switch to the light mode, unselect **Dark Mode**.
- To control the opacity of layout geometries in heatmaps, adjust the slider position as shown in the figure.
- By default, the Layout window displays 128 text labels when you zoom in. To control the number of text objects to display at a given zoom level, set the value in **Text maximum count** drop-down box. This eliminates display issues where text messages block the display.
- To control the decimal precision and font size for text displays, set the values in **Precision of numeric values** and **Font size** drop-down boxes.
- To enable the **Search Nets** widget in the **Net** selector panel, select **Allow Nets selector search**. See [Nets](#) on page 475.
- To set the **Dot Size**, select the object such as **Marker** from the drop-down list and specify the size. This is useful in node voltage heatmaps where a small dot size makes it difficult to visualize or debug.

Querying and Displaying Attributes

You can query the attributes of all instances, wires, vias, and heatmap data displayed in the layout window.

To view the attributes of an instance, perform the following steps:

1. Click the instance to select it.



2. Right-click to obtain the following drop-down list.

3. Select Print attributes.

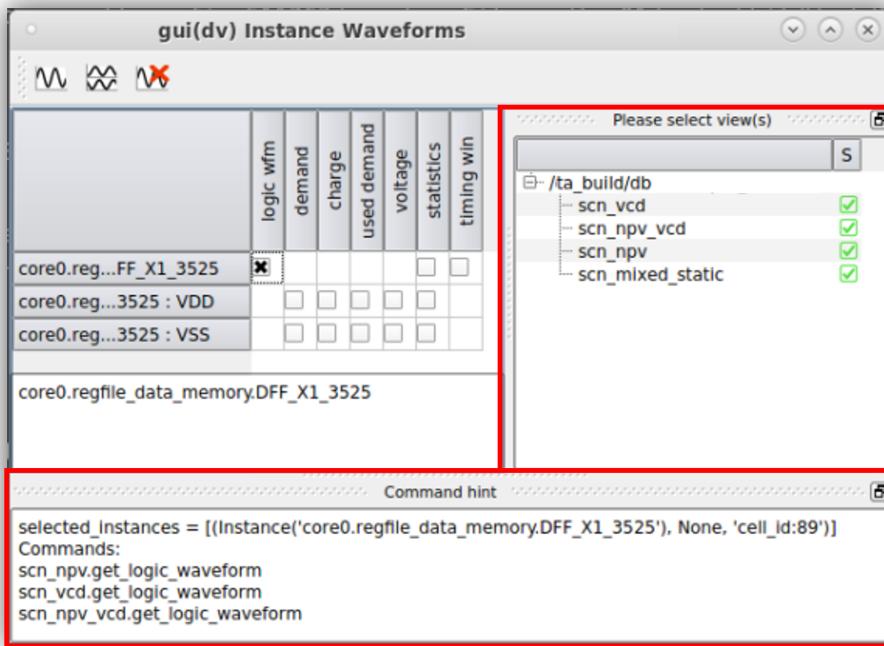
Viewing Current and Logic Waveforms of Instances

You can view waveforms of an instance of the design by using the **Waveform Widget**. See [Instance Attributes and Waveforms](#) on page 221.

You can use the **Waveform Widget** to simultaneously view the transition logic waveforms and current and voltage graphs for multiple instances. See [Viewing Simultaneous Waveforms of Multiple Instances](#) on page 222.

The **Waveform Widget** also includes the following sections:

- **select view(s)** shows all the views that are selected in the **Views** selector panel and those relevant to selected waveforms.
- **Command hint** shows the commands used to get the waveform data.

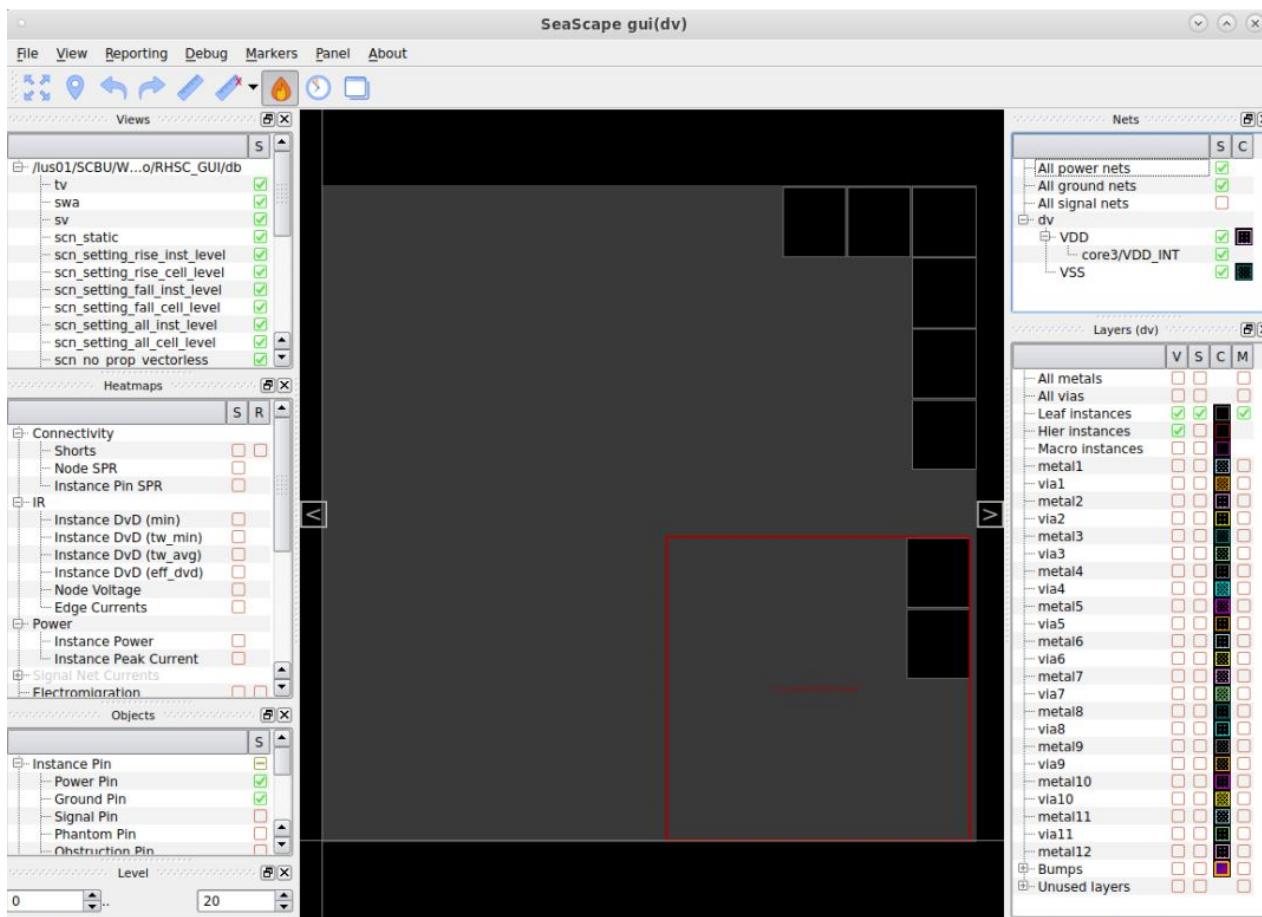


16.3.1. Viewing Selector Panels

By default, the layout window opens without selector panels.

To view the selector panels as shown in the following figure, click **View>Selectors** or press the **S** key.

This displays the selector panels with the layout window.



The GUI selector panels control the information displayed on the layout window. The RedHawk-SC layout window includes the following selector panels:

- [Views](#) on page 474
- [Heatmaps](#) on page 475
- [Objects](#) on page 475
- [Level](#) on page 475
- [Nets](#) on page 475
- [Layers](#) on page 477
- [World View](#) on page 485

Views

To select the views to show in the layout window, use the **Views** selector panel. The panel contains a list of all selectable views that are created and loaded for this run.

Certain heatmaps become available for viewing only when certain views are complete. Therefore, you must select relevant views in the **Views** panel to see certain heatmaps. For example, the **Instance DV** heatmap that displays instance-specific voltage values, becomes available only when you select an AnalysisView.

If your database contains multiple views of the same type and you select a particular heatmap, the heatmaps of all the available views are populated one over the other. To see only the heatmap of a particular view, you must select only that view.

For example, if your database contains multiple AnalysisViews and you select the **Node Voltage** heatmap in the **Heatmaps** panel, the node voltage heatmaps of all the available AnalysisViews are displayed and

overlap one another. To see only the node voltage heatmap of a particular AnalysisView, you must select only that AnalysisView in **Views**.

By default, selecting a downstream view automatically enables viewing relevant heatmaps of the upstream views. For example, selecting an AnalysisView turns on the ExtractView and ScenarioView-specific heatmaps.

Heatmaps

The RedHawk-SC tool automatically generates a default set of heatmaps at certain stages of the analysis flow. For example, the tool generates default **Shorts**, **Instance Pin SPR**, and **Node SPR** heatmaps in ExtractView. The tool also generates the default **Instance Power** and **Instance Peak Currents** heatmaps in ScenarioView.

You can also access these default heatmaps from the interactive shell..

The **Heatmaps** panel categorizes the available heatmaps under different categories, such as, **Connectivity**, **IR**, and **Power**.

To view a heatmap in the layout window, select the heatmap under a given category.

When you select the heatmap, it is added as the last element of the [Layers](#) on page 477 panel. In **Layer** based heatmaps, the [Nets](#) on page 475 panel also effects the displayed heatmap data.

Note: If custom maps are available as a top level key in a saved UserView in the SeaScapeDB, they are included in the **Heatmaps** panel.

Objects

To view objects that are stored as part of each instance, metal layer, or obstruction, use the **Objects** panel. For example, the different instance LEF pins in the design layout or heatmap.

Level

The **Level** panel represents the hierarchical levels (top design being 0) displayed in the Layout window. You can control the hierarchical instances that can be displayed by varying the values. By default, all levels (such as, 0-20) are displayed.

Nets

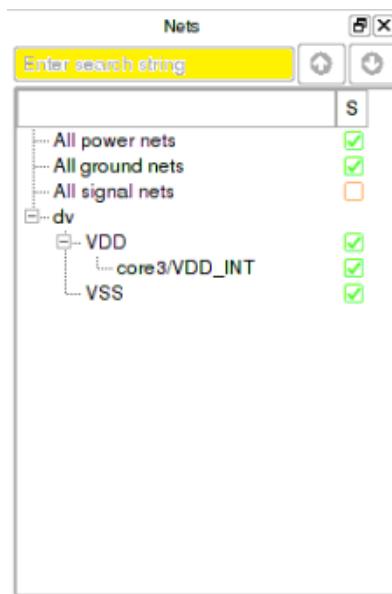
To select the nets to view, use the **Nets** panel. The panel contains the list of all power and ground nets. This enables you to view nets in the design layout and any **Layer** based heatmaps that store net-specific data. For example, the **Currents** heatmap displays the peak current distribution across every metal and via layer for each power and ground net.

The following sections describe some important features of the **Nets** panel:

- [Searching Nets](#) on page 475
- [Using Net-Based Color Mode](#) on page 476

Searching Nets

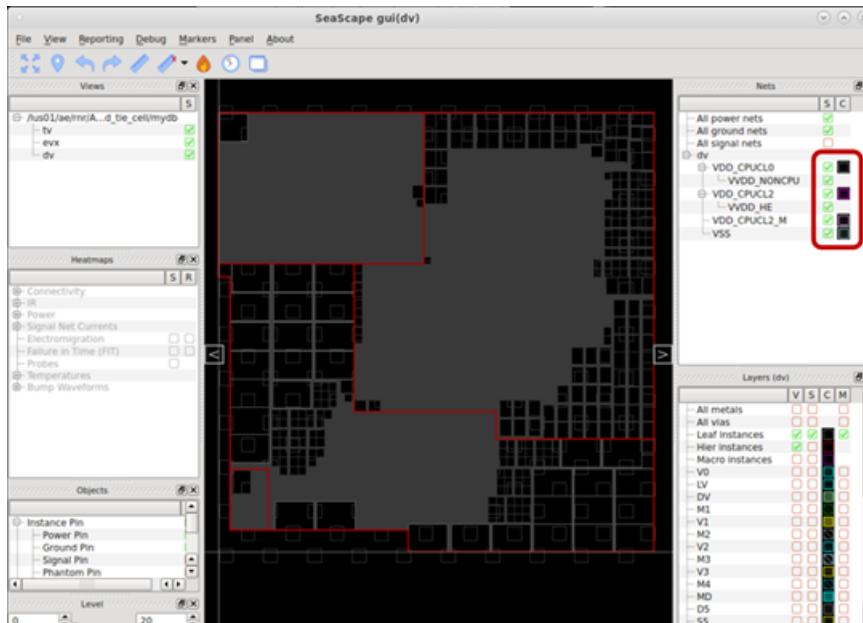
You can search for nets with the **Search Nets** field, as shown in the following figure:



To enable the **Search Nets** field, select **Allow Nets selector search** in layout preferences. See [Setting Layout Window Preferences](#) on page 471.

Using Net-Based Color Mode

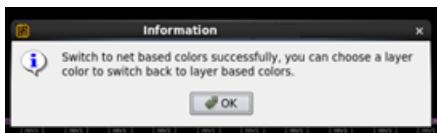
By default, the GUI layout window opens in layer-based color mode, that is, each layer has a corresponding color and stipple pattern. The **Nets** panel has the **C** column to select and view the net color palettes.



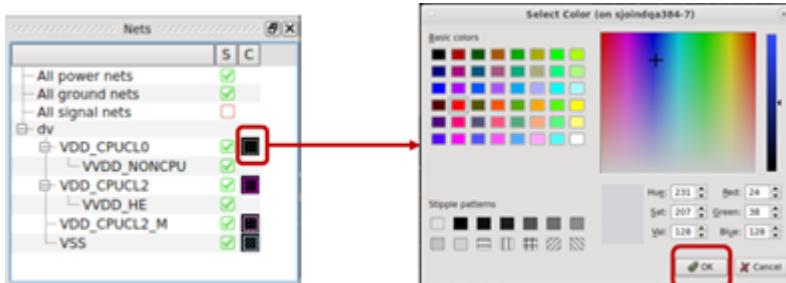
To enable net-based coloring, follow these steps:

- In the **Nets** panel, select the check box under the **C** column head next to any net.
- The **Select Color** dialog box opens. Click **OK** to turn on net-based coloring.

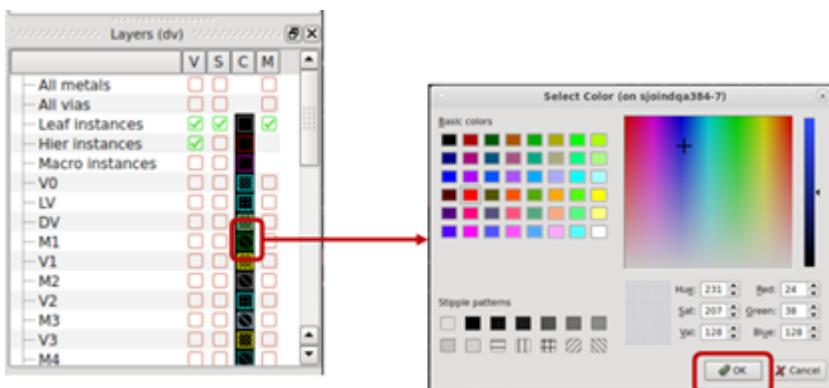
When you turn on net-based coloring for the first time, the tool displays the following popup message:



- If needed, configure the colors from the **Select Color** dialog box.



- To revert to the default layer-based color mode, repeat the same steps in the **Layers panel**. See [Layers](#) on page 477.



Layers

The **Layers** panel contains the list of available metal and via layers from the technology file stack. The following sections describe some important features of the **Layers** panel:

- [Viewing Multiple Objects in Instance Heatmaps](#) on page 478
- [Invoking the ColorMap or Select Color Dialog Box](#) on page 479
- [Defining a Single Color Map Threshold for Multi-Layer Heatmaps](#) on page 482

All default heatmaps that you select are appended to the **Layers** list. The custom heatmaps that you add to the GUI by using the RedHawk-SC interactive shell are also appended to the **Layers** panel.

The **Layers** panel has the following column heads. When you select a layer under the following column heads, the tool performs the following actions.

Table 25: Columns in Layers Panel

Column Head	Action
V View	Makes the selected layer visible in the Layout window.
S Select	Enables you to select the layer in the Layout window.

Column Head	Action
C Color map or legend	Opens either the Select Color or the colormap dialog box. Works as color legend when multiselector (M) is enabled. See Invoking the ColorMap or Select Color Dialog Box on page 479.
M Multiselector	See Viewing Multiple Objects in Instance Heatmaps on page 478.

For example, to view Metal6 and Metal7 nets, check the box under **V** for **Metal6** and **Metal7** rows as shown in the following figure:

Layers (dv)	V	S	C	M
All metals	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
All vias	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Leaf instances	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Mac instances	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
metal1	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
via1	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
metal2	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
via2	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
metal3	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
via3	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
metal4	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
via4	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
metal5	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
via5	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
metal6	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
via6	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
metal7	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
via7	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
metal8	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
via8	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
metal9	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
via9	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>

Viewing Multiple Objects in Instance Heatmaps

In an instance heatmap, the check boxes under the **M** column of the **Layers** panel enable you to view multiple layers, and multiple objects associated with each layer.

To view a map or objects associated with a layer, select the respective check box under the **M** column in the **Layers** panel.

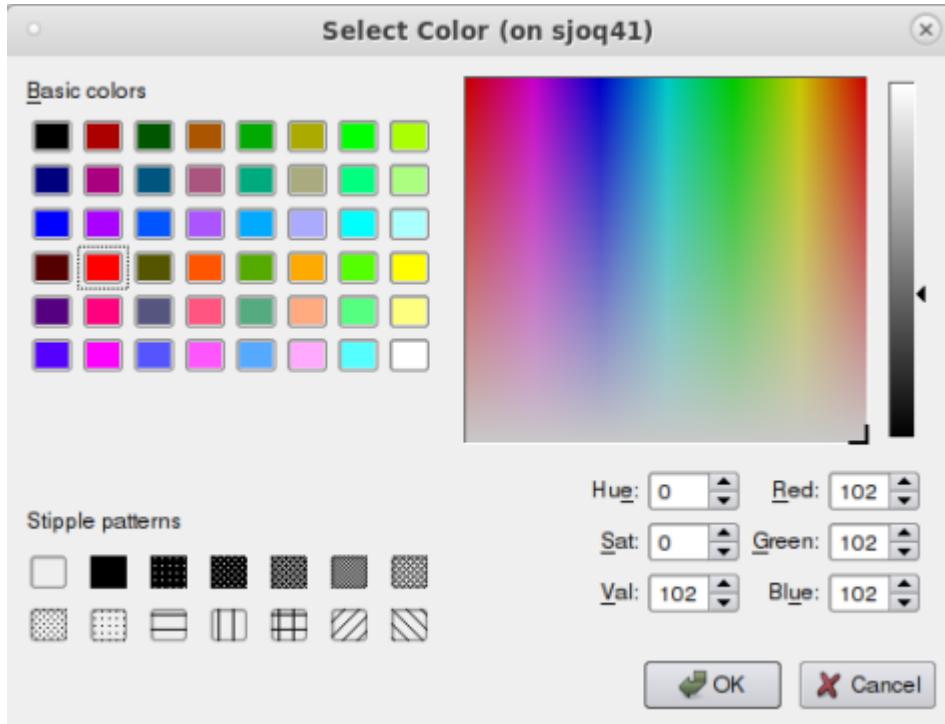
For example, to view the leaf instances of an already selected layer, select the **Leaf instances** check box under the **M** column.

	V	S	C	M
All metals	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
All vias	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Leaf Instances	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
Hier Instances	<input checked="" type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>	<input type="checkbox"/>
Macro Instances	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input checked="" type="checkbox"/>
metal1	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
vial1	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
metal2	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
via2	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
metal3	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
via3	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
metal4	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
via4	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
metal5	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
via5	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
metal6	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
vial6	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
metal7	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
vial7	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
metal8	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
vial8	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
metal9	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
vial9	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
metal10	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
vial10	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
metal11	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
vial11	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
metal12	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
Bumps	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>
Unused layers	<input type="checkbox"/>	<input type="checkbox"/>	<input checked="" type="checkbox"/>	<input type="checkbox"/>

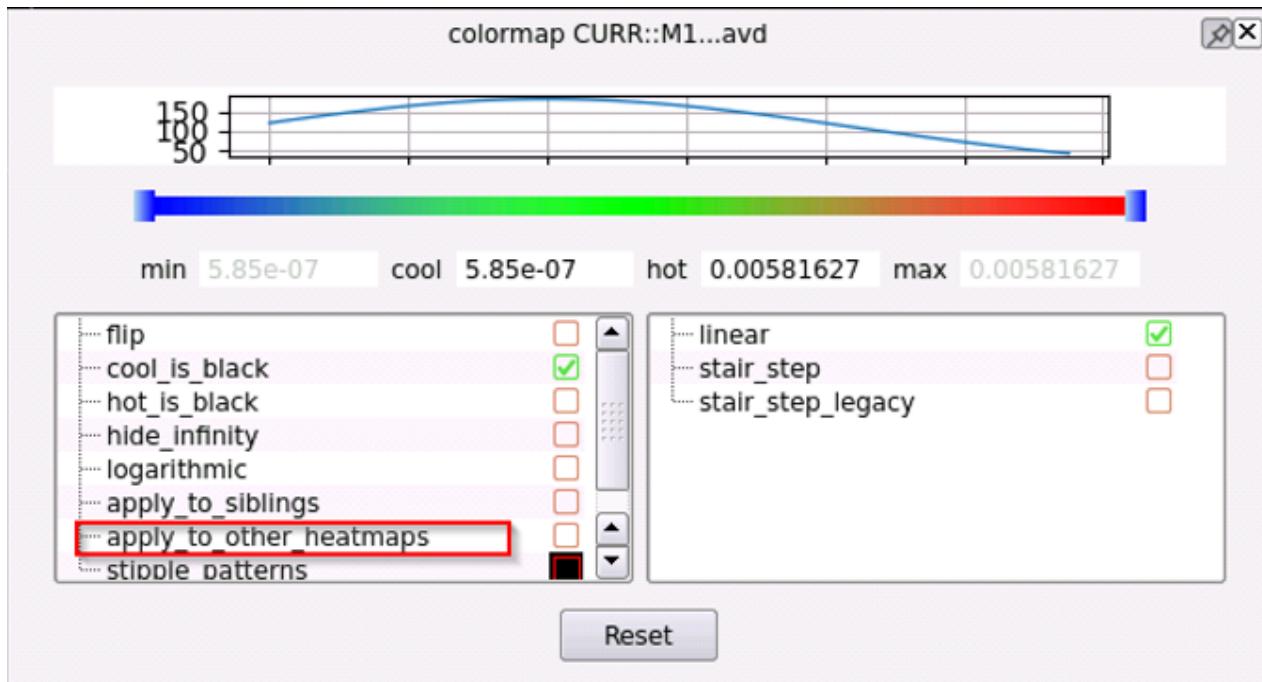
Invoking the ColorMap or Select Color Dialog Box

Depending on the type of heatmap that you select in the **Heatmaps** panel, the tool opens the **colormap** or the **Select Color** dialog box when you select the check box under the **C** column head in the **Layers** panel. See [Layers](#) on page 477.

The **Select Color** dialog box enables you to change the color of the selected layer in the layout window.



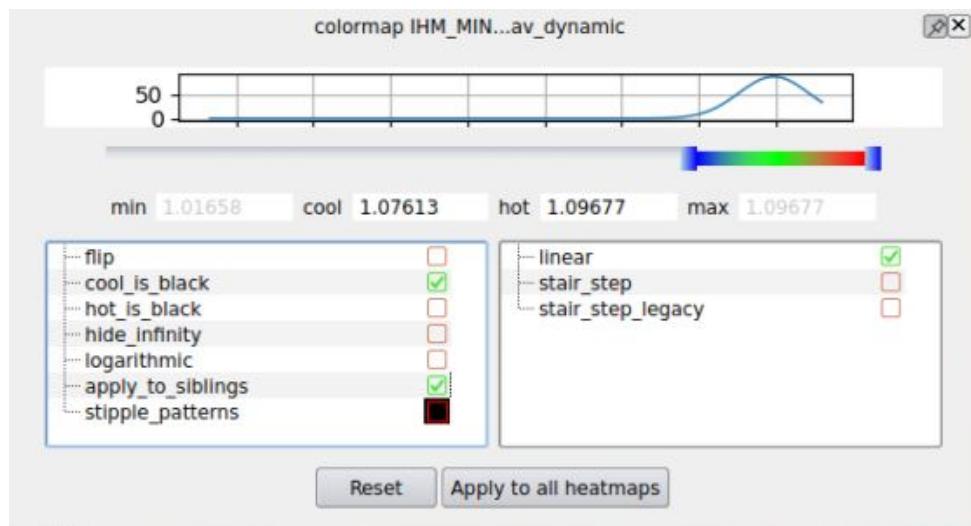
The **colormap** dialog enables you to configure your color settings. By default, the **colormap** settings apply only to the heatmaps that are natively available in the tool.



To also apply the **colormap** settings to custom heatmaps that you have created:

1. Select **View>Preferences>Enable 'Apply to all heatmaps' in colormap window** from the layout window menu.

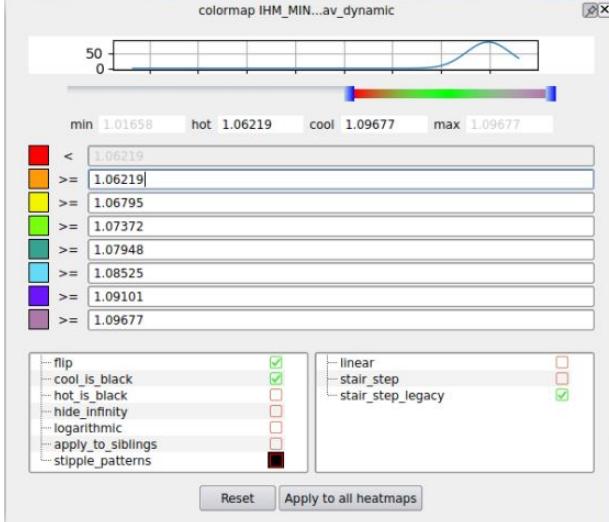
This opens the **colormap** with the **Apply to all heatmaps** button.



2. Click the **Apply to all heatmaps** button.

The following table describes the available **colormap** setting options.

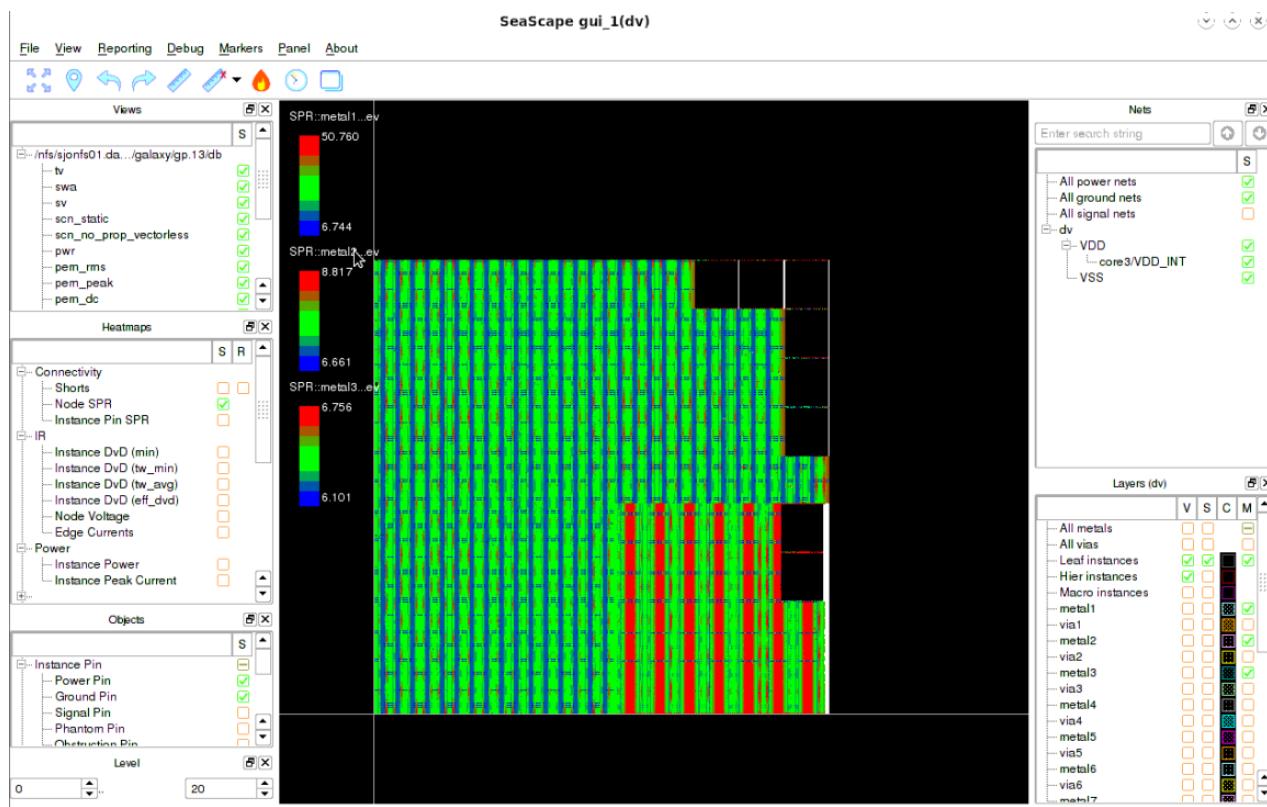
Option	Description
flip	Flips the object colors in the heatmap
cool_is_black	Displays objects with color threshold values below the minimum as black
hot_is_black	Displays objects with color legend values above the maximum threshold as black
hide_infinity	Hides display of objects with very large values. For example in SPR tracing, hides the SPR of a disconnected pin as its resistance value is infinity.
logarithmic	Makes the scale of the heatmap logarithmic. Default is decimal.
apply_to_siblings	See Defining a Single Color Map Threshold for Multi-Layer Heatmaps on page 482.
apply_to_other_heatmaps	Changes all heatmaps in multiple layout windows according to the scale chosen.
stipple_patterns	Enables you to select one of the following stipple patterns:
linear	Default colors and ranges used for heatmaps.
stair_step	Changes the colors and ranges used for heatmaps to a more discrete display from linear .

Option	Description																		
stair_step_legacy	<p>Enables you to customize colors and ranges used for heatmaps. You can directly edit the threshold values:</p>  <p>The dialog box shows a color bar at the top with a gradient from red to purple. Below it, a legend lists color ranges with their corresponding threshold values:</p> <table border="1"> <thead> <tr> <th>Color Range</th> <th>Threshold Value</th> </tr> </thead> <tbody> <tr> <td>Red</td> <td>< 1.06219</td> </tr> <tr> <td>Orange</td> <td>>= 1.06219</td> </tr> <tr> <td>Yellow</td> <td>>= 1.06795</td> </tr> <tr> <td>Green</td> <td>>= 1.07372</td> </tr> <tr> <td>Cyan</td> <td>>= 1.07948</td> </tr> <tr> <td>Blue</td> <td>>= 1.08525</td> </tr> <tr> <td>Purple</td> <td>>= 1.09101</td> </tr> <tr> <td>Black</td> <td>>= 1.09677</td> </tr> </tbody> </table> <p>At the bottom left, there are checkboxes for various options: flip (checked), cool_is_black (checked), hot_is_black (unchecked), hide_infinity (unchecked), logarithmic (unchecked), apply_to_siblings (unchecked), and stipple_patterns (unchecked). At the bottom right, there are checkboxes for colormap types: linear (unchecked), stair_step (unchecked), and stair_step_legacy (checked). Buttons for Reset and Apply to all heatmaps are also present.</p>	Color Range	Threshold Value	Red	< 1.06219	Orange	>= 1.06219	Yellow	>= 1.06795	Green	>= 1.07372	Cyan	>= 1.07948	Blue	>= 1.08525	Purple	>= 1.09101	Black	>= 1.09677
Color Range	Threshold Value																		
Red	< 1.06219																		
Orange	>= 1.06219																		
Yellow	>= 1.06795																		
Green	>= 1.07372																		
Cyan	>= 1.07948																		
Blue	>= 1.08525																		
Purple	>= 1.09101																		
Black	>= 1.09677																		

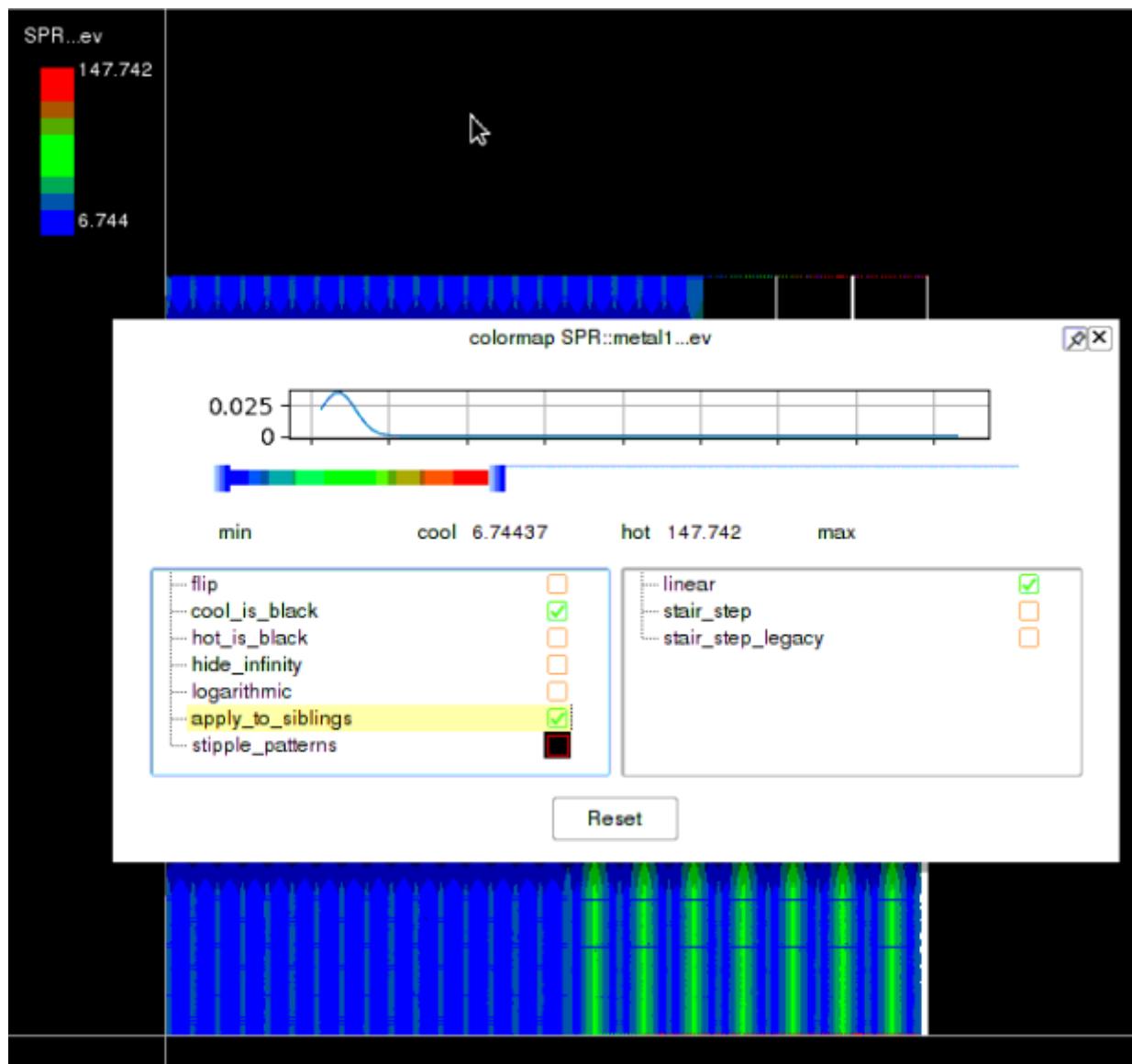
Defining a Single Color Map Threshold for Multi-Layer Heatmaps

The **apply_to_siblings** option is available for multi-layer heatmaps, such as, geometry heatmaps. This option enables you to apply the same color thresholds to each individual layer of a multi-layer heatmap.

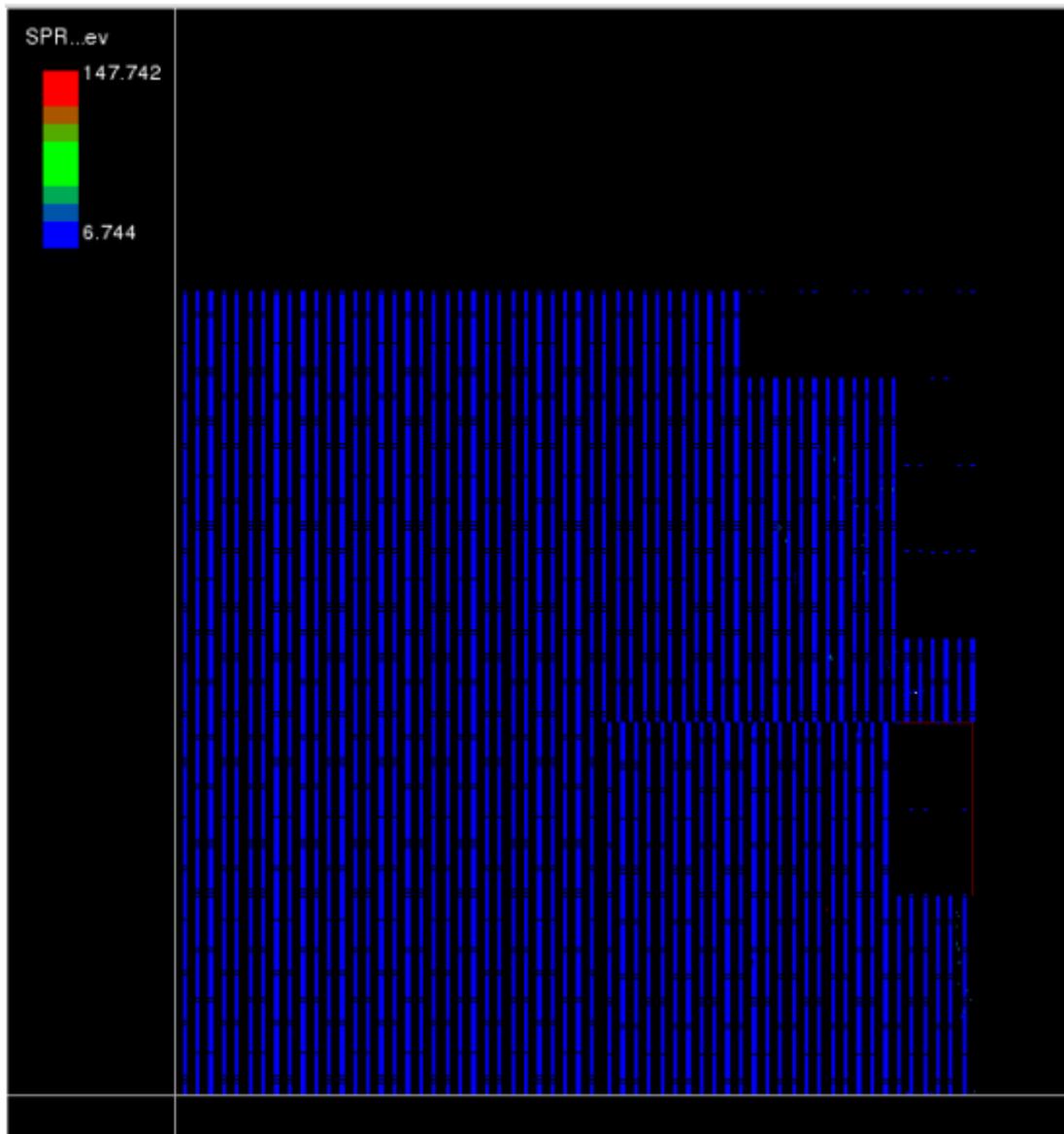
For example, consider the following **Node SPR** heatmap. In the **Layers** panel, multiple layers, **metal1**, **metal2**, and **metal3** are selected. The colormap thresholds for each layer is different.



To define a single threshold for all the layer heatmaps, select **apply_to_siblings** in the **colormap** dialog box.

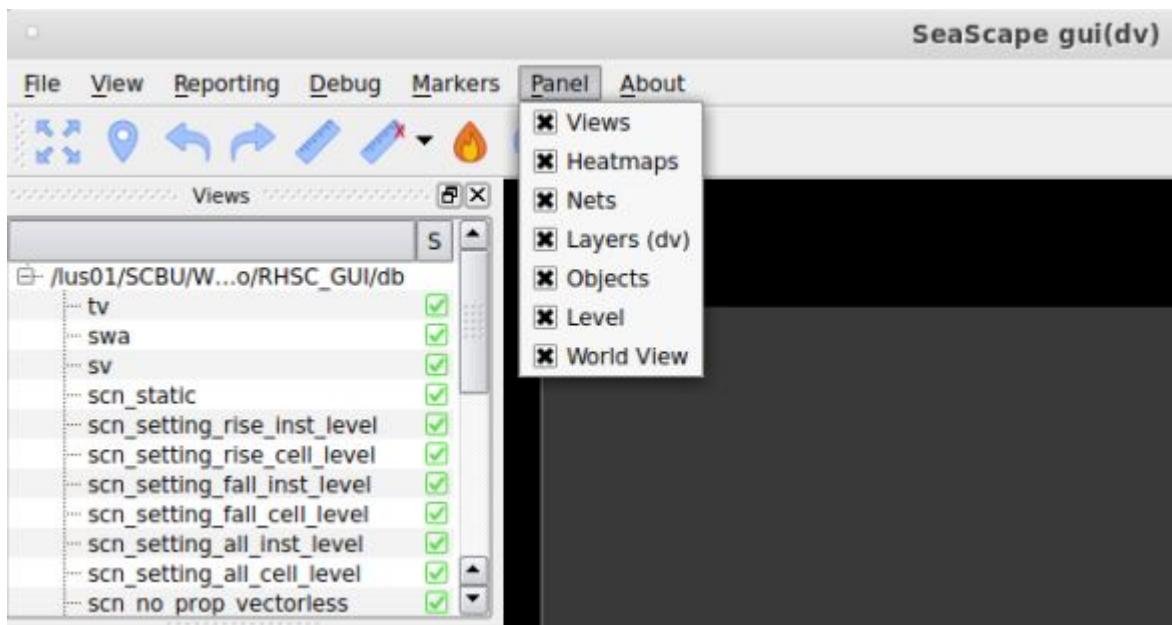


Now, all layer heatmaps have the same thresholds.

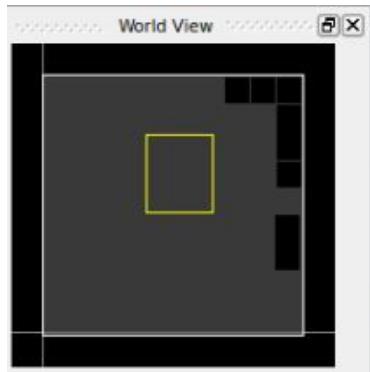


World View

The **World View** panel is not visible by default. To view this panel, select **Panel>World View** from the layout window menu.



The **World View** panel shows the relative position of the display area of the layout window with respect to the whole design. As shown in the following figure, the yellow rectangle shows the present zoom level.



16.3.2. Viewing Analysis Results

The following sections describe how to view and debug the results of different analysis in the layout window.

- Early grid analysis
 - [Viewing Shorts on page 61](#)
 - [Viewing Disconnects in GUI on page 64](#)
 - [Viewing BQM Analysis Results on page 68](#)
 - [Viewing SPR Path For a Location on page 81](#)
 - [Viewing Effective Resistance Heatmaps on page 89](#)
 - [Viewing Missing Vias on page 91](#)
 - [Viewing Resistance Gradient Heatmaps on page 94](#)
 - [Viewing PeakTW Heatmaps on page 97](#)
 - [Viewing Slack Gradient Heatmaps on page 98](#)
 - [Viewing Consolidated Heatmaps on page 99](#)
- Static analysis

- [Viewing Static Analysis Results in GUI](#) on page 129
- [Viewing Electromigration Results in GUI](#) on page 140
- Vectorless dynamic analysis
 - [Viewing Dynamic Analysis Results in GUI](#) on page 219
 - [Viewing DVD Electromigration Results in GUI](#) on page 231
 - [Viewing Instance Decoupling Capacitance Statistics in GUI](#) on page 235
- Signal electromigration analysis
 - [Viewing Signal Electromigration Results in GUI](#) on page 273
- Multichip analysis
 - [Multichip-Specific GUI Features](#) on page 364
 - [Viewing Multichip SPR Data in GUI](#) on page 370

16.4. Managing Workers

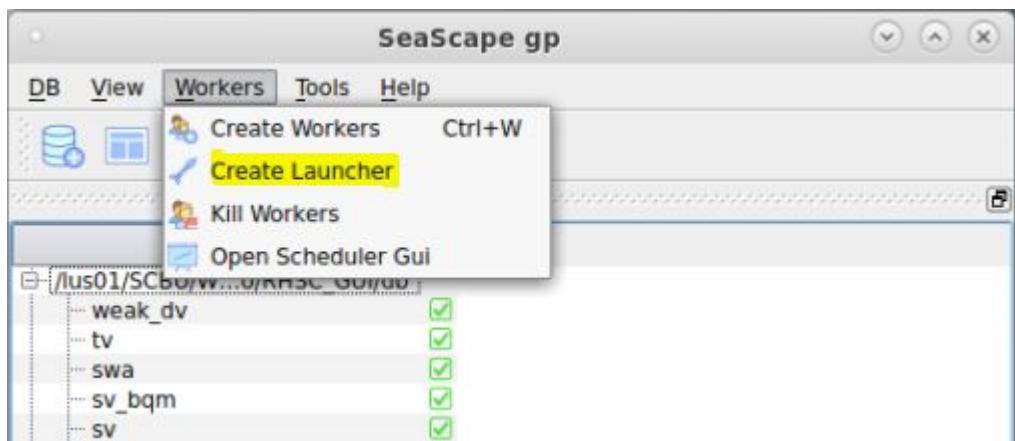
The **Workers** menu item has options to create and manage launcher objects and workers as described in the following sections.

- [Creating Launchers from the Console](#) on page 487
- [Creating and Configuring Workers from Console](#) on page 488
- [Opening the Scheduler GUI](#) on page 489

Creating Launchers from the Console

Launcher objects are SeaScape data structures that instruct the system about how to launch new workers.

1. To create a launcher object, select **Workers>Create Launcher** from the console menu.



The following window opens.

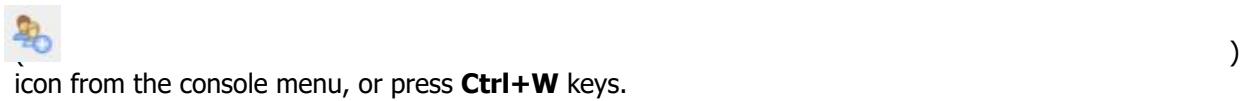


2. Optionally,
 - To select the type of launcher, select one of **local**, **grid**, or **ssh**.
 - Specify the **name** of the launcher. Entries in **bold** are required. Else the tool generates an error.
 - Input the **command** specification to use to launch the workers.
 - Specify the number of **workers_per_launch**.
3. Click **Create Launcher**.

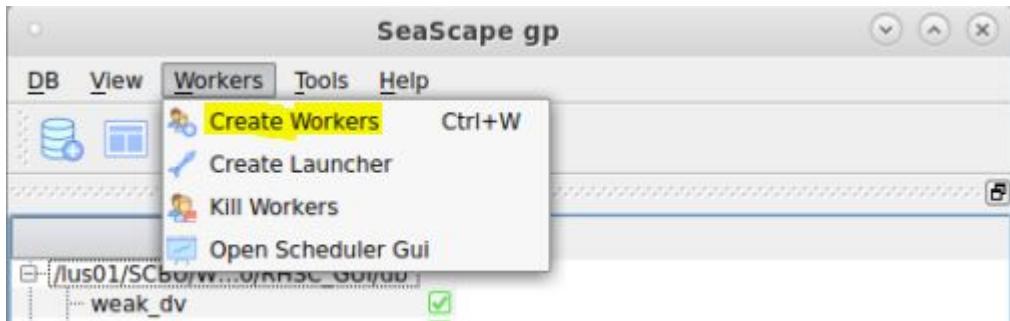
Creating and Configuring Workers from Console

Once a launcher is created, you can launch workers from the console.

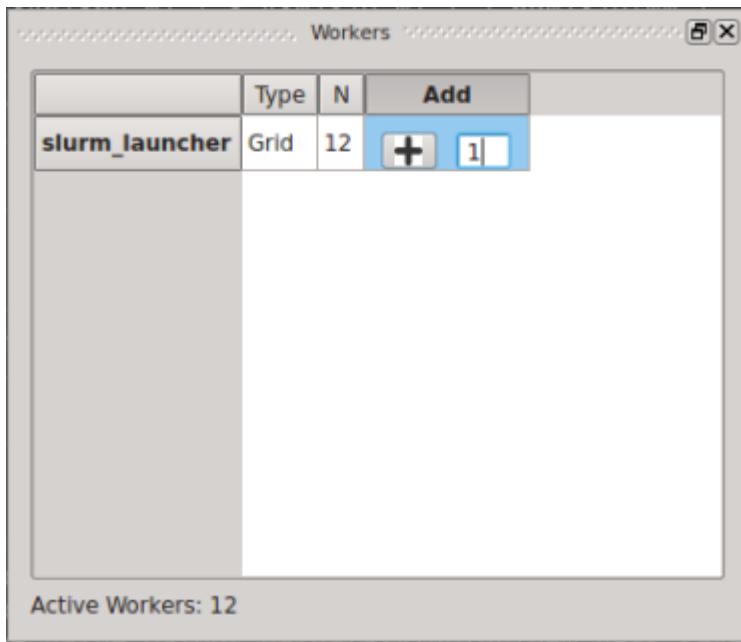
1. To create workers, select **Workers>Create Workers** or click the **Create Workers**



icon from the console menu, or press **Ctrl+W** keys.



The following **Workers** window opens. The window displays all the created launchers with respective **Type** and **N** (number of workers).



2. To change the number of workers per launcher, specify the number of workers under the **Add** column.
3. To launch the workers, click the **+** button under the **Add** column.

Opening the Scheduler GUI

To open the Scheduler GUI to track workers from the console menu, see [Invoking the Scheduler GUI](#) on page 441.

The console **Workers** menu also includes the **Kill Workers** option to kill any launched workers.

16.5. Using the Script Creation Wizard

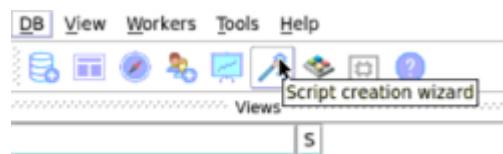
Invoking the Script Creation Wizard

The RedHawk-SC console includes the **Script creation wizard** that enables you to quickly create custom queries in script files that you can save and reuse.

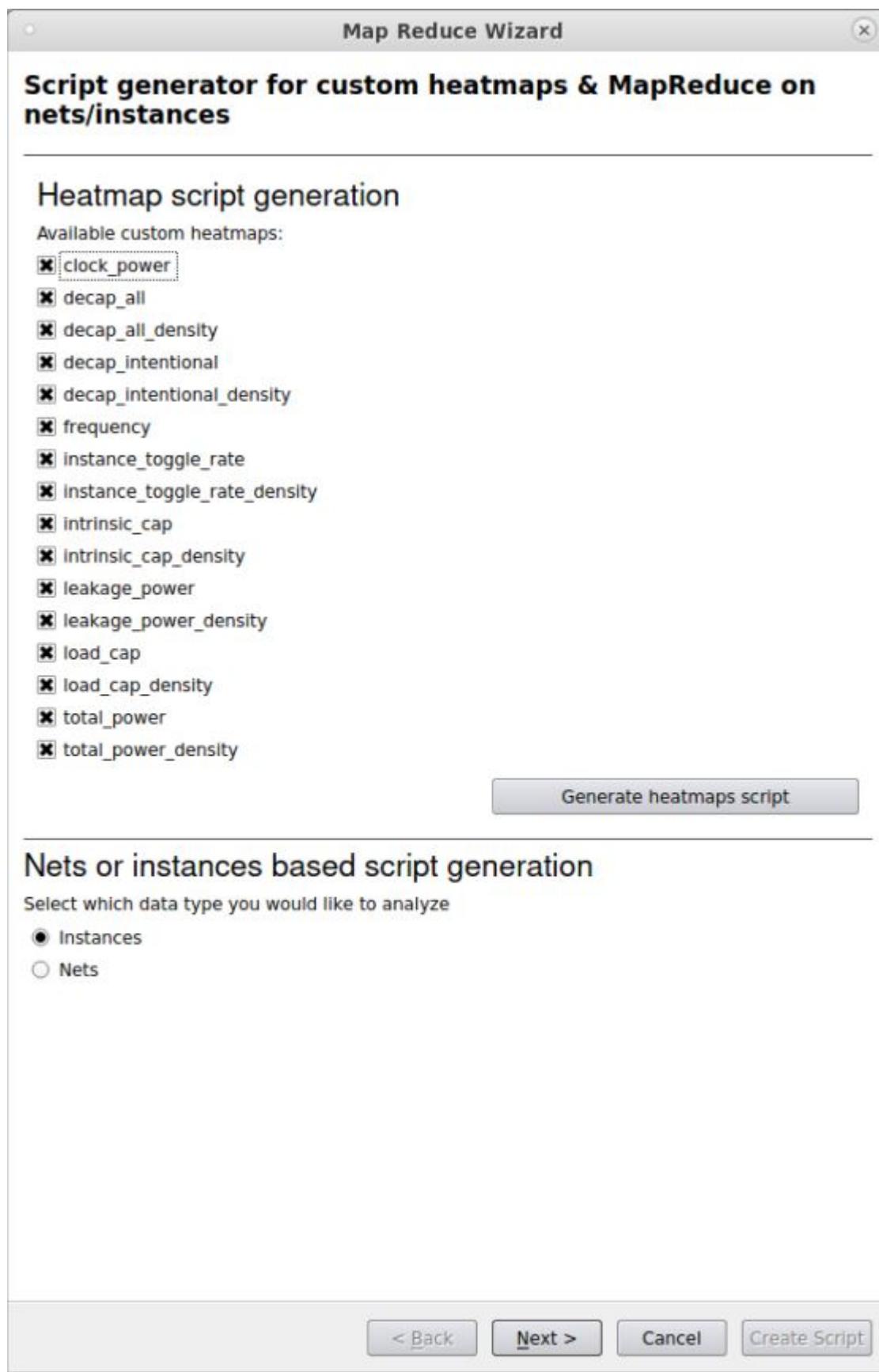
1. To invoke the Script creation wizard, select **Tools>Script creation wizard** or click the **Script creation wizard**



icon from the console menu, or press the **Ctrl+M** keys:



This opens the **Map Reduce Wizard** dialog box and displays a list of heatmap script names that are readily available for your use.



2. Click **Generate heatmaps script**. For detailed query construction, follow the GUI prompts.

16.6. Using RedHawk-SC Help

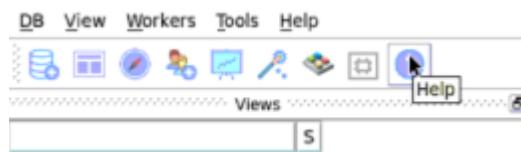
The RedHawk-SC tool includes a help system that you can access from the console.

Invoking Help

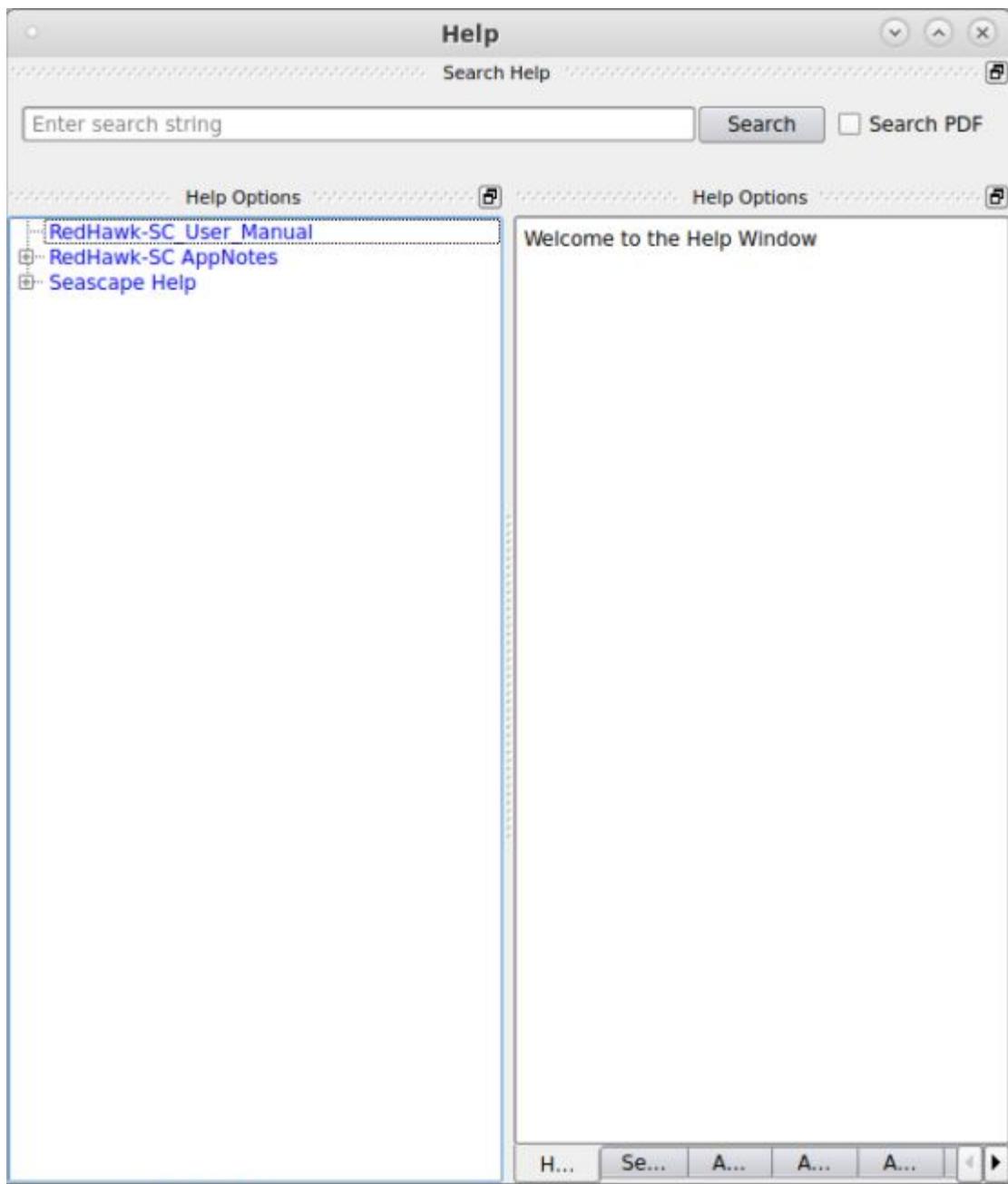
To invoke tool help, select **Help>Help** or click the **Help** icon



from the console menu or press **Ctrl+H** keys.



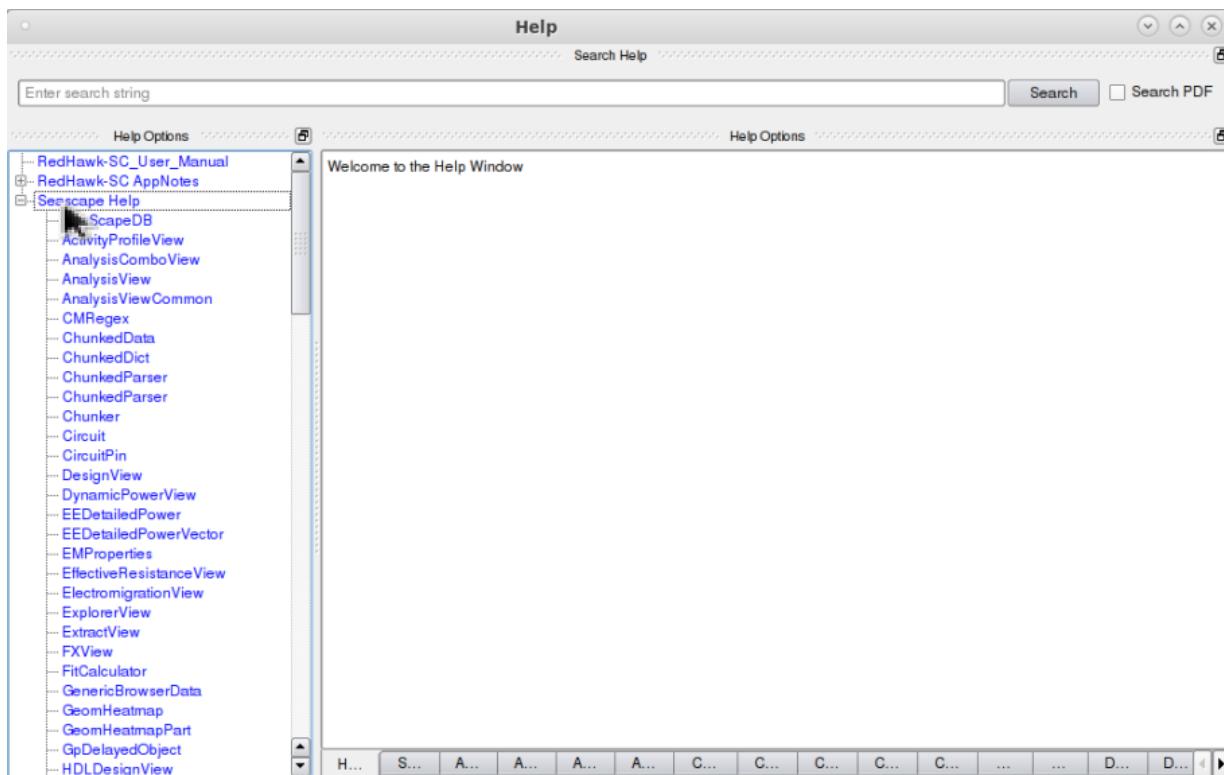
This opens the **Help Browser**. Use the **Search** field to search for help topics including application notes.



Viewing Command Help From Console

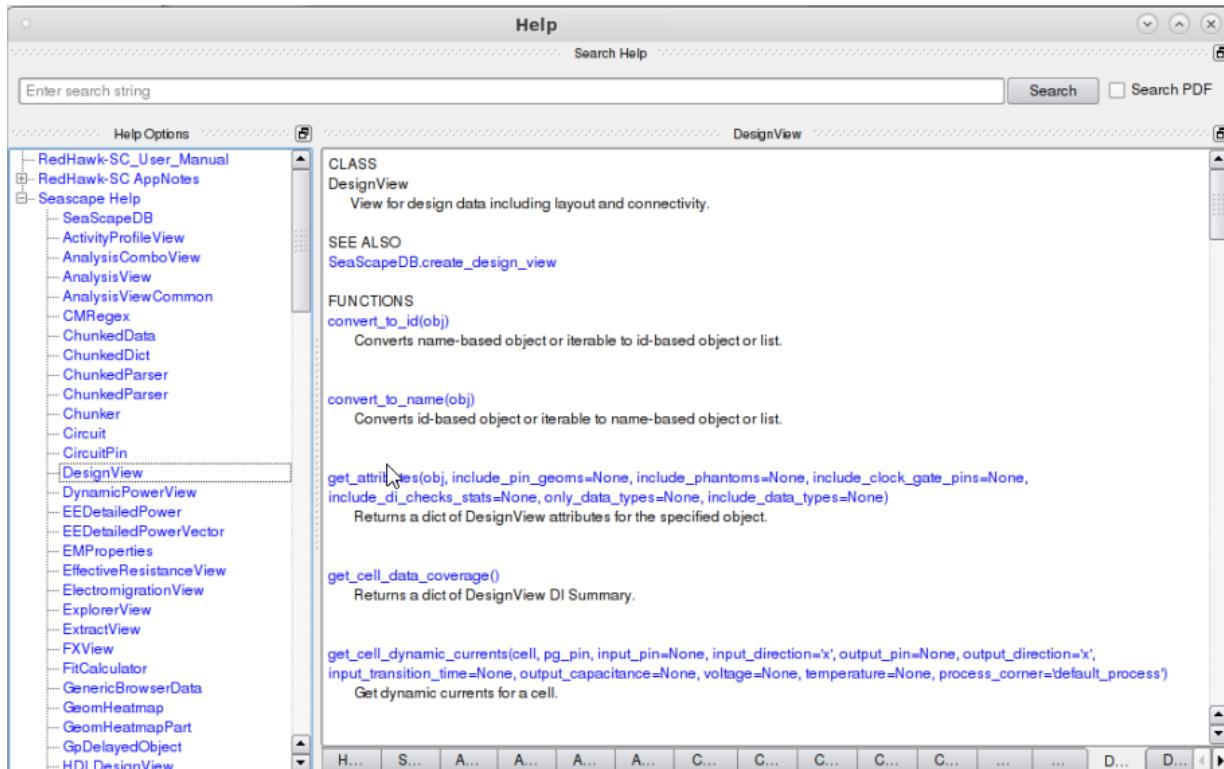
1. To view command help, open the **Help Browser** as described in [Invoking Help](#) on page 491.
2. Click the + sign to the left of **Seascape Help** under the left **Help Options** panel.

This opens the list of views for which help is available, as shown in the following figure.



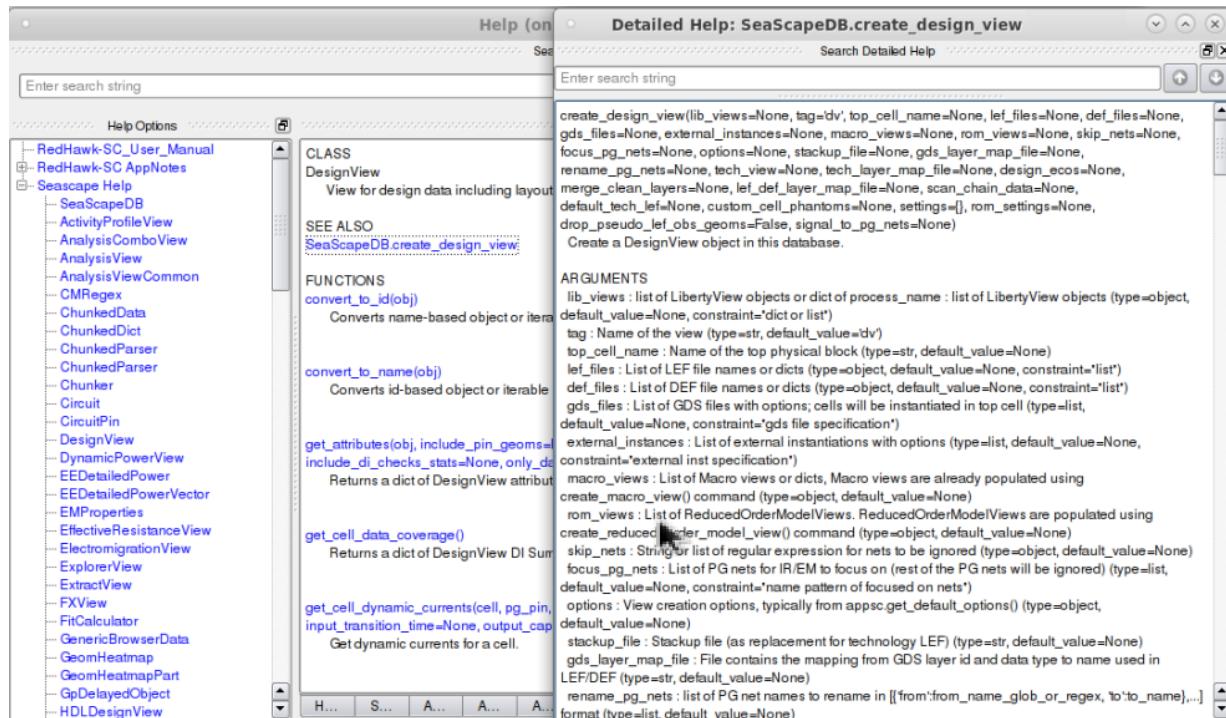
3. Click the view, for example, **DesignView.**

This opens the help page containing relevant commands and functions for view creation, query, and reporting as shown in the following figure.

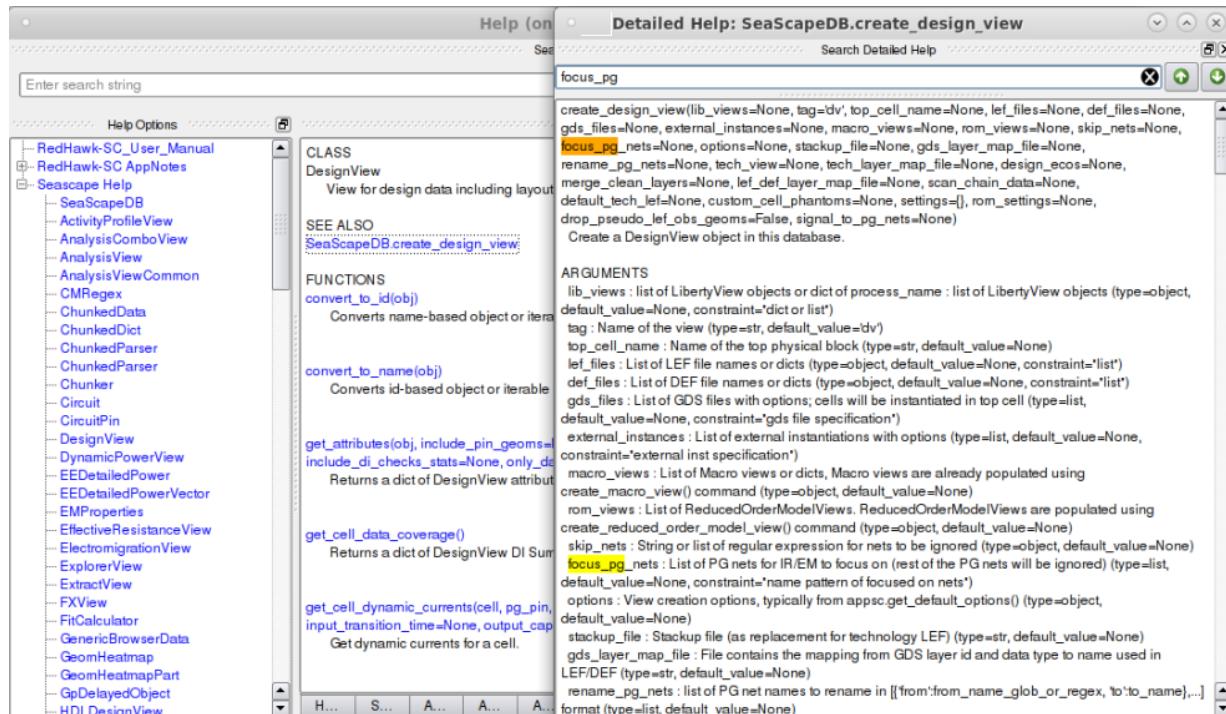


4. Click a command hyperlink to view the command help, for example, **SeaScapeDB.create_design_view.**

This opens the **Detailed Help** browser of the command as shown in the following figure.



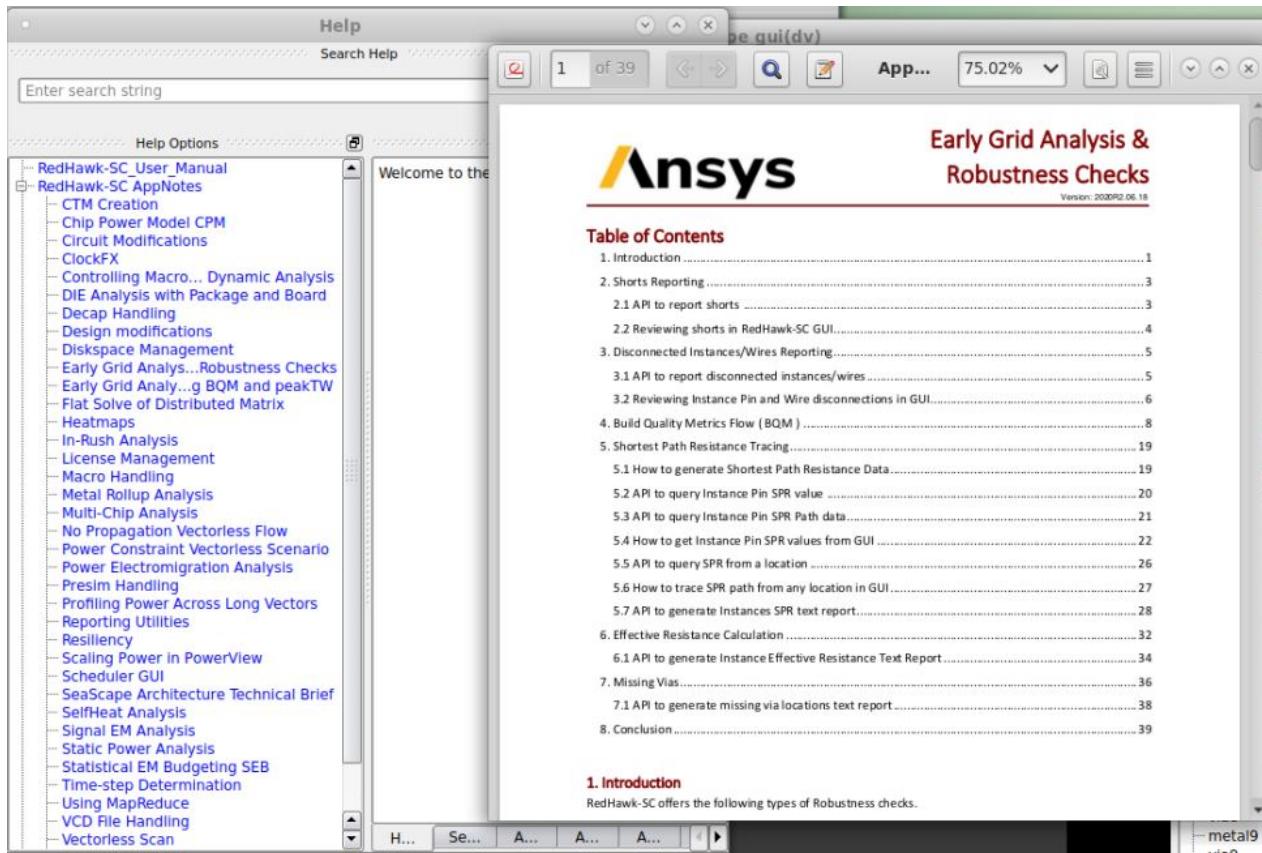
- To search for a specific term, such as, a key or an argument of a command, type the text in the **Search Detailed Help** field, as shown.



Viewing Application Notes

To view application notes, click the + sign to the left of **RedHawk-SC AppNotes** under the left **Help Options** panel.

This displays the list of available application notes for you to open.



17.1.1. Copyright and Trademark Information

© 2022 ANSYS, Inc. Unauthorized use, distribution or duplication is prohibited.

ANSYS, ANSYS Workbench, AUTODYN, CFX, FLUENT and any and all ANSYS, Inc. brand, product, service and feature names, logos and slogans are registered trademarks or trademarks of ANSYS, Inc. or its subsidiaries located in the United States or other countries. ICEM CFD is a trademark used by ANSYS, Inc. under license. CFX is a trademark of Sony Corporation in Japan. All other brand, product, service and feature names or trademarks are the property of their respective owners. FLEXIm and FLEXnet are trademarks of Flexera Software LLC.

Disclaimer Notice

THIS ANSYS SOFTWARE PRODUCT AND PROGRAM DOCUMENTATION INCLUDE TRADE SECRETS AND ARE CONFIDENTIAL AND PROPRIETARY PRODUCTS OF ANSYS, INC., ITS SUBSIDIARIES, OR LICENSORS. The software products and documentation are furnished by ANSYS, Inc., its subsidiaries, or affiliates under a software license agreement that contains provisions concerning non-disclosure, copying, length and nature of use, compliance with exporting laws, warranties, disclaimers, limitations of liability, and remedies, and other provisions. The software products and documentation may be used, disclosed, transferred, or copied only in accordance with the terms and conditions of that software license agreement

ANSYS, Inc. and ANSYS Europe, Ltd. are UL registered ISO 9001: 2015

U.S. Government Rights

For U.S. Government users, except as specifically granted by the ANSYS, Inc. software license agreement, the use, duplication, or disclosure by the United States Government is subject to restrictions stated in the ANSYS, Inc. software license agreement and FAR 12.212 (for non-DOD licenses).

Third-Party Software

See the **legal information** in the product help files for the complete Legal Notice for ANSYS proprietary software and third-party software. If you are unable to access the Legal Notice, contact ANSYS, Inc.

Published in the U.S.A.

