IC Validator
Deck Creation and Validation Toolkit
User Guide

Version M-2016.12, December 2016

SYNOPSYS®

Synopsys, Inc.
690 E. Middlefield Road
Mountain View, CA 94043
www.synopsys.com

Contents

About This User Guide vi

Customer Support viii

1. Introduction to Deck Creation and Validation

DCV Toolkit 1-2

2. DCV Analyzer Tool

Overview 2-2

Running the Tool 2-3
- Command-Line Options 2-4
- Environment Variables 2-6

Analyzing Performance 2-7
- Summary Report File 2-9
- User-Functions File 2-15
- Table Report Files 2-16
- Command Chain Files 2-16

Analyzing and Comparing Hierarchies 2-18
- Tree Summary Files 2-20
- Tree Comparison File 2-23
- Cell Information Files 2-24
- Layer Information Files 2-25

3. DCV Switch Optimizer Tool

Overview 3-2

Command-Line Options 3-3

Runset Switch Description File 3-3

Mutually Exclusive Switch Cases 3-5

Default and Default List Switch Cases 3-5

Ignore Switch Cases 3-6

Error Output Log Files 3-7

Summary Report File 3-7

4. DCV Results Compare Tool

Overview 4-2

Command-Line Options 4-3

Running the Tool 4-5

Setting the Comparison Type 4-6

Input File Formats 4-9

Specifying Custom Rule Mappings 4-9

Including or Excluding Rules 4-10

Waiving Errors With an Error Classification Database 4-10

Rule-Map File 4-11

Comparison Results File 4-14

Report Header 4-14

Comparison Summary 4-15

Detailed Comparison Results 4-16

Preface

This preface includes the following sections:

- About This User Guide
- Customer Support

About This User Guide

This user guide describes the Deck Creation and Validation (DCV) Toolkit utilities for the IC Validator tool.

Audience

This user guide is designed to enhance the knowledge that both beginning and advanced IC Validator users have of the DCV Toolkit.

Related Publications

For additional information about the IC Validator tool, see the documentation on the Synopsys SolvNet® online support site at the following address:

https://solvnet.synopsys.com/DocsOnWeb

You might also want to see the documentation for the following related Synopsys products:

- Custom Compiler™
- IC Compiler™
- IC Compiler™ II
- StarRC™

Release Notes

Information about new features, enhancements, changes, known limitations, and resolved Synopsys Technical Action Requests (STARs) is available in the *IC Validator Release Notes* on the SolvNet site.

To see the *IC Validator Release Notes*,

1. Go to the SolvNet Download Center located at the following address:

   https://solvnet.synopsys.com/DownloadCenter

2. Select IC Validator, and then select a release in the list that appears.

Conventions

The following conventions are used in Synopsys documentation.

Convention	Description
`Courier`	Indicates syntax, such as `write_file`.
`Courier italic`	Indicates a user-defined value in syntax, such as `write_file` *`design_list`*.
`Courier bold`	Indicates user input—text you type verbatim—in examples, such as `prompt>` **`write_file top`**
[ ]	Denotes optional arguments in syntax, such as `write_file [-format` *`fmt`*`]`
...	Indicates that arguments can be repeated as many times as needed, such as *`pin1 pin2 ... pinN`*
|	Indicates a choice among alternatives, such as `low | medium | high`
Ctrl+C	Indicates a keyboard combination, such as holding down the Ctrl key and pressing C.
\	Indicates a continuation of a command line.
/	Indicates levels of directory structure.
Edit > Copy	Indicates a path to a menu command, such as opening the Edit menu and choosing Copy.

Customer Support

Customer support is available through SolvNet online customer support and through contacting the Synopsys Technical Support Center.

Accessing SolvNet

The SolvNet site includes a knowledge base of technical articles and answers to frequently asked questions about Synopsys tools. The SolvNet site also gives you access to a wide range of Synopsys online services including software downloads, documentation, and technical support.

To access the SolvNet site, go to the following address:

https://solvnet.synopsys.com

If prompted, enter your user name and password. If you do not have a Synopsys user name and password, follow the instructions to sign up for an account.

If you need help using the SolvNet site, click HELP in the top-right menu bar.

Contacting the Synopsys Technical Support Center

If you have problems, questions, or suggestions, you can contact the Synopsys Technical Support Center in the following ways:

- Open a support case to your local support center online by signing in to the SolvNet site at https://solvnet.synopsys.com, clicking Support, and then clicking “Open A Support Case.”
- Send an e-mail message to your local support center.
  - E-mail support_center@synopsys.com from within North America.
  - Find other local support center e-mail addresses at

    http://www.synopsys.com/Support/GlobalSupportCenters/Pages
- Telephone your local support center.
  - Call (800) 245-8005 from within North America.
  - Find other local support center telephone numbers at

    http://www.synopsys.com/Support/GlobalSupportCenters/Pages

1

Introduction to Deck Creation and Validation

This chapter provides an introduction to IC Validator Deck Creation and Validation (DCV) Toolkit.

The DCV Toolkit is a set of utilities that can help you with the process of creating and validating DRC, LVS, and Fill runsets. You can configure and execute each tool, review the results, and validate or improve the quality and performance of your runsets.

For more information, see the following section:

- DCV Toolkit

DCV Toolkit

The DCV toolkit currently consists of the following tools:

- DCV Analyzer

  Use this tool to analyze the performance of an IC Validator run. The tool creates organized reports that can help you to identify factors limiting overall performance.

  For more information, see Chapter 2, “DCV Analyzer Tool.”

- DCV Switch Optimizer

  This tool helps you to validate the syntax correctness of an IC Validator runset. The tool identifies and exercises a minimal set of runset preprocessor flow-control switches that exercise all of the code in the runset, and then reports any combinations that produce a runtime parser error.

  For more information, see Chapter 3, “DCV Switch Optimizer Tool.”

- DCV Results Compare

  This tool helps you to validate the quality of IC Validator output results. The tool compares two sets of error output results and identifies any discrepancies between them.

  For more information, see Chapter 4, “DCV Results Compare Tool.”

2

DCV Analyzer Tool

This chapter explains how to run the DCV Analyzer tool and view the reports.

The IC Validator DCV Analyzer tool serves as a starting point for analyzing the performance of an IC Validator run.

For more information, see the following sections:

- Overview
- Running the Tool
- Analyzing Performance
- Analyzing and Comparing Hierarchies

Overview

The DCV Analyzer tool takes the output of an IC Validator run and displays the data in a clear, organized way. In some cases, this data is equivalent to the data created by other sources, such as the scripts in the contrib directory; however, the DCV Analyzer tool uses a more predictable method to create and present the data. In other cases, the tool creates novel data that is not available elsewhere. In either case, you should use this tool as the starting point for any type of IC Validator performance analysis.

You can use the DCV Analyzer tool to

- Analyze performance using the distributed log file from an IC Validator run
- Analyze hierarchy using a tree structure file from an IC Validator run
- Compare hierarchies using two tree structure files from the same or different IC Validator runs

You can perform each of these tasks individually or perform any combination of them in a single DCV Analyzer run. The tool determines which operations to perform based on the input files that you specify.

- Distributed log files contain information about the commands that IC Validator executed during a run.
- Tree structure files contain information about the design hierarchy tree.

These IC Validator output files are located in the run_details directory. For more information about the files, see the *IC Validator User Guide*.

By default, the DCV Analyzer tool takes input from the files that you specify, stores the data in an SQL database, analyzes the data, and provides output in report files. In general, the tool performs the following operations:

1. Parses the distributed log file or tree structure files and loads the input data into an SQL database
2. Analyzes the data from the SQL database and generates reports in the output files

In subsequent runs, the tool can reanalyze the data and regenerate or add reports from the same SQL database.

For performance analysis, the tool provides output in a summary report file and a user-functions file. The summary report file can contain runtime statistics, efficiency metrics, and various tables highlighting discrete aspects of the run, such as the runset functions that took the longest or used the most memory.

Optional output can include individual table report files, in tab-separated-value format, that you can import into a third-party tool, such as the Microsoft® Excel® tool, for further analysis.

You can also generate command chain files for long-pole analysis. In addition, you can customize the number of table rows in the reports.

For hierarchy analysis, the tool provides output in a tree summary report file named tree_summary.txt. Optional output can include cell and layer report files. For hierarchy comparisons, the tool provides additional output in a tree comparison file named tree_compare.txt.

Running the Tool

Before running the DCV Analyzer tool, make sure your `LM_LICENSE_FILE` and `ICV_HOME_DIR` environment variables are set. The `dcv_analyzer` executable is located in the same bin directory as the `icv` executable. The tool creates the database and report files in your current working directory.

The DCV Analyzer command-line syntax is

```
dcv_analyzer [options] [runset.dp.log] [top-cell.treeX] [top-cell.treeX]
```

where

- *options* are described in Table 2-1 on page 2-4
- *runset*.dp.log is the name of the distributed log file from an IC Validator run
- *top-cell*.tree*X* specifies the name of a tree structure file from an IC Validator run

  You can specify one tree structure file for hierarchy analysis or two tree structure files to compare the hierarchies. The files can come from the same or different IC Validator runs. In the tree structure file names, *X* is an integer from 0 to 999. The format of the tree structure files (Milkyway, NDM, LTL, GDSII, or OASIS) does not matter.

For more information, see the following sections:

- Command-Line Options
- Environment Variables

Command-Line Options

Table 2-1 describes the command-line options.

Table 2-1 DCV Analyzer Command-Line Options

Option	Description
`-c` *`count`* `| all`	Specifies the number of records for fields in the report. By default, the tool generates 10 records for most fields. The *count* must be a positive integer. Use `all` if you want the tool to return all records by the sort order.

`-C` *`cell-list`*
`--cell` *`cell-list`*
	Specifies a comma-separated list of cell names in *cell-list*. The tool provides cell statistics for each cell in cell information files named cell_*cellname*.txt. At least one tree structure input file (*top-cell*.tree*X*) must be provided.
	You can use the asterisk wildcard character (*) to specify multiple cells in a name pattern. When using wildcard characters, you should enclose the entire argument in quotation marks to prevent the shell from interpreting files in the current working directory.

`-e` `--export`	Provides the output data in table-based report files named *_table.tsv. You can import these tab-separated-values files into the Microsoft Excel tool for further analysis.
`-h` `--help`	Displays the usage message and exits.

`-l` *`command_key`*
`--long-pole-hist` *`command_key`*
	Specifies a comma-separated list of the command keys (for example, 1234.0.0) used to generate the long-pole history. If the first element in a *command_key* is a positive integer, the tool produces that number of worst-case graphs. The output file name format is cmdchain_.*command_key*.txt. Each command chain file shows the longest serial path of dependencies to reach the command key.

Table 2-1 DCV Analyzer Command-Line Options (Continued)

Option	Description

`-L` *`layer-list`*
`--layer` *`layer-list`*
	Specifies layers in a comma-separated list of layer names, layer number and data type pairs, or both. The tool provides layer statistics for each layer in layer information files named layer_*layer-name*.txt for each layer name and *layer_num*:*data_type*.txt for each layer number and data type pair.
	You must provide at least one tree structure input file (*top-cell*.tree*X*) when you use this option. If *layer-list* contains layer number and data type pairs, the tree structure input file must be a single *top-cell*.tree0 file.
	You can use the asterisk wildcard character (*) to specify multiple layers in a layer name pattern. When using wildcard characters, enclose the entire argument in quotation marks to prevent the shell from interpreting files in the current working directory.
	Wildcard characters are not allowed in a layer number and data type pair. Specify each pair by using the form *`layer-number`*`:`*`data-type`*, where *layer-number* and *data-type* are integers.
	For example, `“M1,1:0,D*”` specifies layer name M1, layer number and data type pair 1:0, and layer name pattern D*. The quotation marks prevent the shell from expanding D*.

`-r` `--report-only`	Regenerates the reports using the data in the SQL database from a previous run instead of parsing and analyzing the data from the distributed log file. At least one input file (dplog.db or tree.db) must be present in the current working directory.
`-t` `--tabs`	Changes the output file name extensions to .tsv and the file format to tab separation instead of fixed-width printing. This option allows you to more easily import an output file into the Microsoft Excel tool or to parse specific fields in a file by using a script.

`-v` *`violation-list`*
`--violation-cmdchain` *`violation-list`*
	Produces the same cmdchain_.*command_key*.txt files that the `-l` command-line option produces, but uses the *violation-list* comments as the input. Enclose each comment in quotation marks and separate the comments with commas. For example, `"*a*","b*"` (where `*` is a wildcard character).

`-V` `--version`	Prints the tool version.

Table 2-1 DCV Analyzer Command-Line Options (Continued)

Option	Description

`-x` *`XREF`*
`--xref` *`XREF`*
	Changes the content of the field used to cross-reference the SQL database. *XREF* can be `key`, `number`, or `line`. The default is `key`.
	• `key` prints the command key
	• `number` prints the command number
	• `line` prints the line number from the dp.log file
	In the following example, line 7105 from a distributed log file contains the command key 1337.0.0 and the command number 3163:
	`Host teralab-48(0) completes 1337.0.0, 3163:`

Environment Variables

You can control the appearance of the standard output and log files by setting the following DCV Analyzer environment variables:

- `DCV_GROUP_LOOP`

  Set this variable to `1` if you want to disable the unrolling of loop data for user function input. You must set this variable before parsing the data. This variable has no effect when you specify the `-r` command-line option.

  Setting `DCV_GROUP_LOOP` affects the results of all user-function-related data.

- `DCV_HEADER_FORMAT`

  Set this variable to control the alignment of the Run Time Summary, Run Information, and Efficiency Metrics sections in the dplog_summary.txt file. The valid settings are `PINCH`, `SPLIT`, `LEFT`, `RIGHT`, and `CENTERED`. The default is `CENTERED`.

- `DCV_PROGRESS_SECONDS`

  Set this variable to change the frequency of Progress Line updates. The value must be an integer. The default is `60`.

- `DCV_PROGRESS_PERCENT`

  Set this variable to change frequency of Progress Line percentage updates. The value must be an integer. The default is 10.

- `DCV_UNICODE`

  Set this variable to `1` to allow the DCV Analyzer tool to print Unicode characters in the output report files. When you set this variable, the tool replaces `+INF` with the infinity character (∞).

  Note:
  Before setting this variable, make sure that everything you are using (shell, display, text editor, and so forth) supports Unicode characters.

The `DCV_PROGRESS_SECONDS` and `DCV_PROGRESS_PERCENT` variables work together logically. So for example, if you set the seconds to `60` and the percent to `50`, and the run takes approximately three minutes, you should expect updates at 60 seconds, at 90 seconds (due to the 50 percent setting), at 120 seconds, and at 180 seconds.

In addition, setting these variables to artificially high values (more seconds than you think the job should take or a percentage equal to 100 or more) effectively disables the timer.

Analyzing Performance

For performance analysis, the DCV Analyzer tool takes input from a distributed log file that you specify and stores the data in an SQL database, named dplog.db, in the current working directory. Then, the tool analyzes the data and generates reports that it saves in output files.

By default, the tool provides summary information for performance analysis in a summary report file, named dplog_summary.txt, and detailed information about the user functions from the runset in a user-functions file, named userfunction_summary.txt. The summary report file can contain runtime statistics, efficiency metrics, and various tables highlighting discrete aspects of the run, such as the runset commands that took the longest or used the most memory.

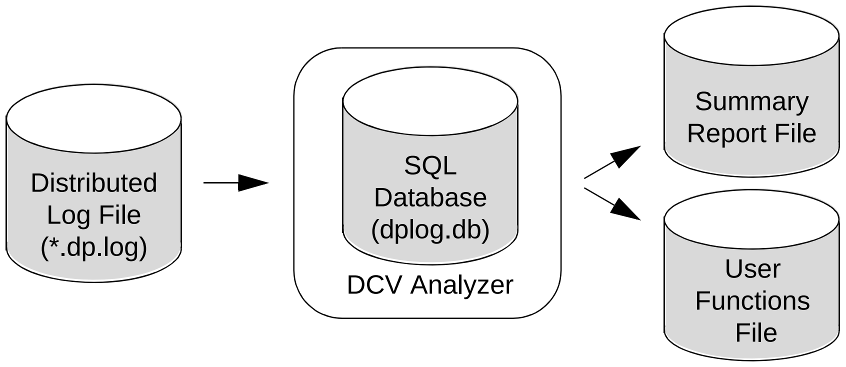
For example, you can enter the following command:

```
# dcv_analyzer runset1.dp.log
```

The tool generates the SQL database, the summary report file, and the user-functions file in the current working directory. Most of the runtime involves creating the SQL database. For subsequent runs with the same runset data, you can use the `-r` command-line option to reanalyze the data and regenerate or add reports from the same SQL database.

Figure 2-1 illustrates this flow.

Figure 2-1 DCV Analyzer Performance Analysis Flow



For example, to increase the number of table rows to 15 in the summary report file, you can enter

```
# dcv_analyzer -c 15 -r
```

The tool can generate additional output files depending on the options that you include on the `dcv_analyzer` command line.

- Table report files

  Use the `-e` command-line option to generate table-based report files, named *table-name*_table.tsv, that provide data from the database tables. You can import these tab-separated values files into the Microsoft Excel tool for further analysis.

- Command chain files

  Use the `-l` command-line option to specify the command keys that the tool uses to generate the long-pole history. The tool generates an output file for each command key. These files, named cmdchain.*command_key*.txt, show selected information about the command-chain critical paths to the specified command keys.

For example, to find information about command key 1234.0.0, export the table-based reports in tab-separated-value files by entering the following command:

```
% dcv_analyzer -l 1234.0.0 -e -r
```

For more information about the performance analysis output files, see the following sections:

- Summary Report File
- User-Functions File
- Table Report Files
- Command Chain Files

Summary Report File

The summary report file is the main DCV Analyzer output file. This file contains the performance information highlighted by each run.

The top of this report contains summary information about

- The method and tool versions used to run the DCV Analyzer and IC Validator tools
- The elapsed time and the sum of the elapsed, user, and system times
- The peak and average memory experienced by each host, the number of CPUs used, and the total physical memory available on the host

  The average memory is calculated by adding, for each aperiodic interval, the peak memory divided by the interval period. The memory is reported in gigabytes (GB).
- Run information that shows many basic statistics for the run

The IC Validator tool reports time measurements in elapsed time, check time, user time, and system time. Table 2-2 describes these time measurements.

Table 2-2 IC Validator Time Measurements

Type	Description
Elapsed time	The time that you can measure with a clock while the job runs (also referred to as the “wall time”). The tool retrieves elapsed times from an independent timer that prints timestamps during the run. The Sum of Command Elapsed Time value is the sum of all the individual commands. For a single-CPU run, the elapsed time is equal to the sum of elapsed times. For multiple-CPU runs, parallel processing makes the sum of elapsed times greater than the IC Validator elapsed time.
Check time	The time measured by a command and reported as `Check Time=`. Most commands report the check time; however, some commands use a different descriptor instead of `Check` and some commands do not report this time. In most cases, the check and elapsed times should be within one second of each other, making them effectively the same.
User time	The time spent by the CPU doing work for the IC Validator tool. Threading makes this number higher than the elapsed time for a single command. These times are reported as `User Time=` for each command. The Sum of Command User Time value is the sum of this number for all commands.

Table 2-2 IC Validator Time Measurements (Continued)

Type	Description
System time	The time reported as `Sys=` for each command. System times represent the amount of time the operating system spends helping the IC Validator tool, for example, to copy a file. In general, system times are relatively low. High system times usually indicate that machine resource contention is affecting the IC Validator runtime. For example, the machine might be paging from high memory usage, or the CPU load average might exceed the amount of available processors on the machine. These problems can exist either because of the current IC Validator run or because of unrelated processes running on the machine.

In addition to the runtime and memory, the basic statistics for an IC Validator run can include information about the parse time, assigned layers, number of commands used in the run, number of DRC rules, and disk statistics. Table 2-3 describes these IC Validator run statistics.

Table 2-3 IC Validator Run Statistics

Type	Description
Parse time	The amount of time the IC Validator tool took to parse the runset. (Cache hits are shown in parentheses.)
Non-empty assign layers	Shows the count of layers in `assign()` function calls that contain at least one object.
Assign layer objects	The sum of all the data read in from `assign()` function calls.
Executed engine commands	A count of the discrete processes the engine performed to complete the run. This count includes the generic processes that are independent of the runset and all of the processes directed by the runset. The same runset can produce different numbers depending on the command-line options that you use to run the IC Validator tool (the number of CPUs) and any preprocessor directives or other flow-control and empty-layer optimizations that the tool performs during the run.
Longest Command Chain	The longest serial path of dependencies in the run. This generally depicts the runtime performance threshold.

Table 2-3 IC Validator Run Statistics (Continued)

Type	Description
Number of rules executed	Shows the number of unique text strings that can potentially write data to the LAYOUT_ERRORS file. A rule in PXL notation is defined as: `Rule1 @= { @ "some-description"; pxl_function( layers, constraints, options); }` These rules do not have to be proper DRC rules. They can be ERC checks or just debug statements that you include in the runset.
Number of rules with violations	Shows the number of rules in the runset that produced violations.
Error-producing commands	The number of engine commands with the following comment in the distributed log file: `Comment: “Some description”` The number of error-producing commands might exceed the number of rules due to duplicate comments in the runset, looping, or other engine optimizations that can occur for more complex commands.
Violation	Any output shape reported by the error-producing commands. The DCV Analyzer tool shows counts for both the rules and error-producing commands and the total violations found.
Disk usage	Reports the amount of space at the peak and end of the IC Validator run.

All efficiency metrics should be well defined by the equations used in the file to derive them and should be consistent with the terms described in Table 2-3.

Example 2-1 shows an example of the summary information at the top of the summary report file.

Example 2-1 Summary Information Example

```
DCV_Analyzer, L-2016.06 2016/05/08
Called as: dcv_analyzer ../run/run_details/analyzerDocs.dp.log

Summary for analysis of
/remote/DCV/lab/DCV_Analyzer_lab/run/run_details/analyzerDocs.dp.log

dp.log Header Information:
-------------------------
  Version L-2016.06 for linux64 - May 08, 2016 cl#3129124
  Called as: icv -f OASIS ../input_files/analyzerDocs.rs
```

```
Run Time Summary:
----------------
                      ICV Elapsed Time: 1.8 hours
              Sum of Command User Time: 3.2 hours
           Sum of Command Elapsed Time: 3.5 hours
            Sum of Command System Time: 0.2 hours

Host Information:
----------------
                               |            Memory (GB)             |
Host Name                 CPUs |  Average     Peak  Physical   Swap | Architecture
               Allocated/Total |     Used     Used Available  Space | Information
-------------- --------------- | -------- -------- --------- ------ | ----------------
attoemt606                1/60 |      0.5      4.8   1,009.7  200.0 | Arch1
attoemt608                1/20 |      1.2      8.9     504.8  200.0 | Arch2
-------------- --------------- | -------- -------- --------- ------ | ----------------
Arch2 (1)                 1/20 |                                    | Intel(R) Xeon(R)
                                                                      CPU E5-2690
                                                                      v2 @ 3.00GHz
Arch1 (1)                 1/60 |                                    | Intel(R) Xeon(R)
                                                                      CPU E7-4890
                                                                      v2 @ 2.80GHz
--------------- -------------- | -------- -------- --------- ------ | ----------------
Total (2)                 2/80 |                                    |


Run Information:
-----------------
                                             Parse Time (seconds): 4 (cached)
                                          Non-Empty Assign Layers: 24
                                             Assign Layer Objects: 280,165,130
                                         Executed Engine Commands: 574
                                    Longest Command Chain (hours): 0.5
                                         Number of Rules Executed: 186
                                  Number of Rules with Violations: 25
                                         Error-Producing Commands: 192
                         Error-Producing Commands with Violations: 25
                                                 Total Violations: 17,969,374
                                             Peak Disk Usage (GB): 8.575
                                            Final Disk Usage (GB): 0.610

Efficiency Metrics:
--------------------
                          Longest Command Chain / ICV Elapsed Time: 25%
       CPU Utilization = User Time / (#Threads * ICV Elapsed Time): 88%
            Sum of Command User Time / Sum of Command Elapsed Time: 91%
              Sum of Command User Time / Sum of Command Check Time: 92%
           Sum of Command Check Time / Sum of Command Elapsed Time: 98%
                                       Average All-Hosts Memory (GB): 1.73
```

The summary information can also include a Qualitative Analysis section that appears after the Efficiency Metrics section if the tool finds any of the following diagnostic conditions during the run. Consider the messages in this section first when analyzing the run.

- The number of allocated CPUs exceeds the number of available CPUs on the host
- The amount of memory used on a host exceeds the amount available

- Restarted commands exceed one percent of the elapsed time on all hosts
- The log file contains a message about the disk filling up

  If this happens without messages in the log file, the DCV Analyzer tool cannot detect it.

The main part of the report contains a series of tables that show detailed information about specific parts of the run. The amount of data presented here is controlled by the `–c` command-line option. The default is 10.

The report contains two types of tables: command-key tables and grouped-row tables. Command-key tables highlight information about discrete commands and always contain a Command Key column. You can trace the data in this column to a specific instance of a command in the distributed log file. Grouped-row tables contain rows that are grouped for statistical analysis by a common quality, such as input layers or intrinsic types. These rows are aggregates between many commands and cannot be traced to specific instances.

The report can contain the following tables:

- Commands sorted by Check Time

  This table shows the discrete commands with the longest check times, as shown in Example 2-2.

Example 2-2 Table Highlighting Information About Discrete Commands

```
Commands sorted by Check Time:
-------------------------------------------------------------------------------------
Check Time User Time Memory  Runset                                            Command
    (min)^     (min)   (GB)  Text                                              Key
       0.3       0.3    0.2  large = external1(lay, distance <= 2 * dist, exte 475.0.0
       0.2       0.2    0.1  internal1(M1, < 0.14, extension = RADIAL, relatio 590.0.0
       0.1       0.2    0.1  m1e2_m1e31_x = enclose_edge(CO, M1, < 0.05, exten 677.0.0
       0.1       0.1    0.1  external1(M1, < 0.14, extension = RADIAL, relatio 591.0.0
       0.1       0.1    0.1  small = external1(lay, distance <= dist, extensio 474.0.0
       0.1       0.1    0.1  via1e21_y = enclose_edge(VIA1, M1, < 0.05, extens 687.0.0
       0.1       0.1    0.1  m1e2_m1e32_y = touching_edge(CO, m1e2_m1e31_x)    680.0.0
       0.1       0.1    0.1  -=_read_library_segment Intrinsic, RunsetText N/A 297.0.0
       0.1       0.1    0.1  output = internal_corner1(red, distance < 3.0, ty 482.0.0
       0.0       0.0    0.1  M2 = assign({ { 12 } })                           358.0.0
-------------------------------------------------------------------------------------
```

- Commands sorted by User Time

  This table shows the discrete commands with the longest user times.

- Commands sorted by Memory

  This table shows the discrete commands with the highest peak memory.

- Commands sorted by Work

  This table shows the discrete commands that use the largest combination of peak memory and user time. These commands use the most overall computer resources.

  ```
  Work = peak memory * user time
  ```

- Commands sorted by Gain

  This table shows commands in which the amount of output shapes have the largest proportional increase from the amount of input shapes to the command:

  ```
  Gain = Output-Objects / Input-Objects
  ```

  Downstream User Time is the amount of CPU time consumed by all commands that depend on the command in question. This is also expressed as a percentage of the Total User Time of the run.

- Error-Producing Commands sorted by Violation Count

  This table shows the error-producing commands that report the most violations based on the distributed log file. Because of hierarchy or error limit settings, the counts in this table might not match the counts in the *cell*.LAYOUT_ERRORS file. The long-pole times represent the slowest path through the dependent commands for each rule.

- Dangling Commands sorted by Long Pole Time

  This table shows the dangling commands, which are commands that do not have child commands and are not producing a violation comment or creating any data. Ideally, this table should not appear because these commands should be optimized away.

- Error-Producing Commands sorted by Long-Pole Time

  This table shows the error-producing commands that cause the run to go the slowest.

  Upstream User Time shows the total amount of CPU power required to process all the dependencies of the rule.

  Restarted Commands shows commands that the tool had to stop and restart in the given long-pole chain, most likely due to memory contention on the host, which the command started. Commands that are restarted retain the same command key.

- Data-Creating Commands sorted by Long-Pole Time

  This table shows the data-creating commands, which are commands without a violation comment that write a file to the output directory. This file can be a layout file, a database for the Synopsys StarRC™ tool, or an intermediate file for an LVS run.

- Error-Producing Commands sorted by Maximum Memory

  This table shows which error-producing commands have commands in their paths that consume the most memory.

- Slowest User Functions sorted by User Time
  Slowest User Functions sorted by Elapsed Time

  These two tables group runtimes by custom PXL functions defined in the runset. The tool sorts the tables based on the times used to group user functions defined by the runset coder. One table is sorted by the CPU (user) time; the other table is sorted by the elapsed time.

  Nested user functions show discrete numbers, not cumulative numbers. In other words, if user function A contains user function B, the effects of the IC Validator intrinsics in function B are not cumulative for the report of function A. The numbers indicate which line of the runset they come from.

  For information about limitations and deciphering the function name, see "User-Functions File" on page 2-15.

The Memory Stats section of the report shows all of the commands that ran for each host when the host reached its peak memory.

User-Functions File

The user-functions file, userfunction_summary.txt, contains detailed information about the user functions contained in the file. This information is discrete for the native intrinsics present in each function. Nested user functions show discrete numbers, not cumulative numbers.

Limitation:

Iterations through loops cannot be separated for IC Validator runs before version K-2015.06-SP1, so the statistics associated with any loop are inflated by the number of times the loop is run. You can achieve the same behavior for newer log files by setting the `DCV_GROUP_LOOP` environment variable to `1`. For more information, see "Environment Variables" on page 2-6.

Also, the use of encryption or the published keyword in the runset obfuscates details in the distributed log file, and the DCV Analyzer tool is unable to present all of the information.

The file shows the hierarchy of all user functions contained within a runset. Each row shows statistical information for all native intrinsics in a user function. To get totals, you must manually add the nested functions to the level of interest. The format is

```
function_name@runset_file_name:line_number.loop_iteration
```

Table Report Files

The table report files, named *table-name*_table.tsv, contain raw data from the database tables used for all of the queries. These files are meant to be used only for debugging or advanced analysis. You can import these tab-separated values files into the Microsoft Excel tool.

Command Chain Files

The command chain files show the longest serial path
of dependencies to reach a given command key. For each command, these dependencies include the

- Elapsed time
- Gap time

  The gap time is the start time of the current command minus the end time of the previous command. For the first command, the gap time is the same as the start time since the previous end time would be 0.
- Memory the command used
- Total of all the output geometric objects and the associated gain

The runset text, truncated at 75 characters with an ellipsis (...), and the parent command keys for each command in the chain appear on the right side of the file. Consider that input counts come from alternate parent commands, which are listed, and not just from the previous command key listed in the file. If you are interested in a parent command, you can regenerate its equivalent file by using the `-r` and `-l` command-line options in a subsequent run. The bottom of the file provides some basic statistical information relative to the chain.

Example 2-3 shows an example of a command chain file.

Example 2-3 Command Chain File Example

```
Command    Elapsed Gap Time Memory       Output Gain Runset                    Parent
   Keys Time (min)    (min)   (GB) Object Count      Text                Command Keys
297.0.0       4.21     0.08   0.31            0    0 -=_read_library_segment
                                                     Intrinsic, RunsetText N/A=
301.0.0       0.38     0.00   0.62            0    0 -=_create_hierarchy_tree
                                                     Intrinsic, RunsetText N/A=-
                                                                             297.0.0
355.0.0       2.77     0.16   0.75   30,222,080 +INF M1 = assign({ { 11 } })
                                                                     301.0.0 297.0.0
356.0.0       0.53     6.05   0.30   30,236,670 1.00 M1 = assign({ { 11 } })
                                                                     355.0.0 320.0.0
416.0.0       1.07     1.69   0.53      242,374 0.00 Generating associated layer
                                                     temp.associated
                                                                     386.0.0 383.0.0
                                                                     323.0.0 356.0.0
694.0.0      11.26    10.17   2.07  121,741,385 1.57 m1e2_m1e31_x = enclose_edge(
                                                       CO, M1, < 0.05,
                                                       extension = NONE,
                                                           look_thru ...
                                                                     425.0.0 356.0.0
                                                                     323.0.0 416.0.0
                                                                             383.0.0
696.0.0       0.74    10.96   0.71      186,539 0.00 Generating associated layer
                                                     m1e2_m1e32_y.associated
                                                                     694.0.0 323.0.0
                                                                             383.0.0
697.0.0       5.11     4.12   1.03  121,780,908 0.72 m1e2_m1e32_y =
                                                     touching_edge(
                                                       CO, m1e2_m1e31_x
                                                     )
                                                                     696.0.0 694.0.0
                                                                     695.0.0 323.0.0
                                                                             383.0.0
698.0.0       0.65     0.43   0.53       20,919 0.00 Generating associated layer
                                                     temp.associated
                                                                     697.0.0 323.0.0
699.0.0       0.80     0.00   0.16            0 0.00 length_edge(
                                                       m1e2_m1e32_y, < 0.001
                                                     )
                                                                     697.0.0 323.0.0
                                                                             698.0.0

Sum of Chain Elapsed Times:          27.52  minutes  (includes Restarted Times)
Sum of Chain Gap Times:              33.66  minutes
Sum of Chain Restarted Times:         0.00  minutes
Sum of Chain User Times:             26.17  minutes
```

The data in the command chain file is similar to the data in the summary report file. However, the command chain file is organized by execution time rather than by statistics. Locating issues created by linear dependencies might be easier in this format.

Analyzing and Comparing Hierarchies

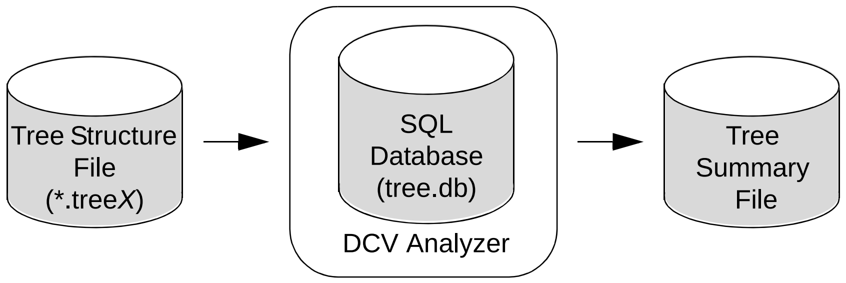
For hierarchy analysis, the DCV Analyzer tool takes input from a tree structure file that you specify and stores the data in an SQL database, named tree.db, in the current working directory. Then the tool analyzes the hierarchy data and generates reports that it saves in output files in the current working directory. By default, the tool provides hierarchy analysis information in a tree summary file named tree_summary.txt.

For example, you can enter the following command:

```
# dcv_analyzer mycell.tree0
```

The tool generates the SQL database and the tree summary file in the current working directory. Most of the runtime involves creating the SQL database. For subsequent runs with the same hierarchy data, you can use the `-r` command-line option to reanalyze the data and regenerate or add reports from the same SQL database. Figure 2-2 illustrates this flow.

Figure 2-2 DCV Analyzer Hierarchy Analysis Flow



The hierarchy analysis operation is completely independent of the performance analysis operation; you can draw any linkage between the results that you want.

For hierarchy comparisons, the DCV Analyzer tool takes input from two tree structure files that you specify and stores the data in two SQL databases, named tree.db and tree2.db, in the current working directory. By default, the tool provides the following information:

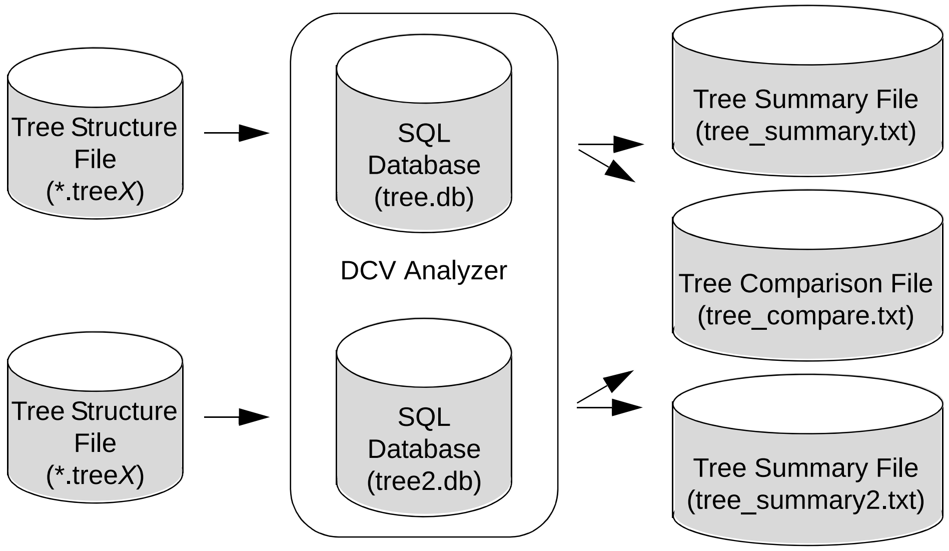
- Hierarchy analysis information in two tree summary files, named tree_summary.txt (from tree.db) and tree_summary2.txt (from tree2.db)
- Hierarchy comparison information in a tree comparison file, named tree_compare.txt, that highlights cross-referenced data between the two tree structure files

The tree structure files do not have to be from the same IC Validator run nor do they have to be of the same type (for example “tree0”). The DCV Analyzer tool can process any two tree structure files. For example, you can enter the following command:

```
# dcv_analyzer mycell.tree0 mycell.tree999
```

The tool generates the SQL databases and the tree summary files in the current working directory. Most of the runtime involves creating the SQL databases. For subsequent runs with the same hierarchy data, you can use the `-r` command-line option to reanalyze the data and regenerate or add reports from the same SQL databases. Figure 2-3 illustrates this flow.

Figure 2-3 DCV Analyzer Hierarchy Comparison Flow


Tree Structure File (*.treeX)
SQL Database (tree.db)
DCV Analyzer
Tree Structure File (*.treeX)
SQL Database (tree2.db)
Tree Summary File (tree_summary.txt)
Tree Comparison File (tree_compare.txt)
Tree Summary File (tree_summary2.txt)

The tool also generates a reference cells file, reference_cells.txt, that compares only the differences found in the Cell Statistics sections of the tree structure files. When the statistics match for a cell, no data for that cell is included in the reference cells file. If the statistics for a cell appear in only one of the tree structure files, the data for that cell is omitted from the reference cells file.

The tool can generate additional output files depending on the options that you include on the `dcv_analyzer` command line.

- Cell information files

  If you use the `-C` command-line option to specify a list of cell names, the tool produces a file named cell_*cell-name*.txt for each cell. These files contain statistical information about the cells.

- Layer information files

  If you use the `-L` command-line option to specify a list of layer names or layer number and data type pairs, the tool produces a file named layer_*layer-name*.txt for each layer name and a file named *layer_num*:*data_type*.txt for each layer number and data type pair. These files contain statistical information about the layers.

For more information about these output files, see the following sections:

- Tree Summary Files
- Tree Comparison File

- Cell Information Files
- Layer Information Files

Tree Summary Files

When you specify a single tree structure input file, the DCV Analyzer tool creates a tree summary file named tree_summary.txt. If you specify a second tree structure input file, the tool also creates a file, named tree_summary2.txt, and a tree comparison file, named tree_compare.txt. The tree_summary2.txt file contains the same data for the second tree structure file that tree_summary.txt contains for the first tree structure file. The label "Summary for analysis of ..." appears at the top of each tree summary file to identify which tree structure file the data belongs to. The Design Statistics section in the tree structure file contains data from the header of the tree structure file. Table 2-4 provides definitions of these design statistics.

Table 2-4 Tree Structure Design Statistic Definitions

Statistic	**Definition**
Hierarchical levels	The number of levels in the design.
Unique cells	The number of unique cells in the design.
Hierarchical placements	The total number of placements (SREFs and AREFs) at each level in the design.
Total design placements	The total number of placements (SREFs and AREFs) in the design.
Exploded references	The total number of exploded references in the design.
Exploded placements	The total number of exploded placements in the design.
Total data count	The total number of all polygons, paths, edges, and rectangles in the design.
SREF placements	The total number of cell references in the design.
AREFs	The total number of array references in the design.
AREF placements	The total number of references in each array reference in the design.
Data	The number of data primitives in each cell of the design.
Text	The number of text primitives in each cell of the design.
Via	The number of via cells in the design.

Example 2-4 shows an example of the Design Statistics section.

Example 2-4 Design Statistics Section Example

```
Design Statistics:
-----------------
    Hierarchical Levels:  3
           Unique Cells:  25,101
Hierarchical Placements:  54,224,801
Total Design Placements:  54,224,801
    Exploded References:  0
    Exploded Placements:  0
       Total Data Count:  2,276,754,400
        SREF Placements:  54,224,801
                  AREFS:  0
        AREF Placements:  0
                   Data:  330,756,000
                   Text:  335,400
                    Via:  2,500

 Level  Placement Count  Path Count   Unique Cell Count
 -----  ---------------  ----------   -----------------
     0                1           1                   1
     1              100         100                 100
     2       54,224,700      25,000              25,000

 Depth       Cell Count
 -----       ----------
     0           25,000
     1              100
     2                1
```

The Placement Count is the total number of cells placed at each level of hierarchy.

The Path Count is the number of unique paths to a cell. For example, `Top – L1 – L2` counts as one path even if cell L2 is placed 500 times.

The unique Cell Count is the number of unique cells that appear at each level of hierarchy. For example, suppose the hierarchy has the following tree structure:

```
TOP        Level=0  Count=   1
  L1A      Level=1  Count=   1
    L2     Level=2  Count= 500
  L1B      Level=1  Count=   1
    L2     Level=2  Count=   1
  L2       Level=1  Count=  10
```

Given this structure, the tree summary file shows the following placement counts:

```
 Level  Placement Count     Path Count   Unique Cell Count
 -----  ---------------     ----------   -----------------
     0                1              1                   1
     1               12              3                   3
     2              510              2                   1
```

Depth is defined as the furthest distance from the leaf cell, which is a cell without child cells. For example, if a design has 10 levels of hierarchy, the top cell always has a depth of 10.

Tree summary files contain the following tables:

- Cells sorted by Total Placements

  This table shows cells that are placed the most times. The placement count is for a specific location in the hierarchy (indicated by the Path), so a cell can appear multiple times in this table.

- Cells sorted by Data

  This table shows cells that contain the most geometric objects (all layers). Text counts are displayed in the table but not included in the sorting.

- Cells sorted by Flat Objects

  This table displays the cells that contribute the most data to the design based on a combination of the number of times they are placed and the amount of geometric objects contained within the cell.

  ```
  Flat Objects = FlatPlacements * (Data + Text)
  ```

- Layers within each Cell sorted by Data

  This table shows the layers by cell that contain the most data. The layer number and data type may not be available for certain data formats.

- Layers sorted by Flat Objects
  Layers sorted by Flat Data
  Layers sorted by Flat Text

  These tables display the layers that contribute the most data to the design based on a combination of the number of times the cells they are in are placed and the amount of geometric objects assigned to that layer.

  ```
  Flat Objects = FlatPlacements * (DataCount + TextCount)
  Flat Data = FlatPlacements * DataCount
  Flat Text = FlatPlacements * TextCount
  ```

Tree Comparison File

When you specify two tree structure input files, the DCV Analyzer tool automatically creates a tree comparison file named tree_compare.txt. You can compare two tree structure files from the same or different IC Validator runs. The format of the tree structure files (Milkyway, NDM, LTL, GDSII, or OASIS) does not matter.

The first two tables in the tree comparison file define the missing cells and extra cells from the second tree structure file, sorted by cell area. By default, only the 10 largest cells are shown, but the table headers indicate the total number of cells. You can use the `–c` command-line option to increase the number of rows in the tables.

Note:
Cells that appear in either the extra cells or missing cells table are not included in the majority of the other tables because they are not in both reports and the tool cannot perform a meaningful comparison.

Many of the tables in the tree comparison file contain the same information as the tables in the tree summary file, with the sorting column presented as a gain comparison from the second input file to the first input file. Because of this switching, the order of the input files on the command line might produce a completely different set of data for each table. Consider the order carefully before running the tool.

The tree comparison file contains the following tables:

- Matching Cells sorted by Total Placements Gain

  This table calculates the ratio of total placements between the files.

- Matching Cells sorted by Data Gain

  This table calculates the ratio of total data (all layers) within each cell between the files.

- Matching Cells sorted by Flat Objects Gain

  This table shows the ratio of total flat objects in each file.

```
Gain = (FlatPlacements2 * (Data2 + Text2))/( FlatPlacements1 * (Data1 + Text1))
```

- Layers in Matching Cells sorted by Data Difference

  This table shows the data by layer and cell difference between the files. The tool uses the absolute values, so the file order should not affect the results.

- Layers within each Cell sorted by Gain

  This table appears twice and shows the same comparison as the previous table, but with the relative differences rather than the absolute differences. The layer and cell names must be the same in both input files for the comparison to be valid. One copy of the table shows the values of the first file to second file comparison, and the other copy of the table shows the second file to first file comparison.

- Matching Layers Flat Objects sorted by Gain
  Matching Layers Flat Data sorted by Gain
  Matching Layers Flat Text sorted by Gain

  These tables show the gain (Flat Objects2/Flat Objects1) ratio between the files. “Data” and “Text” are substituted for “Objects” in their respective tables.

  ```
  Flat Objects = FlatPlacements * (DataCount + TextCount)
  Flat Data = FlatPlacements * DataCount
  Flat Text = FlatPlacements * TextCount
  ```

- Cells sorted by Placement Difference

  This table shows the absolute difference between the two input files of the flat placement count of each cell. Because the tool uses absolute values, the order of the input does not matter. In addition, the missing and extra cells from the first tables should count as “0” in the calculations for this table.

- Cells with changed Status sorted by Flat Placements

  This tables shows matching cells between the input files in which the status does not match. The order is based on the sum of the number of times the cell is placed in each file.

- Status Changed Summary

  This table shows the number of unique pairs of how the status of a given cell might have changed between both input files.

Cell Information Files

When you use the `–C` command-line option to specify a list of cells, the DCV Analyzer tool creates a report file named cell_*cell-name*.txt for each cell in the cell list. If you specify a cell that is not in the data, the tool produces an empty file that contains only table headings.

If you include a wildcard character (*) to specify multiple cell names, you should enclose the entire argument in quotation marks to avoid having the shell interpret the wildcard character (to include files in the current working directory).

For example, `-C "NAND2X0,M*"` produces a file named cell_NAND2X0.txt and a file for each cell with a name that begins with an uppercase M (the cell names are case-sensitive). In this example, if no cells with a name that starts with M exist, the tool produces only the NAND2X0 file.

Cell information files contain the following sections:

- The first section of the report contains statistics, which are similar to the design statistics in the summary report file.
- The second section compares the layers in each cell and the amount of data and text on each layer. The differences reported in this section account for both text and data.
- The third section shows the names of all the immediate child cells with their placement count and status. In this section, the tool compares and calculates differences as appropriate.
- The final section shows all the available paths to the cell of interest in each file, which helps you to determine the parent cells.

You must specify at least one tree structure input file when you use the `–C` command-line option. The DCV Analyzer tool includes data from the input file in the cell information files. If you specify two tree structure files, the tool includes data from both input files side by side, along with a difference column. Differences can be absolute or relative depending on the data, or just an X for a string comparison (Extents and Status).

Layer Information Files

When you use the `–L` command-line option to specify a list of layers, the DCV Analyzer tool creates a layer information file named layer_*layer-name*.txt for each layer in the layer list. Each file contains a list of cells and the number of geometric objects per cell for that layer.

If you include a wildcard character (*) to specify multiple layer names, you should enclose the entire argument in quotation marks to avoid having the shell interpret the wildcard character (to include files in the current working directory).

You must specify at least one tree structure input file when you use the `–L` command-line option. If you specify two tree structure files, the tool includes two lists of cells in the layer information file, one from each tree structure file.

3

DCV Switch Optimizer Tool

This chapter explains how to run the DCV Switch Optimizer tool and view the reports.

The IC Validator DCV Switch Optimizer tool exercises combinations of runset preprocessor flow control switches to identify possible runtime parse errors.

For more information, see the following sections:

- Overview
- Command-Line Options
- Runset Switch Description File
- Error Output Log Files
- Summary Report File

Overview

Many DRC runsets consist of switch blocks that are complex in nature. The manual process of identifying the necessary switch combinations that exercise the entire runset code and detecting any possible parse errors in the runset is often inefficient and requires a lot time and effort.

The DCV Switch Optimizer tool reduces the manual intervention and provides an interface that allows you to exercise the entire runset code optimally and qualify a runset free of parse errors. The tool can exercise a minimal number of user-defined flow control switch combinations to identify possible runtime parse errors. This contrasts with manual runset validation processes that can be error prone (unexpected combinations) or overly conservative, resulting in more runset parse-testing iterations than necessary.

The DCV Switch Optimizer tool takes input from a DRC runset file and produces a log file for each failed parse iteration and a summary report file. The tool scans the runset file to identify all of the switches present in the runset and identifies the minimal set of switch combinations that fully exercise all of the code in the runset.

The tool can also take input from an optional runset switch description file. You should provide this file if the runset contains any of the following special switch types:

- Mutually exclusive switch cases
- Default and default list switch cases
- Ignore switch cases

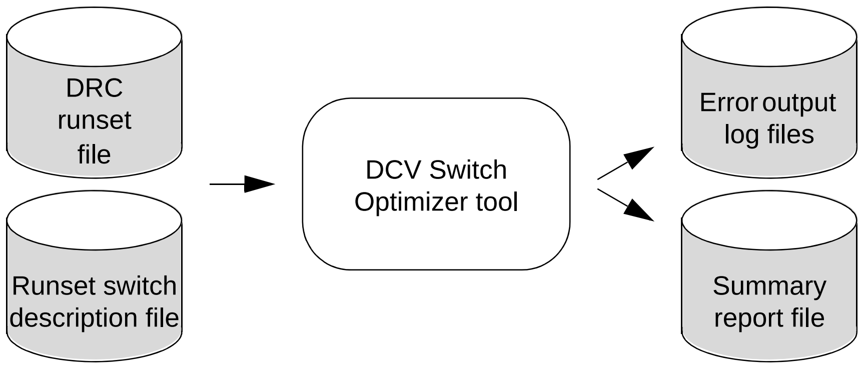
For more information about this file, see “Runset Switch Description File” on page 3-3.

The DCV Switch Optimizer tool performs the following tasks:

1. Examine the entire runset and identify all code blocks and the switches or switch combinations that exercise all code
2. Determine the minimum set of switch combinations that fully cover the runset code
3. Run IC Validator iteratively with each set of switch combinations

Figure 3-1 illustrates this flow.

Figure 3-1 DCV Switch Optimizer Tool Flows



Command-Line Options

The DCV Switch Optimizer tool command-line syntax is

```
dcv_swop runset-file [command-line-options]
```

where *runset-file* specifies the DCV runset file.

Table 3-1 describes the command-line options.

Table 3-1 DCV Switch Optimizer Tool Command-Line Options

Argument	Description
`-h` `--help`	Displays the usage message and exits.
`-i` `--input`	Specifies the path to the runset switch description file.
`-V` `--version`	Prints the tool version.

Runset Switch Description File

The DCV Switch Optimizer tool takes input from a DRC runset file and an optional runset switch description file. You should provide a runset switch description file as input if the DRC runset contains any special switch types.

For example, to specify mutually exclusive switch types, assume that you have the following switches:

```
SwitchA
SwitchB
SwitchC
SwitchD
SwitchE
SwitchF
SwitchG
SwitchH = dx_YES
SwitchI
SwitchJ = dx_NO
```

These switches have the following mutually exclusive conditions:

- `SwitchA`, `SwitchB`, and `SwitchC` are mutually exclusive with each other
- `SwitchB` is mutually exclusive with `SwitchD` and `SwitchE`
- `SwitchE` and `SwitchF` are mutually exclusive with each other
- `SwitchG` is mutually exclusive with `SwitchI` and `SwitchH = dx_YES`
- `SwitchH = dx_YES` and `SwitchJ = dx_NO` are mutually exclusive with each other

Use the following format to describe these mutually exclusive switch conditions:

```
#MUTEX START
       SwitchA {SwitchB, SwitchC}
       SwitchB {SwitchD}
       SwitchB {SwitchE}
       SwitchE {SwitchF}
       SwitchG {SwitchI, SwitchH = dx_YES}
       SwitchH = dx_YES {SwitchJ = dx_NO}
#MUTEX END
```

Note:
The switches within curly braces should always be on the same line in the runset switch description file.

For more information about the special switch types that you can describe in this file, see the following sections:

- Mutually Exclusive Switch Cases
- Default and Default List Switch Cases
- Ignore Switch Cases

Mutually Exclusive Switch Cases

Use the mutually exclusive switch case to identify switches and values that should not be allowed at the same time. For example, if the following switches are turned on at the same time, a parse error results:

```
#ifdef LAYOUT_FORMAT_GDS
   aFFO = assign ( {{ 1, 0 }} );
#endif

#ifdef LAYOUT_FORMAT_OASIS
   aFFO = assign ( {{ 100, 0 }} );
#endif
```

You can avoid having to specify mutually-exclusive switches in the runset switch description file by adding an “error detection” case. For example:

```
#ifdef LAYOUT_FORMAT_GDS
   #ifdef LAYOUT_FORMAT_OASIS
      #error “Can’t do this”
   #endif
#endif
```

The DCV Switch Optimizer tool automatically detects these `#error` cases and prevents corresponding switches from being active at the same time.

Default and Default List Switch Cases

Use the default switch case to identify switches that must be set to some value every time the runset is used. These switches fall into one of the following categories:

- Single default value
- List of default values

Single default value switches must be set to a specific value for every parser iteration. This is typically done to establish a known startup state and is not meant to be modified by the runset users. For example:

```
#DEFAULT START
ABC = ON
FFO = ON
BRR = TOT
PQR = OFF

#DEFAULT END
```

The result is a runset with the following switch settings for each parser iteration:

- `#define ABC`
- `#define FFO`
- `#define BRR TOT`
- `#undef PQR`

Note:
You do not need to specify entries for `#define` statements used to set variables to a specific value (for example, macro expansions).

List of default values switches must be set to a value for every parser iteration, and each value must come from a defined list of valid choices. The DCV Switch Optimizer tool chooses a random value from the list and sets the switch accordingly. A new value is chosen for every parser iteration. For example:

```
#DEFAULT START
ABC = {value_1, value_2, value_3}
#DEFAULT END
```

Ignore Switch Cases

The ignore switch case refers to switches that are to be left alone by the DCV Switch Optimizer tool. These switch settings, and their resulting behavior, are left as coded in the original runset.

The most frequent application where these types of switches occur is in the control of the inclusion of common blocks of PXL code, as shown by the following example:

```
#ifndef ASSIGN_RS
        #include assign.rs
        #define ASSIGN_RS
#endif
```

In this example, the `ASSIGN_RS` switch is used to make sure that the assign.rs file is included only once. This coding style relies on the switch not being turned on at the start.

If the DCV Switch Optimizer tool tries to set the `ASSIGN_RS` switch to `ON` as part of testing and exercising of all the runset code blocks, the assign.rs file would not be included. This results in a parse error because the layers that the tool expects subsequent code to define in the file would not be created.

Example 3-1 Ignore Syntax Example

```
#IGNORE START
ASSIGN_RS
#IGNORE END
```

Note:

An alternative way to handle this example would be to use the Single Value Default case to set `ASSIGN_RS` to `OFF`.

```
#DEFAULT START
ASSIGN_RS = OFF
#DEFAULT END
```

Error Output Log Files

When the DCV Switch Optimizer tool identifies a set of switch options that produces a parse error, the tool produces an error log file. This file contains the IC Validator output for the run and describes why the iteration failed. The tool creates an error log file for each iteration that has a parse error. The file names have the format ParserIteration_*number*.log, where *number* is the parser iteration number where the error occurs.

Summary Report File

The tool produces a report file that summarizes all of the parser iterations. The report contains the following information:

- Total number of iterations required to cover the entire runset code.
- Results of all of the parser iterations.
- Detailed descriptions of iterations that result in a parse error.

The report includes the following sections:

- Parser Results Summary

  This section lists the parser iteration numbers and results. The results are either PASS or FAIL. For failed parser iterations, the report also includes the path to the iteration-specific log file.

  The following example lists a parser iteration that failed and a parser iteration that passed:

```
Parser Iteration 1: FAIL - Check ParserIteration_1.log file for more info...
Parser Iteration 2: PASS
```

- Parser Results Details

  This section contains a detailed description of any iteration that results in a parse error. This information includes the specific switch settings corresponding to the failed parser iteration, a Pass or Fail indicator, and the IC Validator command-line call that produces the parse failure.

The following example shows the description of a parser iteration that failed:

```
#########################
# Parser Iteration 1    #
#########################
TURN ON:
    d_11M_3MX_4FX_2HX_2GX_LB__R0
TURN OFF:
    d_CELL_FINE_SS_YES
Switch Values:
    d__SRULES_MERGED = dx_NO
RESULT: FAIL
ICV Called as: icv -nro -c a -i /remote/us54home1/user/empty_test.gds
-D d_11M_3MX_4FX_2HX_2GX_LB__R0
-D d__SRULES_MERGED=dx_NO /remote/us54home1/user/runset/runset.rs
```

Example 3-2 shows an example of a DCV Switch Optimizer report file.

Example 3-2 DCV Switch Optimizer Report File

```
Runset Switch Combinations - Parser Results Summary
===================================================

Total Iterations Required: 6

Results:
--------
Parser Iteration 1 : FAIL - Check ParserIteration_1.log file for more info...
Parser Iteration 2 : PASS
Parser Iteration 3 : PASS
Parser Iteration 4 : PASS
Parser Iteration 5 : PASS
Parser Iteration 6 : PASS


Runset Switch Combinations - Parse Results Details
===================================================

###########################
# Switch Combination 1 :  #
###########################

TURN ON :

       SWITCH_ABC
       SWITCH_CDE
       SWITCH_PQR
       SWITCH_XYZ

TURN OFF :

       SWITCH_LMN
       SWITCH_FFO
```

```
Switch Values:

       SWITCH_BRR = Mx_NO
       SWITCH_TOT = Mx_YES

ICV Called as : icv –nro –c swop_test_top_cell –i /Path/to/the/dcv_swop_test.gds –D
SWITCH_ABC –D SWITCH_CDE –D SWITCH_PQR –D SWITCH_XYZ –D SWITCH_BRR = Mx_NO –D
SWITCH_TOT = Mx_YES  /path/to/the/runset.rs

RESULT: FAIL
```

4

DCV Results Compare Tool

This chapter explains how to run the DCV Results Compare tool and view the reports.

The IC Validator DCV Results Compare tool compares two sets of error data from different sources and produces a prioritized discrepancy report, in ASCII format, and output that you can view in the IC Validator VUE tool.

For information about this tool, see the following sections:

- Overview
- Command-Line Options
- Running the Tool
- Rule-Map File
- Comparison Results File

Overview

You can use the DCV Results Compare tool to compare the results from different IC Validator runs or to compare IC Validator results with the results from a third-party tool. Performing such comparisons manually can be a tedious, error-prone process. Although the IC Validator LVL tool can compare two layouts, only the DCV Results Compare tool provides an error-versus-error comparison for runset results.

The DCV Results Compare tool performs an automated results-versus-results comparison using error results as input. To run the tool, you must specify the comparison type, the baseline and comparator results, and the layout data. You choose the type of comparison to be made and the input formats for the error results and layout data.

The comparison type controls the nature of the comparison. Depending on the comparison type that you specify, the tool can report discrepancies if the error markers do not

- Match exactly
- Match exactly within an expanded region
- Overlap except for an edge abutment or corner touch
- Overlap except for a corner touch
- Overlap at all

The Results Compare tool automatically detects the input formats of the error results and layout data.

- The baseline results are the error markers that the tool uses as reference data.
- The comparator results are the error markers that you want to compare against the reference data.
- The layout data can be the original layout database that you used to generate the error results or a skeleton database that contains just the layout hierarchy data.

This input data can have different formats. Error results can be in an IC Validator error file, a PYDB database, an OASIS file, a GDSII file, a third-party ASCII error file, or a DCV internal error format file. The layout data can be in an OASIS file or a GDSII file.

By default, the results comparison is a one-to-one mapping based on matching rule names. You can specify custom rule mappings by creating an input rule-map file or by editing the optional default rule-map file that the tool creates.

The tool can also generate a default rule-map file that you can modify and use as input for a subsequent run.

Note:
A rule-map file is required when the error input data is in GDSII or OASIS format.

You can select a subset of rules to use during a run by including or excluding individual rule groups defined in the rule-map file.

You can also set waivers that prevent the tool from reporting “known” results comparison discrepancies by specifying an error classification database. The tool marks these waived discrepancies in the reports that it generates. The tool limits waiver usage to approved results.

The DCV Results Compare tool produces a comparison report that summarize the details of how the comparator results correlate with the baseline results. The report contains a report header, a comparison summary, and detailed comparison results.

Command-Line Options

Before running the DCV Results Compare tool, make sure your `LM_LICENSE_FILE` and `ICV_HOME_DIR` environment variables are set. The `dcv_rct` executable is located in the same bin directory as the `icv` executable. The tool creates the output database and report files in your current working directory.

The DCV Results Compare tool command-line syntax is

```
dcv_rct
    [options]
    overlap | exact | fuzzy_exact
    baseline_data
    comparator_data
    layout_data
```

where

- *`baseline_data`* specifies the results file that contains the reference error data
- *`comparator_data`* specifies the error file that contains the error data for comparison with the reference error data
- *`layout_data`* specifies either the original layout database used to generate the error data or a skeleton database that contains just the layout hierarchy data

Table 4-1 describes the command-line options.

Table 4-1 DCV Results Compare Tool Command-Line Options

Option	**Description**
`--all_rules`	Disables rule filtering. By default, the tool excludes rules from the rule map file for which the baseline and comparator data contain no violations and does not include these rules in the output report.

Table 4-1 DCV Results Compare Tool Command-Line Options (Continued)

Option	Description
`--convert`	Converts error data from any supported input format to the internal DCV error map format used by the tool.
`--cpydb_name` *`error-db`*	Specifies the name of the error classification database (cPYDB) that contains error waivers. You must use the `--cpydb_path` option to specify the path to the database.
`--cpydb_path` *`path-name`*	Specifies the path to the error classification database (cPYDB) that contains waivers for comparison discrepancies. You must also use the `--cpydb_name` option to specify the name of the database.
`--csv`	Saves the summary report in a comma separated value (CSV) format file, in addition to the comparison report file. You can import this CSV file into a third-party tool, such as the Microsoft® Excel® tool, for further analysis.
`--dbname` *`database`*	Specifies the error database (PYDB) that the tool reads for error input.
`--delta` *`value`*	Specifies the tolerance value that the tool uses to expand the baseline and comparator data for the `fuzzy_exact` comparison type. The tool uses this value as a multiplier of the input database resolution. The maximum value is `10`. The default is `5`.
`-dp`*`#_of_hosts`*	Networks a distributed run using the specified number of distributed processing hosts, all running on the local host. For example, `-dp4` specifies four hosts. The default is two hosts, `-dp2`, using only as many processors as there are distributed processing clients on a machine for dynamic threading. If you want to use all of the processors on the machines for dynamic threading, use the `-turbo` option. For more information about distributed processing, see the *IC Validator User Guide*.
`-g`	Generates a rule-map file containing default rules that you can modify and use as input for a subsequent run. The tool determines these default rules based on the data it is comparing when you do not specify an input rule-map file with the `-m` option.
`-h` `--help`	Displays the usage message and exits.

Table 4-1 DCV Results Compare Tool Command-Line Options (Continued)

Option	Description

`-i edge | all`
`--include edge | all`
	Specifies the types of error marker interactions that represent a match when you specify the `overlap` comparison type. By default, the tool reports discrepancies for error markers that do not interact in any way. Use `edge` to report discrepancies for error markers that do not interact except for edge abutments or point touches. Use `all` to report discrepancies for error markers that do not interact except for point touches only.

`-m` *`map-file`*
`--map` *`map-file`*
	Specifies a rule-map file containing the rules that the tool uses to compare results. If you do not specify this file, the tool determines default rules based on the data it is comparing. To generates a default rule-map file that you can modify and use for subsequent runs, specify the `-g` option.

`--skeleton`	Creates a layout database that contains only the layout hierarchy data from the original layout database, which is required input when you use this option. For subsequent runs, you can specify this skeleton layout database as input instead of the original layout database.
`--srg` *`rule-group-list`*	Specifies one or more rule groups, from the rule-map file, that the tool uses to compare the results. Separate individual rule groups with commas.
`-turbo`	Allows the tool to use all of the processors on the machines for dynamic threading. If you do not use this option, the tool can use only as many processors as there are distributed processing clients on a machine for dynamic threading.
`--urg` *`rule-group-list`*	Specifies one or more rule groups, from the rule-map file, that the tool excludes from the results comparison. Separate individual rule groups with commas.

Running the Tool

The DCV Results Compare tool compares errors from two data files and reports discrepancies based on the comparison type that you specify. The tool also provides command-line options that you can use to

- Specify custom rule mappings in a rule-map file
- Select rules that you want to include or exclude
- Waive individual discrepancies by specifying an error classification database

To compare error data, you specify the type of comparison you want to perform, the files that contain the errors you want to compare, and the layout data. For example, to compare the results from a baseline file named TOP.LAYOUT_ERRORS and a comparator file named TOP.db with the original layout data in a file named TOP_input.gds, and report discrepancies for errors that do not match exactly, enter

```
% dcv_rct exact TOP.LAYOUT_ERRORS TOP.db TOP_input.gds
```

The available comparison types are `overlap`, `exact`, and `fuzzy_exact`. For more information, see “Setting the Comparison Type” on page 4-6. For information about the file formats that you can use for the input files, see “Input File Formats” on page 4-9.

You can specify either the original layout database used to generate the results or a skeleton database that contains just the layout hierarchy data. The tool uses this data to resolve hierarchical differences in the output results or if either the baseline or comparator results are in a text file format.

The Results Compare tool performs a rule-based error-versus-error comparison of the error data and creates a comparison report listing the discrepancies sorted by severity. The tool generates both a report file, *top_cell*.RCT.report, and a VUE-compatible file, *top_cell*.RCT.vue, where *top_cell* is the name of the top cell in the design.

You can also save the report in a comma-separated value (CSV) format file by using the `--csv` command-line option. You can import this file into a third-party tool (such as the Microsoft Excel tool) for further analysis.

For information about the report file, see “Comparison Results File” on page 4-14. For information about using the IC Validator VUE tool, see the *IC Validator VUE User Guide*.

For more information, see the following sections:

- Specifying Custom Rule Mappings
- Including or Excluding Rules
- Waiving Errors With an Error Classification Database

Setting the Comparison Type

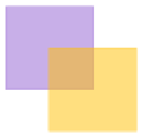
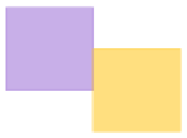
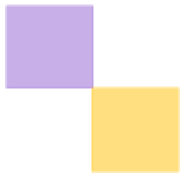
The comparison type controls the nature of the comparison by setting the criteria for matches between baseline and comparator error markers. To set the comparison type for a run, you must specify `overlap`, `exact`, or `fuzzy_exact`. The tool performs an automated results-versus-results comparison of the error data and reports PASS for errors that match and FAIL for errors that fail the comparison.

Use `overlap` to match error markers that overlap. You can include edge abutments and corner touches by using the -i command-line option.

- To report a match for error markers that have an edge abutment or a corner touch but do not overlap, use the `-i edge` option.
- To report a match for error markers that have a corner touch but do not overlap or have an edge abutment, use the `-i all` option.

Table 4-2 shows the truth tables for the `overlap` comparison type.

Table 4-2 Interactive Matches

Comparison type	Overlap	Edge abutment	Point touch
			
`overlap`	PASS	FAIL	FAIL
`overlap -i edge`	PASS	PASS	FAIL
`overlap -i all`	PASS	PASS	PASS

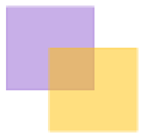
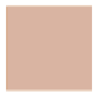
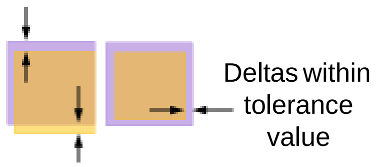
Use `exact` to match error markers that match exactly.

Use `fuzzy_exact` to match error markers that match exactly within an expanded region. You can define a tolerance value for this region by using the `--delta` command-line option to specify a multiplier of the input database resolution. The maximum value is `10`. The default is `5`.

For example, if the input database resolution is 0.001 um, the default tolerance value for the expanded region is 0.005 um. The tool expands the error markers from the baseline and comparator data, and then evaluates them to determine if they match within these expanded regions.

Table 4-3 shows the truth tables for the `exact` and `fuzzy_exact` comparison types.

Table 4-3 Exact Matches

Comparison type	Overlap	Exact	Fuzzy exact
			
`exact`	FAIL	PASS	FAIL
`fuzzy_exact`	FAIL	PASS	PASS

Input File Formats

The Results Compare tool automatically detects the input data formats. The baseline and comparator data do not have to be in the same format. The valid formats for the baseline and comparator data files are

- IC Validator LAYOUT_ERRORS file
- IC Validator error database (PYDB)
- DCV error map database
- GDSII
- OASIS
- Third-party ASCII error file

Note:
When the baseline or comparator data is in the IC Validator error database format, use the `--dbname` command-line option to specify the PYDB file name.

The input data does not need to be flattened because the tool preserves the hierarchy of the input data during a comparison.

The valid formats for the original layout data file are

- GDSII
- OASIS

Specifying Custom Rule Mappings

By default, the DCV Results Compare tool creates rule mappings based on a 1:1 correspondence of matching rule names extracted from the baseline and comparator error input. These default 1:1 rule mappings are sometimes inadequate. For example, you might need to map multiple rules together in an *N*:1, 1:*M*, or *N*:*M* mapping. You might also encounter cases where rule names do not match and the tool creates “place-holder” rule names.

You can provide customized rule mappings in a rule-map file, which you specify by using the `-m` command-line option. For example, to specify a rule-map file named TOP.RCT.rule_map, enter

```
% dcv_rct -m TOP.RCT.rule_map exact TOP.LAYOUT_ERRORS TOP.db TOP_input.gds
```

Note:
If your error input data is in GDSII or OASIS format, you must specify a rule-map file.

The tool generates a default rule-map file when you use the `-g` command-line option. You can modify this file and use it as input in a subsequent run. Use this option if you know in advance that the default 1:1 rule mappings are inadequate.

```
% dcv_rct -g exact TOP.LAYOUT_ERRORS TOP.db TOP_input.gds
```

For more information about rule-map files, see “Rule-Map File” on page 4-11.

Including or Excluding Rules

You can limit the comparison to a subset of the available rules by adding rules to an empty selection or by subtracting rules from a universal selection. You specify the rules by using their group identifiers in the rule-map file. The tool selects the entire rule group associated with an identifier to ensure a correct comparison.

To include rules, use the `--srg` command-line option to specify the rules or rule groups. For example, to use only rule groups 1, 2, and 4, enter

```
% dcv_rct -m TOP.RCT.rule_map --srg 1,2,4 exact TOP.LAYOUT_ERRORS TOP.db TOP_input.gds
```

To exclude rules, use the `--urg` command-line option to specify the rules or rule groups. For example, to exclude rule groups 2 and 5, enter

```
% dcv_rct -m TOP.RCT.rule_map --urg 2,5 exact TOP.LAYOUT_ERRORS TOP.db TOP_input.gds
```

The Results Compare tool automatically filters rules for which the data does not contain violations by default. You can disable this filtering by using the `--all_rules` command-line option.

```
% dcv_rct --all_rules exact TOP.LAYOUT_ERRORS TOP.db TOP_input.gds
```

Waiving Errors With an Error Classification Database

The DCV Results Compare tool allows you to specify waivers in an error classification database that prevent the tool from reporting “known” results comparison discrepancies. The tool marks these waived discrepancies in the reports that it generates.

To specify the error classification database (cPYDB), use the `--cpydb_name` and `--cpydb_path` command-line options.

Error classification for the DCV Results Compare tool is similar to the IC Validator error classification flow. The only difference is that you specify the error classification database on the command line by using the `--cpydb_name` and `--cpydb_path` options.

In general, follow these steps to set waivers for comparison discrepancies:

1. Run the DCV Results Compare tool on the input error data and layout data.
2. Classify the comparison discrepancies that you want to waive and save them in an error classification (cPYDB) database by using the IC Validator VUE tool or the pydb_report utility.

   You can merge new classifications into an existing error classification database. The tool appends the newly classified errors to the database and updates existing errors that have changed.

   For more information about classifying errors, see the *IC Validator VUE User Guide*. For information about error classification databases, see the *IC Validator User Guide*.
3. Rerun the DCV Results Compare tool with the error classification database to preclassify the discrepancies.

   Use the `--cpydb_name` and `--cpydb_path` command-line options to specify the name and location of the error classification database.

Rule-Map File

The rule-map file specifies how the DCV Results Compare tool compares the baseline and comparator error input. By default, the tool uses a 1:1 mapping based on matching rule identifier names. The rule-map file consists of six comma-separated fields. Table 4-4 describes these fields.

Table 4-4 Fields in a Rule-Map File

Field	Description
Rule group ID	Defines the correspondence between the baseline and comparator results to be compared. The results from rules that contain the same rule group ID are merged for comparison.
Baseline rule identifier	Identifies the process ground rule from the baseline results, for example, GRMx.S.2.
Baseline (*layer*;*datatype*)	Identifies the baseline *layer*;*datatype* assigned for the generation (default case) of the baseline error shapes. For GDSII or OASIS error files, this data corresponds to the *layer*;*datatype* assigned to the error output.
Comparator rule identifier	Identifies the process ground rule from the comparator results, for example, GR.Mx.S.2.

Table 4-4 Fields in a Rule-Map File(Continued)

Field	Description
Comparator (*layer*;*datatype*)	Identifies the *layer*;*datatype* assigned for the generation (default case) of the comparator error shapes. For GDSII or OASIS error files, this data corresponds to the *layer*;*datatype* assigned to the error output.
Description	Identifies the rule. This descriptive comment typically corresponds to the Process Design Manual rule name or rule descriptor, for example, GRMx.S.2 : Mx minimum.

The comparison result descriptions are text strings that describe each result being compared. The default rule-map file uses rule descriptions, if they are available, or creates default text descriptions. The tool uses these descriptions in the output report.

Example 4-1 shows the format of a rule map file with three rule groups.

Example 4-1 Rule-Map File Format

```
RuleGroup, BaselineRule, Baseline(L;D), ComparatorRule, Comparator(L;D), Description
      1  ,         F   ,         0;0  ,    F          ,            0;0  ,    Rule F
      2  ,         F.1 ,         1;0  ,    F_1        ,            1;0  ,    Rule F.1
      3  ,         B   ,         2;0  ,    RCT_genC_B ,            2;0  ,    Rule B
```

Note:
In a real rule-map file, the *layer*;*datatype* assignments might not be this well ordered.

The first non-empty line in the file is the header line. Newlines are significant only at the end of a record, and multiple contiguous newlines are treated as one newline. Other white space is ignored unless quoted. Commas can also be quoted, in which case they are not counted as a field separator.

The layer and datatype values from the baseline and comparator results are required. The lack of a value designates a NULL entry. However, an empty string ("") is not counted as a NULL value.

By default, the DCV Results Compare tool creates rule groups with 1:1 mappings based on matching layers and datatypes in the baseline and comparator data. When rule names do not match, or a rule in one file is not in the other file, the tool automatically generates a dummy rule name of the form RCT_genB_*rule-name* or RCT_genC_*rule-name*, where B indicates baseline data, C indicates comparator data, and *rule-name* is the generated rule name.

Note:
A rule identifier that appears in more than one rule group causes an error.

You can create *M*:*N* rule mappings by modifying the 1:1 mappings that the tool generates. Make sure you use the same rule group for all of the rules that are part of a particular *M*:*N* mapping.

In Example 4-2, rule groups 1 and 2 are 1:1 maps, group 3 is a 1:3 map, and group 4 is a 2:4 map:

Example 4-2 Rule Mappings Organized by Rule Groups

```
RuleGroup, BaselineRule, Baseline(L;D), ComparatorRule, Comparator(L;D),   Description
        1,       M1.S.2,         100;0,          M1_S_2,           15;0,  "M1 minimum space"
        2,       Mx.S.1,         101;0,          M1_S_1,          101;0,  "M1 minimum width"
        3,       Bx.R.3,         102;0,          B1_R_3,            1;0,     "B1 touching C"
        3,             ,              ,          B2_R_3,            1;1,     "B2 touching C"
        3,             ,              ,          B3_R_3,            1;2,     "B3 touching C"
        4,      GRD.R.1,         103;0,        GRD_R_1a,          200;0,      "Grid check 1"
        4,             ,              ,        GRD_R_1b,          201;0,      "Grid check 1"
        4,      GRD.R.2,         103;1,        GRD_R_2a,          202;0,      "Grid check 2"
        4,             ,              ,        GRD_R_2b,          202;0,      "Grid check 2"
```

Rule correspondence is controlled through the rule group identifier. In Example 4-3, rule groups 3 and 4 contain a rule name mismatch and rule groups 5 through 7 contain a 1:*N* rule mapping, which in this example is a single baseline rule matched by two rules in the comparator data.

Example 4-3 Rule Correspondence

```
RuleGroup, BaselineRule, Baseline(L;D), ComparatorRule, Comparator(L;D), Description
      1  ,           F  ,          0;0  ,             F ,      0;0   ,        "Rule F"
      2  ,         F.1  ,          1;0  ,           F_1 ,      1;0   ,        "Rule F.1"
      3  ,           B  ,          2;0  ,    RCT_genC_B ,      2;0   ,        "Rule B"
      4  , RCT_genB_BB  ,          3;0  ,            BB ,      3;0   ,        "Rule BB"
      5  ,          Fx  ,          4;0  ,   RCT_genC_Fx ,      4;0   ,        "Rule Fx"
      6  , RCT_genB_x1  ,          5;0  ,            x1 ,      5;0   ,        "Rule x1"
      7  , RCT_genB_x2  ,          6;0  ,            x2 ,      6;0   ,        "Rule x2"
```

Example 4-4 shows a customized version of the same rule-map file.

Example 4-4 Updated or Edited Rule-Map File

```
RuleGroup, BaselineRule, Baseline(L;D), ComparatorRule, Comparator(L;D), Description
      1  ,      F      ,        0;0  ,          F      ,        0;0  ,      "Rule F"
      2  ,      F.1    ,        1;0  ,          F_1    ,        1;0  ,      "Rule F.1"
      3  ,      B      ,        2;0  ,          BR     ,        3;0  ,      "Rule B"
      4  ,      FX     ,        4;0  ,          FX1    ,        5;0  ,      "Rule Fx1"
      4  ,             ,             ,          FX2    ,        6;0  ,      "Rule Fx2"
```

When you modify a rule-map file, ensure that

- Every line in the file contains six comma-separated fields, specifies a rule group ID, and has at least one baseline rule identifier or comparator rule identifier.
- Every rule identifier specifies a corresponding *layer*;*datatype* assignment.

  For example, if you specify a baseline rule identifier, you must also specify a *baseline-layer*;*datatype* assignment.

- Every unique rule group identifier has at least one baseline rule identifier and at least one comparator rule identifier.

Note:
Although the rule descriptions in the last field of each line are optional, you should include rule descriptions because the tool needs them to generate meaningful report information.

Comparison Results File

The DCV Results Compare tool produces a comparison report that summarizes the details of how the comparator results correlate with the baseline results. This report contains a report header, a comparison summary, and a detailed comparison results section.

- The report header contains information about the tool release version, the total number of errors (from the baseline and comparator input), and the matched errors and unmatched errors (discrepancies) per rule.
- The comparison summary section summarizes each rule, indicating whether the rule has passed the comparison criteria or failed due to discrepancies.
- The detailed comparison results section gives detailed information about each rule. The section includes rule comments, rule identifiers, and any missed or false errors (discrepancies).

The report file name takes the form *top_cell*.RCT.report, where *top_cell* is the name of the top cell in the design.

For more information, see the following sections:

- Report Header
- Comparison Summary
- Detailed Comparison Results

Report Header

The report header contains the following information:

- Banner Information, which contains the tool release version
- Tool input information such as the baseline, comparator, and layout files and the tool CALLED as information

Comparison Summary

The comparison summary section summarizes each rule, indicating whether the rule has passed the comparison criteria or failed due to discrepancies. The summary includes

- The rule group numbers, baseline and comparator rule identifiers, results (PASS or FAIL), total baseline errors, and total comparator errors
- The number of discrepancies in the comparator results that are not in the baseline results (false errors).
- The number of discrepancies in the baseline results that are not in the comparator results (missed errors).

The results in this section are organized in a decreasing order of priority:

- Rules with the largest number of missed errors and no false errors
- Rules with the largest number of false errors and no missed errors
- Rule with the greatest difference between missed and false errors
- Rules with no discrepancies (passing results)

Two or more rules that have equal priorities are ordered based on their rule-group IDs.

Example 4-5 shows an example of comparison report summary. The summary results are sorted based on severity. The Missed Errors column shows the number of error results in the baseline data that are not in the comparator data. The False Errors column shows the number of error results in the comparator data that are not in the baseline data.

Example 4-5 Comparison Report Summary Example

```
          ------------------
          COMPARISON SUMMARY
          ------------------

Rule           Baseline     Comparator Result     Total      Total      Missed      False
Group         Rule Name      Rule Name          Baseline Comparator     Errors     Errors

   3             RULE_2         RULE_2   FAIL       2739       3528        850          0
  14             RULE_4         RULE_4   FAIL        851        460        209          0
  48             RULE_F         RULE_F   FAIL        228         96        120          0
   9            RULE_A1        RULE_A1   FAIL          3          2         50          0
  18            RULE_B1        RULE_B1   FAIL          9          9         24          0
  32             RULE_C         RULE_C   FAIL         11          1          5          0
  26            RULE_6b        RULE_6b   FAIL      12467    1589096          0     125670
   5             RULE_8         RULE_8   FAIL       2735     633190          0      96500
  16             RULE_H         RULE_H   FAIL        417      64417          0       3015
  17            RULE_SS        RULE_SS   FAIL        328       1314         39        589
  21            RULE_SX        RULE_SX   FAIL         16         16         12         64
  22            RULE_S2        RULE_S2   FAIL        370        371         30          1
  20            RULE_Se        RULE_Se   FAIL      18497        481        108        124
  23           RULE_Se3       RULE_Se3   PASS        299        299          0          0
```

In this example,

- The first six rule groups in the example have errors in the baseline data but none in the comparator data. The rules are sorted by number of discrepancies.
- The next three rule groups have errors in the comparator data but not in the baseline data. The rules are sorted by number of discrepancies.
- The next four rule groups have discrepancies in both the baseline and comparator data. The rules are sorted by largest delta to smallest delta.
- For the last rule group, the comparison matched. These results are sorted by the number of errors in the baseline data (Total Baseline).

Detailed Comparison Results

The detailed comparison results section gives detailed information about each rule, including rule comments, rule identifiers, and any missed or false errors. The results are similar to the presentation of errors in a LAYOUT_ERRORS file; the main difference is the details provided with the discrepancies.

Note:
The output report format can change depending on the command-line options that you specify.

The report displays basic rule group information and the results for each missed and false error for each rule group ID in ascending order from 1 to *N*. Example 4-6 shows the detailed results for a rule group with both missed and false errors.

Example 4-6 Comparison Report Detailed Results Section

```
          ---------------------------
          DETAILED COMPARISON RESULTS
          ---------------------------

Rule Group: 1
Baseline Rule Names: Rule_ABC
Comparator Rule Names: RCT_Rule_ABC
Description: rule_comment_or_description
Compare Result: FAIL
-------------------------------------------------------------------------

Missed errors: 8
-----------------------------------------------------------
    RCT (lower left x, y) (upper right x, y)
-----------------------------------------------------------
    (683.8800, 1864.2140) (683.9600, 1864.9060)
    (688.6800, 1864.2140) (688.7600, 1864.9060)
    (683.8800, 1858.4540) (683.9600, 1860.4800)
    (679.0800, 1858.9200) (679.1600, 1860.4800)
    (678.7600, 1857.1200) (678.8400, 1858.6800)
    (664.3600, 1864.2140) (664.4400, 1864.9060)
    (663.4000, 1858.4540) (663.4800, 1860.4800)
    (661.1600, 1858.4540) (661.2400, 1860.4800)

False errors: 6
-----------------------------------------------------------
    RCT (lower left x, y) (upper right x, y)
-----------------------------------------------------------
    (683.8800, 1864.2140) (683.9600, 1864.9060)
    (688.6800, 1864.2140) (688.7600, 1864.9060)
    (683.8800, 1858.4540) (683.9600, 1860.4800)
    (679.0800, 1858.9200) (679.1600, 1860.4800)
    (678.7600, 1857.1200) (678.8400, 1858.6800)
    (664.3600, 1864.2140) (664.4400, 1864.9060)
```

If you specify a waiver database, the report can contain additional sections for waived comparison discrepancies. Example 4-7 shows an example of waived comparison discrepancies.

Example 4-7 Comparison Report Detailed Results for Waived Discrepancies

```
        --------------------
        WAIVED DISCREPANCIES
        --------------------

Rule Group: 21
Baseline Rule Names: Rule_BCD
Comparator Rule Names: RCT_Rule_BCD
Description: rule_comment_or_description
Compare Result: WAIVED
-------------------------------------------------------------------------

Missed errors waived: 2
-----------------------------------------------------------
    RCT (lower left x, y) (upper right x, y)
-----------------------------------------------------------
    (588.2150, 451.7400) (588.2200, 451.7450)
    (588.2100, 451.7350) (588.2150, 451.7400)

False errors waived: 0
```