SIEMENS EDA

Calibre® MPCverify User's and Reference Manual

Software Version 2021.2

SIEMENS

Support Center: support.sw.siemens.com
Send Feedback on Documentation: support.sw.siemens.com/doc_feedback_form

Table of Contents

List of Figures

List of Tables

Chapter 1
Introduction to Calibre MPCverify

Calibre® MPCverify is a grid-based mask process simulator and MPC results verification tool that is designed to predict a mask manufacturing process. It supports the industry-standard MPC usage models and seamlessly integrates with other Calibre tools.

Calibre MPCverify Quickstart

You use Calibre MPCverify as a programming language, building one or more Verification Rule checks to run a simulation. Calibre MPCverify requires a previously-created e-beam and etch model files for use in the simulations.

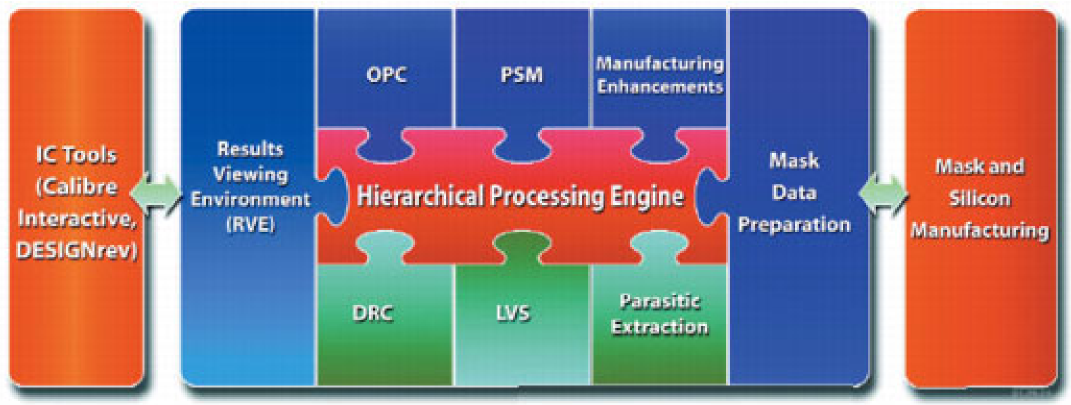
Table 1-1. Calibre MPCverify Quickstart Table

You Perform These Tasks	1. Write a Calibre MPCverify command setup file to set up your environment. 2. Write an SVRF rule file. 3. Run with Calibre® nmDRC™. 4. Examine results.
Related Information — Learn How to Use Calibre MPCverify	
Read First	• “Calibre MPCverify Application Examples” on page 33
Then Read	• “Using Calibre MPCverify” on page 23
	Contains detailed usage information for Calibre MPCverify in Batch Mode, including in-depth coverage of sections covered in this chapter.
Command Dictionary	“Calibre MPCverify Function Reference” on page 45

Calibre MPCverify Workflow

Calibre MPCverify is part of a suite of tools for the Calibre Mask Data Preparation (MDP) flow. Calibre MDP allows for seamless continuation of the data manipulations required for Resolution Enhancement Technology (RET) techniques to the mask data format conversion in one batch run, keeping data hierarchically represented as long as possible.

Figure 1-1. Calibre Design-To-Silicon Flow



The Calibre MDP suite of tools include:

- Calibre® MDP Embedded SVRF™ — Allows you to insert a block of Standard Verification Rule Format (SVRF) commands in a FRACTURE, Calibre MDPverify, MDP EMBED, or DENSITY_CONVOLVE invocation. SVRF commands are documented in the *Standard Verification Rule Format (SVRF) Manual*.
- Calibre® MDP EMBED™— Provides section-based processing for some SVRF commands, essentially providing section-based DRC capability. Section-based processing is an alternate method of layout processing that may improve scalability and turnaround time for some limited cases such as mask rule checks (which are DRC-like checks on mask data).
- Calibre® MDPverify™ — Allows you to evaluate fractured MDP data by performing geometrical comparisons of fractured data output to the original layout data, database layer, or fractured data in a different pattern file.
- Calibre® MDPmerge™ — For VSB11 only, reads a VSB11 job deck and merges individual chips and chip placements into a single chip and placement, respectively, based on certain criteria. The output is a new job deck consisting of the merged chips and copies of chips that did not meet the merge criteria.
- Calibre® MDPstat™ — Analyzes vector beam data such as NUFLARE, JEOL, Hitachi, and OASIS.MASK formatted data allowing you to assess the quality of results of the fracture.

- Calibre® MPCpro™ — A rule-based MPC application that applies pattern-based mask process corrections specified by SVRF commands to your design data prior to fracturing. This includes correcting linearity and proximity effects as well as longer-range processing effects such as fogging and develop- and etch-loading.
- Calibre® MASKOPT™ — Modifies the input geometry to reduce shot count and improve mask write time through rule-based vertex alignment.
- Calibre® nmMPC™ and Calibre® nmCLMPC — Calibre nmMPC is a suite of functions used for modeling, simulating, and correcting distortions of the mask manufacturing process. Calibre nmCLMPC performs the same functions, but is targeted for curvilinear layouts with skew edge polygons. Models generated by this process are utilized by Calibre nmMPC, Calibre nmCLMPC, and FRACTURE tools to accurately reflect the final output of the mask process. Refer to the *Calibre nmMPC and Calibre nmCLMPC User's and Reference Manual* for further information.
- Calibre® MDPview™ — A Graphical User Interface (GUI) viewer that enables you to visually and interactively inspect the results of fracture and verify operations (pattern files, job decks), as well as large post-tape-out OASIS®[1] files. Calibre MDPview utilities are TCL-based commands that allow script-based manipulation of fractured data and job decks, and include conversion of pattern files or job decks to OASIS (job smashing), conversion of MEBES job decks from MEBES to other formats, and pattern file quality, optimization and informational utilities. This tool is documented in the *Calibre MDPview User's and Reference Manual*.
- Calibre® Job Deck Editor™ — A tool invoked from Calibre MDPview that is used to generate an optimized chip placement on the wafer, with the ability to manually edit chips in the job deck. This tool is documented in the *Calibre Job Deck Editor User's Manual*.
- Calibre® MDPDefectAvoidance™ — A tool invoked from Calibre MDPview that is used to find shifts in a layout to prevent extreme ultraviolet lithography (EUVL) blank mask defects from appearing on patterns. This tool is documented in the *Calibre MDPDefectAvoidance User's Manual*.
- Calibre® DefectReview® — A tool that provides efficient analysis, classification, and trend analysis of defects identified by mask inspection systems. The software includes features for easy and fast defect navigation, visual display, defect selection and filtering, defect classification, clustering, sophisticated CD analysis, analysis over multiple inspections, and repeatability and trend analysis. This tool is documented in the *Calibre DefectReview User's Manual*.

1. OASIS® is a registered trademark of Thomas Grebinski and licensed for use to SEMI®, San Jose. SEMI® is a registered trademark of Semiconductor Equipment and Materials International.

- Calibre® MDPAutoClassify® — A tool that is used for automatic classification of defects observed on a blank substrate before any patterning is performed on the substrate. This tool is documented in the *Calibre MDPAutoClassify User's Manual*.
- Calibre® DefectClassify™— A tool that enables automatic classification of defects observed on a patterned mask. This tool is documented in the *Calibre DefectClassify User's Manual*.

For more information on the tools in this workflow, see the *Calibre Post-Tapeout Flow User's Manual*.

The following table lists all related Calibre products and their documentation.

Table 1-2. Related Products and Their Manuals

Related Products	Documentation
All post-tapeout products	*Calibre Post-Tapeout Flow User's Manual*
Calibre® FRACTUREc™ Calibre® FRACTUREh™ Calibre® FRACTUREi™ Calibre® FRACTUREj™ Calibre® FRACTUREm™ Calibre® FRACTUREn™ Calibre® FRACTUREp™ Calibre® FRACTUREt™ Calibre® FRACTUREv™ Calibre® MDPmerge™ Calibre® MDPstat™ Calibre® MDPverify™ Calibre® MPCpro™ Calibre® MASKOPT™ Calibre® MDP Embedded SVRF	*Calibre Mask Data Preparation User's and Reference Manual* *Calibre Release Notes*
Calibre® nmMPC™ Calibre® nmCLMPC Calibre® nmOPC™	*Calibre nmMPC and Calibre nmCLMPC User's and Reference Manual* *Calibre nmOPC User's and Reference Manual*

Table 1-2. Related Products and Their Manuals (cont.)

Related Products	Documentation
Calibre® MDPview™	*Calibre MDPview User's and Reference Manual* *Calibre DESIGNrev Layout Viewer User's Manual* *Calibre Release Notes*
Calibre® nmDRC™ Calibre® nmDRC-H™	*Calibre Release Notes* *Calibre Verification User's Manual* *Standard Verification Rule Format (SVRF) Manual*
Calibre® WORKbench™	*Calibre WORKbench User's and Reference Manual*
Tcl/Tk Batch Commands	*Calibre DESIGNrev Reference Manual*
Calibre® Metrology API (MAPI)	*Calibre Metrology API (MAPI) User's and Reference Manual*
Calibre® Metrology Interface	*Calibre Metrology Interface (CMi) User's Manual*
Calibre® Job Deck Editor	*Calibre Job Deck Editor User's Manual*
Calibre® MDPDefectAvoidance™	*Calibre MDPDefectAvoidance User's Manual*
Calibre® OPCverify™	*Calibre OPCverify User's and Reference Manual*
Calibre® DefectReview™	*Calibre DefectReview User's Manual*
Calibre® MDPAutoClassify™	*Calibre MDPAutoClassify User's Manual*
Calibre®DefectClassify™	*Calibre DefectClassify User's Manual*

Calibre MPCverify Prerequisites

Before you run Calibre MPCverify, there are a number of prerequisites that you will need first.

Users who intend to run Calibre MPCverify should be familiar with Calibre procedures for MPC, OPC, and DRC. An understanding of programming Standard Verification Rules Format (SVRF) rule decks and Tcl is highly useful.

You will also require the following to run Calibre MPCverify:

- **Licensing** — The Calibre MPCverify license is required to run this tool. In addition, viewing the results of a Calibre MPCverify run requires a Calibre WORKbench license. Refer to the *Calibre Administrator's Guide* for more information on licensing these products.
- **Platform support** — Calibre MDPview is available on all supported platforms found in the *Calibre Administrator's Guide*. Refer to that document for instructions on how to install Calibre software.
- **Required files —** The following files are required to run Calibre MPCverify:
    - An SVRF file to call the Calibre MPCverify command setup file
    - A Calibre MPCverify command setup file to define the Calibre MPCverify operating parameters and run one or more user-programmed verification rule checks.

Once you have the prerequisites, you run Calibre MPCverify by entering the following command in a console window:

```
% calibre -drc -hier -turbo -turbo_litho svrf_filename
```

Calibre MPCverify Key Concepts

There are several key concepts that must be understood prior to using Calibre MPCverify.

Calibre MPCverify Setup File

Calibre MPCverify command setup files are text files. They contain configuration commands and setlayer commands.

This is an example Calibre MPCverify setup file:

```
LITHO FILE mpcv [/*
 processing_mode flat
 tilemicrons 75
 modelpath                  models
 etch_model_load   etch0    etch.mod
 ebeam_model_load  ebeam0   ebeam.mod

 layer Target
 layer MpcDL0
 layer MpcDL1

 # Simulate contour
 setlayer sim = ebeam_simulate  MpcDL0 ebeam_model ebeam0 dose 1.0 \
                                MpcDL1 ebeam_model ebeam0 dose 1.5 \
                                etch_model etch0

 # Line ends epe check
 setlayer epeLE =  measure_epe sim Target inside lineEnds \
                   epe_spacing 0.005 function max \
                   min_featsize 0.030 max_edgelen 0.35\
                   fragment only not > -0.0002 < 0.0002 \
                   output_expanded_edges 0.001 trim_ends 0.002 \
                   property { max
                   } classify {
                       context Target
                       halo 0.05
                       score bin_size 0.001
                       worst 50 exact}
*/]
```

Configuration commands set up the working environment for Calibre MPCverify.

Table 1-3. Required Configuration Command Tasks

Task	Command
Set model path	modelpath

Table 1-3. Required Configuration Command Tasks (cont.)

Task	Command
Import ebeam model	ebeam_model_load
Import etch model	etch_model_load
Set up layers	background[1] and layer

1. The background command is required for syntax compliance only and is not otherwise used by Calibre MPCverify. However, you must always define background opposite to the layer (for example, background set to dark and layer set to clear), otherwise the tool will issue a syntax error.

Setlayer commands perform Calibre MPCverify operations. Setlayer commands all have the form:

```
setlayer name = operation arguments
```

Calibre MPCverify supports three types of setlayer commands:

- Contour generation
- DRC-type layer manipulation commands
- Verification commands, which include polygon modification and check commands

Important Calibre MPCverify Setlayer Operations

Most users of Calibre MPCverify will probably utilize the following verification control operations.

Note

This is a subset of the full operations list; the complete list and descriptions of the Calibre MPCverify operations can be found in the section "Setlayer Operations Reference" for the batch commands.

Calibre MPCverify Generators

Generators create needed data (usually context layers) for subsequent Calibre MPCverify checks.

Contour Generators

Use the following generators to build contours around target layer shapes:

Table 1-4. Contour and Tolerance Zone Generators

Operation	Description
ebeam_simulate	Simulates a contour with ebeam (or ebeam plus etch models). Many verification checks require a contour layer.
veb_simulate	Applies an etch bias based on an etch model to a layer.

Filter Generators

Use filter layers to designate areas of interest, in order to exclude other areas from the computational checks.

Table 1-5. Filter Generators

Operation	Description
filter_generate	Generates a filter layer for the measure_cd command.
identify_corner	Identifies line fragments of edges near corners.
identify_edge	Identifies line ends or jogs.

Verification Checks

Use verification checks on constructed context layers to find results.

Table 1-6. Verification Checks

Operation	Description
contour_diff	Calculates the error factor between two contours.
measure_cd	Checks width and space CD accuracy.
measure_epe	Checks if a layer's EPE fails a specified constraint.
nilscheck	Checks the Normalized Image Log-Slope (NILS) for two or more contours for a specified range constraint.

Example Rule File Syntax

The SVRF rule file contains all instructions to pass to the Calibre engine.

The following is an example SVRF rule file:

```
LAYOUT SYSTEM OASIS

// --- Read in OPCed data
LAYOUT PATH    "$MPC_OUT"
LAYOUT PRIMARY "*"
LAYOUT INPUT EXCEPTION SEVERITY PRECISION_RULE_FILE 0
LAYOUT ULTRA FLEX YES 256
precision 10000

// --- Output oasis file
DRC RESULTS DATABASE "$MPCV_OUT" OASIS PSEUDO CBLOCK
DRC MAXIMUM RESULTS ALL   // write all results to the output

// --- Read input layers
LAYER M0              100
LAYER SB_layer        101
LAYER SB_mpc          102
LAYER main_feature    103

// --- Define norminal dose value
VARIABLE DV0   1.00    // nominal dose

//output contour
sim = LITHO MPCVERIFY  M0 SB_layer SB_mpc main_feature  \
                       MAP sim  FILE mpcv
sim {COPY sim  } DRC CHECK MAP sim 900 0

// --- EPE checks for Line Ends, Space Ends, and all others
epeLE    = LITHO MPCVERIFY  M0 SB_layer SB_mpc main_feature  \
                            MAP epeLE  FILE mpcv
epeRest  = LITHO MPCVERIFY  M0 SB_layer SB_mpc main_feature  \
                            MAP epeRest FILE mpcv

// Send the ascii format outputs to an RVE database.
rdb_epeLE   {DFM RDB epeLE "$epe_out" MAXIMUM ALL ALL CELLS NOEMPTY \
            CHECKNAME epeLE      }
rdb_epeRest {DFM RDB epeRest "$epe_out" MAXIMUM ALL ALL CELLS NOEMPTY \
            CHECKNAME epeRest    }
```

```
// --- MPC Verification LITHO FILE
LITHO FILE mpcv [/*
  svrf_var_import DV0 DV0p5 DV0m5
  tilemicrons 100
  processing_mode flat
  modelpath                  $env(MODELS_DIR)
  etch_model_load   etch0    $env(ETCH_MOD)
  ebeam_model_load  ebeam0   $env(EBEAM_MOD)

  layer M0
  layer SB_layer
  layer SB_mpc
  layer main_feature

# --- Generate filters for SB only
  setlayer lineEnds     = identify_edge SB_layer length<= 0.120 \
                          length1 >= 0.15 length2 >= 0.15 \
                          corner1 convex corner2 convex extend -0.035 \
                          expand 0.004
  setlayer corners      = filter_generate SB_layer expand 0.005 \
                          convex 0.035 concave 0.035
  setlayer allFilters   = or lineEnds corners

# --- Simulate contour
  setlayer sim     = ebeam_simulate \
                     SB_mpc ebeam_model ebeam0 dose $DV0\
                     main_feature ebeam_model ebeam0 dose $DV0 \
                     etch_model etch0

# --- Line ends
  setlayer epeLE  = measure_epe sim SB_layer inside lineEnds\
                     epe_spacing 0.005 function  max min_featsize 0.030 \
                     max_edgelen 0.35 fragment only not > -0.001 < 0.001 \
                     output_expanded_edges 0.001 trim_ends 0.002 \
                     property { max
                     }  classify {
                          context SB_layer
                          halo 0.05
                          anchor SB_layer
                          score bin_size 0.001
                          worst 50 exact}

# --- Measure EPE outside all filte
  setlayer epeRest = measure_epe sim SB_layer outside allFilters \
                      epe_spacing 0.005 function average \
                      min_featsize 0.030 max_edgelen 0.35 \
                      fragment only not > -0.001 < 0.001 \
                      output_expanded_edges 0.001 trim_ends 0.002 \
                      property { average
                      }  classify {
                           context SB_layer
                           halo 0.05
                           anchor SB_layer
                           score bin_size 0.002
                           worst 50 exact}
*/]
```

Call Calibre MPCverify using a standard SVRF rule file, with the following requirements:

- LAYER statements that map to the design file layer numbers.
- One or more LITHO MPCVERIFY calls to the Calibre MPCverify command setup file to generate derived layers.
    - The order you give the names of the input layers to the command determines the order of their assignment in the Calibre MPCverify setup file.
    - The name of a context layer in the Calibre MPCverify setup file must match the derived layer name.
- Rule checks to analyze the results.

Syntax Conventions

The command descriptions use font properties and several metacharacters to document the command syntax.

Table 1-7. Syntax Conventions

Convention	Description
Bold	Bold fonts indicate a required item.
Italic	Italic fonts indicate a user-supplied argument.
Monospace	Monospace fonts indicate a shell command, line of code, or URL. A bold monospace font identifies text you enter.
<u>Underline</u>	Underlining indicates either the default argument or the default value of an argument.
UPPercase	For certain case-insensitive commands, uppercase indicates the minimum keyword characters. In most cases, you may omit the lowercase letters and abbreviate the keyword.
[ ]	Brackets enclose optional arguments. Do not include the brackets when entering the command unless they are quoted.
{ }	Braces enclose arguments to show grouping. Do not include the braces when entering the command unless they are quoted.
‘ ’	Quotes enclose metacharacters that are to be entered literally. Do not include single quotes when entering braces or brackets in a command.
| or ||	Vertical bars indicate a choice between items. Do not include the bars when entering the command.

Table 1-7. Syntax Conventions (cont.)

<table>
<tr><th>Convention</th><th>Description</th></tr>
<tr><td>…</td><td>Three dots (an ellipsis) follows an argument or group of arguments that may appear more than once. Do not include the ellipsis when entering the command.</td></tr>
<tr><td colspan="2">Example:
  DEVice {<i>element_name</i> [‘(’<i>model_name</i>‘)’]}
    <i>device_layer</i> {<i>pin_layer</i> [‘(’<i>pin_name</i>‘)’] …}
    [‘<’<i>auxiliary_layer</i>‘>’ …]
    [‘(’<i>swap_list</i>‘)’ …]
    [<u>BY NET</u> | BY SHAPE]</td></tr>
</table>

Chapter 2
Using Calibre MPCverify

Calibre MPCverify is primarily run in batch command mode and requires input scripts. It is recommended that you have knowledge of SVRF commmand scripting prior to using Calibre MPCverify.

Usage Flow for Calibre MPCverify

The usage model for Calibre MPCverify requires you to follow a specific flow.

Usage Model Process Flow

Tip

Setup file syntax is *case-sensitive*.

Step 1 - Create a Calibre MPCverify Setup File

You configure Calibre MPCverify from a setup file (or a setup file block inside an SVRF rule file) that you create, using the following steps:

1. Select one ebeam model.
2. Select one etch model.
3. Define layers in the design for Calibre MPCverify:
    a. Specify input layers using the layer command.
    b. Specify Calibre MPCverify and SVRF output layers using the setlayer command.
4. Create rule checks inside the setup file using the setlayer command operations.

Step 2 - Create SVRF Rule File Calls

You run Calibre MPCverify from an SVRF file (also known as a “rule file”) that you create, using the following steps:

1. Define the layers in the design for SVRF for use as input layers.
2. Define one or more LITHO MPCVERIFY calls, deriving them as SVRF output layers from the input layers. The LITHO MPCVERIFY calls use the setup file you created in “Step 1 - Create a Calibre MPCverify Setup File” as their FILE argument.
3. Add one or more DRC CHECK MAP calls to output the derived layers to an ASCII or OASIS database file.
4. Optionally, add additional rule checks in the SVRF file, such as using scoring checks or DFM ANALYZE.
5. Run Calibre on the SVRF rule file.

Calibre MPCverify Setup File Creation

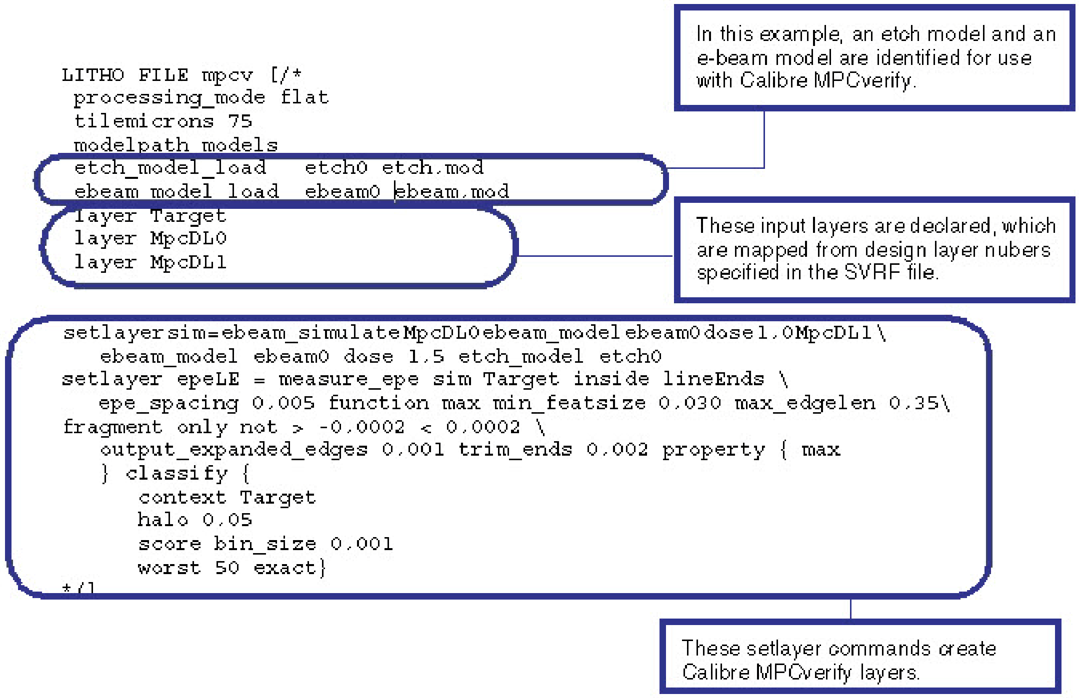
Calibre MPCverify uses a specific setup file (in ASCII text) that contains a block of configuration parameters and operations that you want Calibre MPCverify to perform. It is not the same as the setup file used by Calibre WORKbench; you will be calling this file exclusively from Calibre MPCverify.

Note

Arguments for the Calibre MPCverify setup file can be found on the LITHO MPCVERIFY syntax page.

The following is an example Calibre MPCverify setup file:

Figure 2-1. Calibre MPCverify Setup File Example (mpcverify.in)



Selecting E-Beam Models

You must supply an E-beam model (either individually or as part of a litho model) to Calibre MPCverify.

E-beam models simulate the behavior of the partial e-beam effect (forward scattering and blur effect, the PEC and fogging correction usually applied on an EBEAM writer). Calibre uses this information to calculate the e-beam lithography effects.

- Specify optical models with the **ebeam_model_load** setup file command:

  ```
  ebeam_model_load model_label filename
  ```

- You may need to supply a pathname to the **modelpath** setup file command for Calibre to find your model directory.

Selecting Etch Models

You only need to supply an etch model to Calibre MPCverify if you want to account for etch effects.

Etch models simulate the behavior of etch loading and micro-loading effects of the resist and absorber during the etch process. Calibre uses this information to calculate the etch effects on the design.

- Specify etch models with the **etch_model_load** setup file command:

  ```
  etch_model_load etch_model_label filename
  ```

- You may need to supply a pathname to the **modelpath** setup file command for Calibre to find your model directory name.

Calibre MPCverify Layer Definition in the Design

A design file consists of one or more named and numbered layers, which represent polygons in a topographical fashion. Layers can also be MPC correction layers, SRAF layers, or other non-printing layers (such as for text information).

- The Calibre MPCverify tool requires the identification of one or more of the layers in the design as input layers.
- Calibre MPCverify can create SVRF output on one or more output layers, when used in conjunction with the DRC CHECK MAP SVRF command.
- Calibre MPCverify also uses special layers for its own operations.

You can visually inspect layers (original/input and results/output) in Calibre WORKbench.

Layer Mapping Operations

Layers in the design must be mapped to identify them for processing by Calibre MPCverify.

Mapping Input Layers for Calibre MPCverify

Use the **layer** setup file command to identify the input layers present in your design, such as in the following example:

```
layer mask visible dark
```

- The order you define your layers in is important; each name will later be linked to the layer statement by the positional order that the SVRF rule file LITHO MPCVERIFY calls use.
- The defined layer inputs must match the layers defined in the design that you run Calibre MPCverify on, or Calibre returns an error.

Once you define layers, you will be able to later use them to receive input layer arguments from the SVRF file call to Calibre MPCverify:

```
derived_layer_name = LITHO MPCVERIFY input_layer1 ... input_layerN FILE
setup MAP output_layer
```

Remember, Calibre MPCverify uses the input layers in the order given (not their layer number or name in the design file) for its computations.

This mapping-by-order method used by Calibre MPCverify is an enhancement over other Calibre LITHO commands (such as Calibre® OPC™, Calibre® ORC™, and Calibre® PRINTimage™), which require the setup file layer names to match the SVRF layer names. The enhanced method allows you to write setup file layer statements using generic names that are not required to match the SVRF layer names passed in to Calibre MPCverify, which means you can more easily re-use different existing SVRF files with different Calibre MPCverify setup files.

Example (Mapping Input Layers)

The following example code is placed in the Calibre MPCverify file, *mpcverify.in*:

```
ebeam_model_load ebeam0 ebeam.mod
background clear
layer m1 visible dark
setlayer image1 = ebeam_simulate m1 ebeam_model ebeam0 dose 1.0
```

In the SVRF file, the following call code is placed:

```
L1 = LITHO MPCVERIFY m1 FILE mpcverify.in MAP image1
```

Mapping Calibre MPCverify Layers and Output Layers

In this section, you will use the setlayer command to create Calibre MPCverify layers, and learn how to designate an output layer.

You must also define the Calibre MPCverify layer and output layers that will be used in your later SVRF call to Calibre MPCverify, which appears in the SVRF file as follows:

```
output_layer = LITHO MPCVERIFY input_layer1 ... input_layerN FILE setup MAP
MPCverify_layer
```

Calibre MPCverify Layers

Many Calibre MPCverify operations rely on manipulating the input layers using the setlayer commands. These commands fall into one of three categories:

- The ebeam_simulate command, which generates contour layers from an input layer
- DRC-type commands, which perform geometrical Boolean and measurement operations
- Verification commands, which are a number of utility commands to create and check design areas from input and contour layers

  Some verification operations can generate error layers; others are designed to be used as intermediate generation steps.

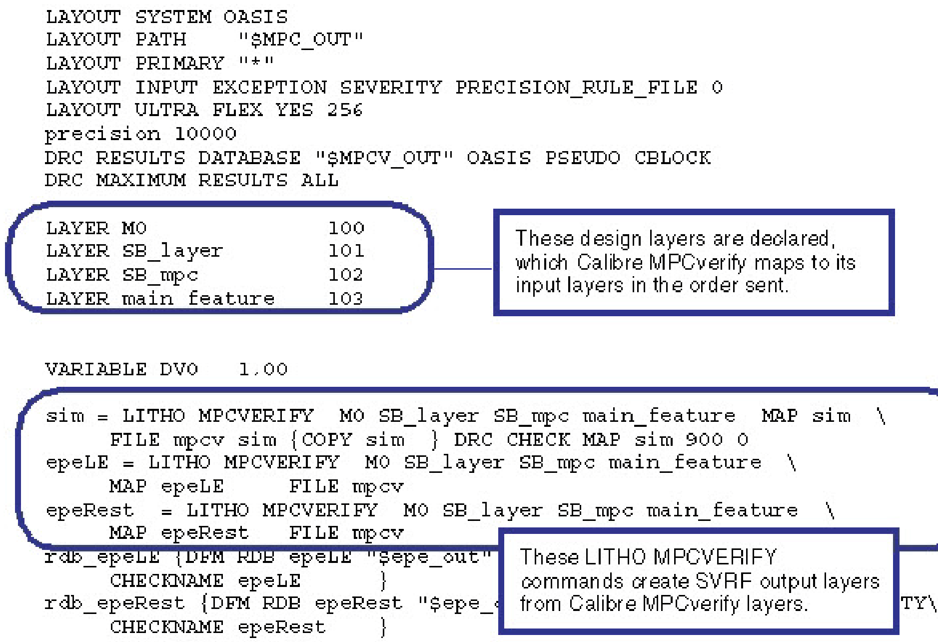
All three types of commands use the same syntax:

```
setlayer MPCverify_layer_name = layer_operation
```

Calibre MPCverify Rule File Creation

You use the SVRF rule file as input to Calibre. These instructions that you understand the basic construction of a rule file; the instructions in this section only detail the additional information you will need to run Calibre MPCverify as a Calibre LITHO command.

Figure 2-2. Calibre MPCverify SVRF File Example



Defining MPCverify Operations in the SVRF

Mapping Design Layers

You define the relationship between the design file and Calibre MPCverify using the LAYER SVRF command:

LAYER *label layer#*

The layer label can be anything, as long as it is used consistently in the later calls to LITHO MPCVERIFY.

Note

The layer numbers must correspond to layers that exist in the design file.

Defining LITHO MPCVERIFY Calls

To run LITHO MPCVERIFY, you construct one or more SVRF output layer statements, using the following syntax:

svrf_output_layer = LITHO MPCVERIFY *input_layer_list* FILE *setupfile* MAP *MPCverify_layer_name*

- *svrf_output_layer* is a unique name you will use if you are planning to create rule checks in SVRF.
- *input_layer_list* is the list of design layer names you are passing to Calibre MPCverify, in the order you want them to be used by Calibre MPCverify.

  Remember that Calibre MPCverify maps the inputs (layer commands in the setup file) according to the order you pass them into Calibre MPCverify (LITHO MPCVERIFY commands in the SVRF file), which allows you to re-use existing SVRF rule files without renaming layers in either the rule file or Calibre MPCverify setup file.

  Note

  A common error is to pass the layer names into your Calibre MPCverify file in the wrong order.

- *setupfile* is the fully qualified pathname and filename of the Calibre MPCverify file you created earlier.
- ***MPCverify_layer_name*** is the name of any Calibre MPCverify layer you created inside the setupfile (setlayer command). Each LITHO MPCVERIFY statement can have only one Calibre MPCverify layer mapped to it, as described in the following section.

  Note

  After a call to LITHO MPCVERIFY, you can also further manipulate the derived SVRF layer as you would any other SVRF output layer (some Calibre MPCverify output layers may not be suitable for further modification, however).

Understanding Calibre MPCverify to SVRF Output Layers

You instruct Calibre to map outputs from Calibre MPCverify into SVRF layers using the MAP command inside each LITHO MPCVERIFY command, in the SVRF rule file.

You can only use the LITHO MPCVERIFY MAP (output) statement in the SVRF file on a Calibre MPCverify layer you defined in the Calibre MPCverify setup file, using the **setlayer** Calibre MPCverify command:

setlayer *MPCverify_layer_name* = *layer_operation*

Calibre automatically prunes any setlayer commands whose output is not in the dependency graph of outputs selected using MAP from the execution graph. The pruned layers are not run, and do not contribute to the runtime.

In this way, MAP is similar to DRC SELECT CHECK in a rule file.

Example (Output Layers)

In this example, the **setlayer** command produces a Calibre MPCverify layer called “image1.” Calibre maps this layer to the output SVRF layer L1 using the MAP command on the LITHO MPCVERIFY command.

```
LITHO FILE MPCverifysetup [
  ebeam_model_load ebeam0 ebeam.mod
  background clear
  layer mask visible dark
  layer sraf visible dark

  setlayer image1 = ebeam_simulate mask ebeam_model ebeam0 dose 1.0\
                    sraf ebeam_model ebeam0 dose 1.5
]
L1 = LITHO MPCVERIFY POLY_OPC SBARS_OPTIMAL FILE MPCverifysetup MAP image1
```

If your Calibre MPCverify layer has properties attached, you must also add a COPY statement to add the layer containing the properties to the layout:

```
L1 {COPY L1} DRC CHECK MAP L1 404
```

Attempting to use a layer with properties without using a COPY statement on it first will result in an error.

Add Database and Output Reporting

Some Calibre MPCverify commands can output polygons with attached properties as a result of function calls.

- If your output does not contain any property reporting arguments, use the COPY and DRC CHECK MAP commands to output the results of a LITHO MPCVERIFY call to an ASCII (or OASIS) RDB database file.

  ```
  L1 {COPY L1} DRC CHECK MAP L1 202 // copies the output to layer 202
  DRC CHECK MAP L1 ASCII "verify_output2.asc"
                                   // sends output to a file
  ```

- If you want to generate an output file that contains the properties assigned by Calibre MPCverify, you must use a DFM RDB statement inside of a rule check to capture the properties.

  ```
  rdb_fetch_property { DFM RDB min_with_props \
                      "verify_output2.asc" NOEMPTY}
  ```

  This example code assumes that you have an result layer “min_with_props” with attached properties inside your setup file.

Concurrency Execution of Multiple Calibre MPCverify Statements in Calibre

Concurrency mechanisms allow Calibre to complete operations faster by taking advantage of operations that can be run independently.

Generating Multiple Output Layers

Calibre MPCverify can generate multiple output layers in a single run, using a concurrency mechanism. The concurrency mechanism is triggered automatically when multiple LITHO MPVERIFY layer operations are contained in the SVRF call that have exactly the same arguments, but have different MAP outputs.

Example (Multiple Output Layers)

```
L1 = LITHO MPCVERIFY POLY ASSIST FILE setup MAP image1
L2 = LITHO MPCVERIFY POLY ASSIST FILE setup MAP image2
L3 = LITHO MPCVERIFY POLY ASSIST FILE setup MAP image3
```

In this example, the layers L1, L2, and L3 are run concurrently, because the LITHO MPCVERIFY commands match, except for the MAP statements, which point to different Calibre MPCverify layers.

The following Calibre transcript shows that concurrency was successfully recognized. You will notice that several LITHO MPCVERIFY statements are printed, followed by a single interpreter module message, followed by multiple cell and tile completion information lines.

```
OV1 = LITHO MPCVERIFY MAP i1 POLY TEST FILE MPCverify.in
OV2 = LITHO MPCVERIFY MAP i2 POLY TEST FILE mpcverify.in
OV3 = LITHO MPCVERIFY MAP i3 POLY TEST FILE mpcverify.in
OV4 = LITHO MPCVERIFY MAP i4 POLY TEST FILE mpcverify.in
----------------------------------------------------------------
----- LITHO MPCVERIFY INTERPRETER MODULE -----
----------------------------------------------------------------
[...]
```

Concurrency Execution of Multiple Setlayer Operations in Calibre MPCverify

Concurrency mechanisms allow Calibre MPCverify to complete certain setlayer operations in parallel to reduce runtime.

The following operations support concurrency (see the appropriate syntax page for information):

- ebeam_simulate
- external, internal, and enclosure
- measure_cd
- area_compute and area_ratio
- measure_epe

Setlayer Operations

Within Calibre MPCverify itself, concurrency is supported between certain setlayer commands. This means that in some cases, you can compute multiple “setlayer” commands at the same time. In contrast to Calibre’s layer operations, where concurrency of layer generation operations is automatically calculated, the “setlayer” concurrency within Calibre MPCverify requires you to group together all lines in the setup file which are to be computed concurrently.

To observe which setlayer operations were able to be run concurrently, save the output transcript, and look for the Calibre MPCverify section. A listing of “CONCURRENT GROUP” statements in the transcript indicates setlayer operations that were grouped for concurrency.

Chapter 3
Calibre MPCverify Application Examples

Calibre MPCverify may be best understood by examining sample applications that highlight some of the best practices for MPC verification.

Full Contour-Based Verification With Two Dose Levels

Compared to Optical Process Correction (OPC), Mask Process Correction (MPC) uses additional dose assignments besides edge correction, which can provide more freedom for better CD and edge slope (contrast) control. In an actual design, some areas may have very poor edge slope that will significantly impact the mask quality (for example, the line end of small features like SB or Tip to Tip features). The quality control is very challenging with only edge correction; by applying higher doses for these local areas together with proper edge correction, the mask quality can be greatly improved.

The number of dose levels depends on the e-beam writer. Regular VSB e-beam tool writers only allow one dose level (nominal dose 1.0). Some advanced VSB e-beam writers allow eight or more dose levels. In Calibre nmMPC, rule-based dose assignment is applied. If we apply eight dose levels, eight different layers have to be read into Calibre MPCverify and the appropriate dose level must be assigned; in MPC verification, all the dose parameters used in correction must be included for the contour generation in Calibre MPCverify.

In this example, only two dose levels are assigned in Calibre nmMPC. The nominal dose 1.0 is assigned for the main features (including dummy fill), dose 1.4 is applied to SB. During Calibre MPCverify contour generation, the main features must have dose 1.0 and SB must have dose 1.4.

```
//////////////// RUN CONTROL /////////////////////////
#DEFINE EPECHECKS
// Run the actual EPE checks
#UNDEFINE FILTER_SMALL
// Filters out all sub-resolution MPC test structures (useful when
// running on test chips)
#DEFINE SIMOUT
// Activates output of simulation contours (expensive for large clips)
#UNDEFINE PWOUT
// Activates simulation of process window (+/- 5% dose)
#DEFINE DEBUG
// Activate additional detailed output
//////////////////////////////////////////////////////
LAYOUT SYSTEM OASIS

LAYER IGNORE 999

LAYER MAP >=0 DATATYPE >= 0 999

// --- Read in mask data
LAYOUT PATH    "$VERIF_IN"
LAYOUT PRIMARY "*"

// --- Activate ULTRAflex mode
LAYOUT ULTRA FLEX YES 512
RET GLOBAL LITHOFLAT YES

// --- MPC output should always be at 10k PRECISION
PRECISION 10000

// --- Take pre-bias value from .csh to be consistent with Calibre nmMPC
//     correction recipe
VARIABLE PRE_BIAS ENVIRONMENT

// --- Output oasis file
DRC RESULTS DATABASE "$VERIF_OUT" OASIS PSEUDO CBLOCK
DRC MAXIMUM RESULTS ALL   // write all results to the output

// --- Read in the MPC'ed dose layers
LAYER MAP 10  DATATYPE == 1000  1000
LAYER MpcDL0In                  1000    // Dose level 0
LAYER MAP 10  DATATYPE == 1001  1001
LAYER MpcDL1In                  1001    // Dose level 1
LAYER MAP 10  DATATYPE == 2001  1010
LAYER InFill                    1010    // Biased Fill layer
LAYER MAP 62  DATATYPE == 1000  1100
LAYER InMain                    1100    // Main target layer
LAYER MAP 62  DATATYPE == 1002  1200
LAYER InSraf                    1200    // SRAF target layer

// --- Collect all target polygons on one layer
LAYER MpcTarget                  1100 1200

// --- Need to add (biased) fill pattern to dose layer 0 to get correct
//     contours
MpcDL0All        = OR MpcDL0In InFill

#IFDEF $APPLY_PRE_BIAS
```

```
   // --- Target shapes received pre-bias
   TargetBiasedTmp  = SIZE MpcTarget BY PRE_BIAS
#ELSE
   TargetBiasedTmp  = COPY MpcTarget
#ENDIF

// --- Define dose value for each dose layer
VARIABLE DV0   1.00    // nominal dose
VARIABLE DV1   1.40

// --- Define dose values at +5% dose
VARIABLE DV0p5 1.05    // nominal dose
VARIABLE DV1p5 1.47

// --- Define dose values at -5% dose
VARIABLE DV0m5 0.95    // nominal dose
VARIABLE DV1m5 1.33

// --- Filter line ends
VARIABLE leWmax  0.300  // maximum width of LE
VARIABLE leLmin  0.150  // minimum length to qualify as LE

// --- Remove extremely small lines and spaces that might be on a test
//     chip or because the design was clipped
TargetBiasedMrc1 = SIZE TargetBiasedTmp  BY 0.020 UNDEROVER
TargetBiased     = SIZE TargetBiasedMrc1 BY 0.020 OVERUNDER

#IFDEF FILTER_SMALL
// --- Filter small features which are beyond process limit (only to be
//     used with test chips)
VARIABLE lineFlt  0.045
VARIABLE spaceFlt 0.040
VARIABLE contFlt  0.075

smallFeatures1 = TargetBiased WITH WIDTH <lineFlt
smallContacts  = RECTANGLE TargetBiased <=contFlt BY <=3*contFlt

   smallSpace  = EXTERNAL TargetBiased <=spaceFlt OPPOSITE REGION
   smallAll    = size (OR smallFeatures1  smallContacts smallSpace) \
                 by 1.0
   FltAll      = COPY smallAll
#IFDEF DEBUG
   FltAll      {COPY  FltAll      } DRC CHECK MAP FltAll        333 0
#ENDIF
#ENDIF

#IFDEF DEBUG
// --- write out input layers for debugging purposes
   InFill        {COPY  InFill       } DRC CHECK MAP InFill        10 2001
   MpcDL0In      {COPY  MpcDL0In     } DRC CHECK MAP MpcDL0In      10 1000
   MpcDL1In      {COPY  MpcDL1In     } DRC CHECK MAP MpcDL1In      10 1001
   InMain        {COPY  InMain       } DRC CHECK MAP InMain        62 1000
   InSraf        {COPY  InSraf       } DRC CHECK MAP InSraf        62 1002
#ENDIF


#IFDEF SIMOUT
```

```
   sim         = LITHO MPCVERIFY TargetBiased  MpcDL0All MpcDL1In \
                 MAP sim   FILE mpcv
   sim          {COPY sim           } DRC CHECK MAP sim           900    0
   TargetBiased {COPY  TargetBiased } DRC CHECK MAP TargetBiased  10 2000
#ENDIF

#IFDEF PWOUT
   sim_p5      = LITHO MPCVERIFY TargetBiased  MpcDL0All MpcDL1In \
                 MAP sim_p5    FILE mpcv
   sim_m5      = LITHO MPCVERIFY TargetBiased  MpcDL0All MpcDL1In \
                 MAP sim_m5    FILE mpcv
   sim_p5       {COPY sim_p5        } DRC CHECK MAP sim_p5       900 2
   sim_m5       {COPY sim_m5        } DRC CHECK MAP sim_m5       900 3
#ENDIF

#IFDEF EPECHECKS
// --- EPE checks for Line Ends, Space Ends, and all others
   epeLEAll     = LITHO MPCVERIFY TargetBiased MpcDL0All MpcDL1In \
                  MAP epeLE      FILE mpcv
   epeSEAll     = LITHO MPCVERIFY TargetBiased MpcDL0All MpcDL1In \
                  MAP epeSE     FILE mpcv
   epeRestAll   = LITHO MPCVERIFY TargetBiased MpcDL0All MpcDL1In \
                  MAP epeRest   FILE mpcv
#IFDEF FILTER_SMALL
   epeLE        = NOT  epeLEAll   FltAll
   epeSE        = NOT  epeSEAll   FltAll
   epeRest      = NOT  epeRestAll FltAll
#ELSE
   epeLE        = COPY  epeLEAll
   epeSE        = COPY  epeSEAll
   epeRest      = COPY  epeRestAll
#ENDIF

#IFDEF DEBUG
// --- Output derived layers for debugging
   lineEnds    = LITHO MPCVERIFY TargetBiased MpcDL0All MpcDL1In \
                 MAP lineEnds   FILE mpcv
   spaceEnds   = LITHO MPCVERIFY TargetBiased MpcDL0All MpcDL1In \
                 MAP spaceEnds  FILE mpcv
   lineEnds    {COPY  lineEnds    } DRC CHECK MAP lineEnds      910 1
   spaceEnds   {COPY  spaceEnds   } DRC CHECK MAP spaceEnds     910 2
   epeLEAll    {COPY  epeLEAll    } DRC CHECK MAP epeLEAll      920
   epeSEAll    {COPY  epeSEAll    } DRC CHECK MAP epeSEAll      920 2
   epeRestAll  {COPY  epeRestAll  } DRC CHECK MAP epeRestAll    920 3
   epeLE       {COPY  epeLE       } DRC CHECK MAP epeLE         921 1
   epeSE       {COPY  epeSE       } DRC CHECK MAP epeSE         921 2
   epeRest     {COPY  epeRest     } DRC CHECK MAP epeRest       921 3
#ENDIF

// --- RDB output
// ------------------------
// Send the ascii format outputs of all EPE checks to an RVE database.
   rdb_epeLE     { DFM RDB epeLE     "$RDB_OUT" MAXIMUM ALL ALL \
                   CELLS NOEMPTY CHECKNAME epeLE       }
   rdb_epeSE     { DFM RDB epeSE     "$RDB_OUT" MAXIMUM ALL ALL \
                   CELLS NOEMPTY CHECKNAME epeSE       }
   rdb_epeRest   { DFM RDB epeRest   "$RDB_OUT" MAXIMUM ALL ALL \
                   CELLS NOEMPTY CHECKNAME epeRest     }
```

```
// endif EPECHECKS
#ENDIF

// --- MPC Verification LITHO FILE
LITHO FILE mpcv [/*
   svrf_var_import DV0 DV1 DV0p5 DV1p5 DV0m5 DV1m5 leWmax leLmin

# --- Larger tile size can have positive effect on runtime but needs
#     larger main memory on remote machines
   tilemicrons 75

# --- We take the model path and file names from the environment
   modelpath                  $env(MODELS_DIR)
   etch_model_load   etch0    $env(ETCH_MOD)
   ebeam_model_load  ebeam0   $env(EBEAM_MOD)

# --- Sequence needs to follow the sequence in LITHO MPCVERIFY command
   layer Target
   layer MpcDL0
   layer MpcDL1

# --- Generate filters for line Ends, spaceEnds, corners, and jogs
   setlayer lineEnds    = identify_edge Target length <= $leWmax \
                          length1 >= $leLmin length2 >= $leLmin \
                          corner1 convex corner2 convex extend -0.035 \
                          expand 0.004
   setlayer spaceEnds   = identify_edge Target length <= $leWmax \
                          length1 >= $leLmin length2 >= $leLmin \
                          corner1 concave corner2 concave extend -0.035 \
                          expand 0.004
   setlayer corners     = filter_generate Target expand 0.005 \
                          convex 0.035 concave 0.035
   setlayer jogs        = filter_generate Target expand 0.005 \
                          jog 0.030 0.020
   setlayer allFilters  = or lineEnds spaceEnds corners jogs

# --- Simulate contour
   setlayer sim         = ebeam_simulate MpcDL0 ebeam_model ebeam0 \
                          dose  $DV0 MpcDL1 ebeam_model ebeam0 \
                          dose $DV1 etch_model etch0
# --- Simulate contour at +5% dose
   setlayer sim_p5      = ebeam_simulate MpcDL0 ebeam_model ebeam0 \
                          dose $DV0p5 MpcDL1 ebeam_model ebeam0 \
                          dose $DV1p5 etch_model etch0
# --- Simulate contour at -5% dose
   setlayer sim_m5      = ebeam_simulate MpcDL0 ebeam_model ebeam0 \
                          dose $DV0m5 MpcDL1 ebeam_model ebeam0 \
                          dose $DV1m5 etch_model etch0
# --- Measure EPE at line ends
   setlayer epeLE       = measure_epe sim Target inside lineEnds \
                          epe_spacing 0.005 function max \
                          min_featsize 0.030 max_edgelen 0.35 \
                          fragment only not > -0.0002 < 0.0002\
                          output_expanded_edges 0.001 trim_ends 0.002\
                          property { max
                          } classify {
                              context Target
                              halo 0.05
```

```
                              anchor Target
                              score bin_size 0.001
                              worst 50 exact}

# --- Measure EPE at space ends
   setlayer epeSE     = measure_epe sim Target inside spaceEnds \
                        epe_spacing 0.005 function  min \
                        min_featsize 0.030 max_edgelen 0.35 \
                        fragment only not > -0.0002 < 0.0002 \
                        output_expanded_edges 0.001 trim_ends 0.002 \
                        property { min
                        }  classify {
                             context Target
                             halo 0.05
                             anchor Target
                             score bin_size 0.001
                             worst 50 exact}

# --- Measure EPE outside all filters
   setlayer epeRest    = measure_epe sim Target outside allFilters \
                        epe_spacing 0.005 function  average \
                        min_featsize 0.030 max_edgelen 0.35 \
                        fragment only not > -0.0002 < 0.0002  \
                        output_expanded_edges 0.001 trim_ends 0.002 \
                        property { average
                        }  classify {
                             context Target
                             halo 0.05
                             anchor Target
                             score bin_size 0.002
                             worst 50 exact}
*/]
```

Full Contour-Based Verification With Eight Dose Levels

Using additional dose assignments in Calibre nmMPC provides more freedom for better CD and edge slope (contrast) control.

The number of dose levels depends on the e-beam writer. Regular VSB e-beam tool writers only allow one dose level (nominal dose 1.0). Some advanced VSB e-beam writers allow eight or more dose levels. In Calibre nmMPC, rule-based dose assignment is applied. If we apply eight dose levels in the Calibre nmMPC correction recipe, those 8 dose layers must be read into Calibre MPCverify and the appropriate dose level must be assigned to each layer.

In this example, eight dose levels are applied to eight different layers, and the dose ranges from 1.0 to 1.7.

```
//////////////// RUN CONTROL /////////////////////////
#DEFINE EPECHECKS
// run the actual EPE checks
#UNDEFINE FILTER_SMALL
// filters out all sub-resolution MPC test structures (useful when running
// on test chips)
#DEFINE SIMOUT
// activates output of simulation contours (expensive for large clips)
#UNDEFINE PWOUT
//  activates simulation of process window (+/- 5% dose)
#DEFINE DEBUG
// activate additional detailed output
/////////////////////////////////////////////////////

LAYOUT SYSTEM OASIS

LAYER IGNORE 999
LAYER MAP >=0 DATATYPE >= 0 999

// --- Read in mask data
LAYOUT PATH    "$VERIF_IN"
LAYOUT PRIMARY "*"

// --- Activate ULTRAflex mode
LAYOUT ULTRA FLEX YES 512
RET GLOBAL LITHOFLAT YES

// --- MPC output should always be at 10k PRECISION
PRECISION 10000

// --- Take pre-bais value from .csh to be consistent with nmMPC
//     correction recipe
VARIABLE PRE_BIAS ENVIRONMENT

// --- Output Oasis file
DRC RESULTS DATABASE "$VERIF_OUT" OASIS PSEUDO CBLOCK
DRC MAXIMUM RESULTS ALL   // write all results to the output

// --- Read in the MPC'ed dose layers
LAYER MAP  0  DATATYPE 0  1000
LAYER MpcDL0              1000
LAYER MAP  0  DATATYPE 1  1001
LAYER MpcDL1              1001
LAYER MAP  0  DATATYPE 2  1002
LAYER MpcDL2              1002
LAYER MAP  0  DATATYPE 3  1003
LAYER MpcDL3              1003
LAYER MAP  0  DATATYPE 4  1004
LAYER MpcDL4              1004
LAYER MAP  0  DATATYPE 5  1005
LAYER MpcDL5              1005
LAYER MAP  0  DATATYPE 6  1006
LAYER MpcDL6              1006
LAYER MAP  0  DATATYPE 7  1007
LAYER MpcDL7              1007
LAYER MAP 99  DATATYPE 0  1099
LAYER MpcTarget           1099
```

```
// --- Define dose value for each dose layer
VARIABLE DV0   1.00    // nominal dose
VARIABLE DV1   1.10
VARIABLE DV2   1.20
VARIABLE DV3   1.30
VARIABLE DV4   1.40
VARIABLE DV5   1.50
VARIABLE DV6   1.60
VARIABLE DV7   1.70
// --- Define dose values at +5% dose
VARIABLE DV0p5 1.05    // nominal dose +5%
VARIABLE DV1p5 1.155
VARIABLE DV2p5 1.26
VARIABLE DV3p5 1.365
VARIABLE DV4p5 1.47
VARIABLE DV5p5 1.575
VARIABLE DV6p5 1.68
VARIABLE DV7p5 1.785
// --- Define dose values at -5% dose
VARIABLE DV0m5 0.95    // nominal dose -5%
VARIABLE DV1m5 1.045
VARIABLE DV2m5 1.14
VARIABLE DV3m5 1.235
VARIABLE DV4m5 1.33
VARIABLE DV5m5 1.425
VARIABLE DV6m5 1.52
VARIABLE DV7m5 1.615

// --- Filter line ends
VARIABLE leWmax  0.300  // maximum width of LE
VARIABLE leLmin  0.150  // minimum length to qualify as LE

#IFDEF $APPLY_PRE_BIAS
   // --- Target shapes received pre-bias
   TargetBiasedTmp  = SIZE MpcTarget BY PRE_BIAS
#ELSE
   TargetBiasedTmp  = COPY MpcTarget
#ENDIF

// --- Remove extremely small lines and spaces we might have on a test
//     chip or because we clipped the design

TargetBiasedMrc1 = SIZE TargetBiasedTmp  BY 0.015 UNDEROVER
TargetBiased     = SIZE TargetBiasedMrc1 BY 0.015 OVERUNDER

#IFDEF FILTER_SMALL
// --- Filter small features which are beyond process limit (only to be
//     used with test chips)
   VARIABLE lineFlt  0.030
   VARIABLE spaceFlt 0.030
   VARIABLE contFlt  0.055

   smallFeatures1 = TargetBiased WITH WIDTH <lineFlt
   smallContacts  = RECTANGLE TargetBiased <=contFlt BY <=3*contFlt
   smallSpace     = EXTERNAL TargetBiased <=spaceFlt OPPOSITE REGION
   FltAll         = size (OR smallFeatures1  smallContacts smallSpace) \
                    by 0.5
#IFDEF DEBUG
```

```
   FltAll      {COPY  FltAll      } DRC CHECK MAP FltAll        930 1
#ENDIF
#ENDIF

#IFDEF SIMOUT
   sim        = LITHO MPCVERIFY TargetBiased  MpcDL0 MpcDL1 MpcDL2 \
                MpcDL3 MpcDL4 MpcDL5 MpcDL6 MpcDL7 MAP sim    FILE mpcv
   sim          {COPY sim           } DRC CHECK MAP sim          900 0
   TargetBiased {COPY  TargetBiased } DRC CHECK MAP TargetBiased 900 99
#IFDEF DEBUG
   MpcDL0 {COPY  MpcDL0 } DRC CHECK MAP MpcDL0 1000 0
   MpcDL1 {COPY  MpcDL1 } DRC CHECK MAP MpcDL1 1000 1
   MpcDL2 {COPY  MpcDL2 } DRC CHECK MAP MpcDL2 1000 2
   MpcDL3 {COPY  MpcDL3 } DRC CHECK MAP MpcDL3 1000 3
   MpcDL4 {COPY  MpcDL4 } DRC CHECK MAP MpcDL4 1000 4
   MpcDL5 {COPY  MpcDL5 } DRC CHECK MAP MpcDL5 1000 5
   MpcDL6 {COPY  MpcDL6 } DRC CHECK MAP MpcDL6 1000 6
   MpcDL7 {COPY  MpcDL7 } DRC CHECK MAP MpcDL7 1000 7
#ENDIF
#ENDIF

#IFDEF PWOUT
   sim_p5  = LITHO MPCVERIFY TargetBiased  MpcDL0 MpcDL1 MpcDL2 \
             MpcDL3 MpcDL4 MpcDL5 MpcDL6 MpcDL7 MAP sim_p5    FILE mpcv
   sim_p     {COPY sim_p5        } DRC CHECK MAP sim_p5     900 1
   sim_m5  = LITHO MPCVERIFY TargetBiased  MpcDL0 MpcDL1 MpcDL2 \
             MpcDL3 MpcDL4 MpcDL5 MpcDL6 MpcDL7 MAP sim_m5    FILE mpcv
   sim_m5    {COPY sim_m5        } DRC CHECK MAP sim_m5    900 2
#ENDIF

#IFDEF EPECHECKS
// --- EPE checks for Line Ends, Space Ends, and all others
   epeLEAll     = LITHO MPCVERIFY TargetBiased MpcDL0 MpcDL1 MpcDL2 \
                  MpcDL3 MpcDL4 MpcDL5 MpcDL6 MpcDL7 MAP epeLE   FILE mpcv
   epeSEAll     = LITHO MPCVERIFY TargetBiased MpcDL0 MpcDL1 MpcDL2 \
                  MpcDL3 MpcDL4 MpcDL5 MpcDL6 MpcDL7 MAP epeSE   FILE mpcv
   epeRestAll   = LITHO MPCVERIFY TargetBiased MpcDL0 MpcDL1 MpcDL2 \
                  MpcDL3 MpcDL4 MpcDL5 MpcDL6 MpcDL7 MAP epeRest FILE mpcv
   epeLE        = NOT  epeLEAll   FltAll
   epeSE        = NOT  epeSEAll   FltAll
   epeRest      = NOT  epeRestAll FltAll

#IFDEF DEBUG_MORE
// --- Output
   lineEnds    = LITHO MPCVERIFY TargetBiased MpcDL0 MpcDL1 MpcDL2 \
      MpcDL3 MpcDL4 MpcDL5 MpcDL6 MpcDL7 MAP lineEnds   FILE mpcv
   spaceEnds   = LITHO MPCVERIFY TargetBiased MpcDL0 MpcDL1 MpcDL2 \
      MpcDL3 MpcDL4 MpcDL5 MpcDL6 MpcDL7 MAP spaceEnds  FILE mpcv
   lineEnds     {COPY  lineEnds     } DRC CHECK MAP lineEnds      910 1
   spaceEnds    {COPY  spaceEnds    } DRC CHECK MAP spaceEnds     910 2
   epeLEAll     {COPY  epeLEAll     } DRC CHECK MAP epeLEAll      920 1
   epeSEAll     {COPY  epeSEAll     } DRC CHECK MAP epeSEAll      920 2
   epeRestAll   {COPY  epeRestAll   } DRC CHECK MAP epeRestAll    920 3
   epeLE        {COPY  epeLE        } DRC CHECK MAP epeLE         921 1
   epeSE        {COPY  epeSE        } DRC CHECK MAP epeSE         921 2
   epeRest      {COPY  epeRest      } DRC CHECK MAP epeRest       921 3
#ENDIF
```

```
// --- RDB output
// ------------------------
// Send the ascii format outputs to an RVE database.
   rdb_epeLE    { DFM RDB epeLE     "$RDB_OUT" MAXIMUM ALL ALL \
                CELLS NOEMPTY CHECKNAME epeLE       }
   rdb_epeSE    { DFM RDB epeSE     "$RDB_OUT" MAXIMUM ALL ALL \
                CELLS NOEMPTY CHECKNAME epeSE       }
   rdb_epeRest  { DFM RDB epeRest   "$RDB_OUT" MAXIMUM ALL ALL \
                CELLS NOEMPTY CHECKNAME epeRest     }
// endif EPECHECKS
#ENDIF

// --- MPC Verification LITHO FILE
LITHO FILE mpcv [/*
   svrf_var_import DV0 DV1 DV2 DV3 DV4 DV5 DV6 DV7 DV0p5 DV1p5 DV2p5 \
                   DV3p5 DV4p5 DV5p5 DV6p5 DV7p5 DV0m5 DV1m5 DV2m5 \
                   DV3m5 DV4m5 DV5m5 DV6m5 DV7m5 leWmax leLmin

# --- Larger tile size can have positive effect on runtime but needs #
#     larger main memory on remote machines
   tilemicrons 75

# --- We take the model path and file names from the environment
   modelpath                     $env(MODELS_DIR)
   etch_model_load    etch0      $env(ETCH_MOD)
   ebeam_model_load   ebeam0     $env(EBEAM_MOD)

# --- Sequence needs to follow the sequence in LITHO MPCVERIFY command
   layer Target
   layer MpcDL0
   layer MpcDL1
   layer MpcDL2
   layer MpcDL3
   layer MpcDL4
   layer MpcDL5
   layer MpcDL6
   layer MpcDL7


# --- Generate filters for line Ends, spaceEnds, corners, and jogs
   setlayer lineEnds     = identify_edge Target length <= $leWmax \
                           length1 >= $leLmin length2 >= $leLmin \
                           corner1 convex corner2 convex extend -0.035 \
                           expand 0.004
   setlayer spaceEnds    = identify_edge Target length <= $leWmax \
                           length1 >= $leLmin length2 >= $leLmin \
                           corner1 concave corner2 concave extend -0.035 \
                           expand 0.004
   setlayer corners      = filter_generate Target expand 0.005 \
                           convex 0.035 concave 0.035
   setlayer jogs         = filter_generate Target expand 0.005 \
                           jog 0.030 0.020
   setlayer allFilters   = or lineEnds spaceEnds corners jogs

# --- Simulate nominal contour
   setlayer sim          = ebeam_simulate \
                           MpcDL0 ebeam_model ebeam0 dose $DV0 \
                           MpcDL1 ebeam_model ebeam0 dose $DV1 \
```

```
                        MpcDL2 ebeam_model ebeam0 dose $DV2 \
                        MpcDL3 ebeam_model ebeam0 dose $DV3 \
                        MpcDL4 ebeam_model ebeam0 dose $DV4 \
                        MpcDL5 ebeam_model ebeam0 dose $DV5 \
                        MpcDL6 ebeam_model ebeam0 dose $DV6 \
                        MpcDL7 ebeam_model ebeam0 dose $DV7 \
                        etch_model etch0
# --- Simulate contour at +5% dose
   setlayer sim_p5     = ebeam_simulate \
                        MpcDL0 ebeam_model ebeam0 dose $DV0p5 \
                        MpcDL1 ebeam_model ebeam0 dose $DV1p5 \
                        MpcDL2 ebeam_model ebeam0 dose $DV2p5 \
                        MpcDL3 ebeam_model ebeam0 dose $DV3p5 \
                        MpcDL4 ebeam_model ebeam0 dose $DV4p5 \
                        MpcDL5 ebeam_model ebeam0 dose $DV5p5 \
                        MpcDL6 ebeam_model ebeam0 dose $DV6p5 \
                        MpcDL7 ebeam_model ebeam0 dose $DV7p5 \
                        etch_model etch0
# --- Simulate contour at -5% dose
   setlayer sim_m5     = ebeam_simulate \
                        MpcDL0 ebeam_model ebeam0 dose $DV0m5 \
                        MpcDL1 ebeam_model ebeam0 dose $DV1m5 \
                        MpcDL2 ebeam_model ebeam0 dose $DV2m5 \
                        MpcDL3 ebeam_model ebeam0 dose $DV3m5 \
                        MpcDL4 ebeam_model ebeam0 dose $DV4m5 \
                        MpcDL5 ebeam_model ebeam0 dose $DV5m5 \
                        MpcDL6 ebeam_model ebeam0 dose $DV6m5 \
                        MpcDL7 ebeam_model ebeam0 dose $DV7m5 \
                        etch_model etch0

# --- Measure EPE at line ends
   setlayer epeLE      = measure_epe sim Target inside lineEnds \
                        epe_spacing 0.005 function  max \
                        min_featsize 0.030 max_edgelen 0.35 \
                        fragment only not > -0.001 < 0.001 \
                        output_expanded_edges 0.001 trim_ends 0.002\
                        property { max
                        }  classify {
                            context Target
                            halo 0.05
                            anchor Target
                            score bin_size 0.001 property max
                        }

# --- Measure EPE at space ends
   setlayer epeSE      = measure_epe sim Target inside \
                        spaceEnds epe_spacing 0.005 function  min \
                        min_featsize 0.030 max_edgelen 0.35 \
                        fragment only not > -0.001 < 0.001 \
                        output_expanded_edges 0.001 trim_ends 0.002 \
                        property { min
                        }  classify {
                            context Target
                            halo 0.05
                            anchor Target
                            score bin_size 0.001 property min
                            worst 500 exact}
```

```
# --- Measure EPE outside all filters
   setlayer epeRest     = measure_epe sim Target outside \
                          allFilters epe_spacing 0.005 function  average \
                          min_featsize 0.030 max_edgelen 0.35 \
                          fragment only not > -0.001 < 0.001  \
                          output_expanded_edges 0.001 trim_ends 0.002 \
                          property { average
                          } classify {
                              context Target
                              halo 0.05
                              nchor Target
                              score bin_size 0.002 property average
                              worst 500 exact}
*/]
```

Chapter 4
Calibre MPCverify Function Reference

Calibre MPCverify is a command language that has an extensive set of setup commands and operations.

Constraints

Certain layer operations are measurement-oriented and therefore carry constraints. Constraints are intervals of non-negative real numbers; input data that falls within the specified constraint of an operation is generally output data.

The syntax for constraints uses one of the six keywords (operators) shown in Table 4-1.

Table 4-1. Constraints

Operators	Constraint Notation	Alternate Constraint Notation	Mathematical Notation
<	< a	NA	x < a
>	> a	NA	x > a

Table 4-1. Constraints (cont.)

Operators	Constraint Notation	Alternate Constraint Notation	Mathematical Notation
<=	<= a	NA	$x \leq a$
>=	>= a	NA	$x \geq a$
==	== a	NA	$x = a$
!=	!= a	NA	$x \neq a$
> and <	> a < b	< b > a	$a < x < b$
>= and <	>= a < b	< b >= a	$a < x \leq b$
> and <=	> a <= b	<= b > a	$a < x \leq b$
>= and <=	>= a <= b	<= b >= a	$a \leq x \leq b$

Notice that the syntax for the last four forms is simply a combination of that for the first four forms. In most cases, a >= 0 and a < b. The constraint "< 0" is not allowed because strictly negative constraint values are not possible in SVRF operations. The constraint "<= 0" is permitted.

Not all operations accommodate all types of constraints or all values of the numbers a and b. For example, setting a=0 in the following constraint is not valid for use with the With Width operation, because a polygon with zero width is not possible for this operation.

```
WITH WIDTH layer1 >= a < b // not valid
```

Restrictions are discussed, when applicable, with the description of the operation.

As an example, the constraint < 4 denotes all non-negative numbers less than 4, and the constraint >= 5 < 7 denotes all numbers greater than or equal to 5 and less than 7.

Some Calibre MPCverify operations have a ratio mode that can be specified; when constraints are specified with a ratio mode, the second value is used as a ratio of the first, using 1.0 as "equal". For example, the constraint "< 1.5" is read as "50 percent greater than".

Property, Classification, Limit, and Histogram Blocks

Several Calibre MPCverify commands can write out properties to an RDB database. Results of such Calibre MPCverify checks that have this additional information are viewable in Calibre RVE.

Note

Properties created using Calibre MPCverify are not the same as GDS or OASIS properties that are stored in the design.

Calibre MPCverify property output must also be explicitly coded into your SVRF rule file. See the section "Writing Properties to an RDB File" on page 50 for more information.

Calibre MPCverify properties are floating point numbers representing additional information attached to error polygons. For example, suppose a measure_cd command returns 127 result polygons. If the measure_cd command was specified with the max, min, and average properties, each returned polygon would have three properties attached to it with the associated values.

Only properties that are explicitly requested are attached.

The full syntax notation for a property block is as follows:

```
setlayer derived = function_name .... [property '{' propfunction1
[propfunction2]
[propfunction3]
... # more properties as needed
'}'] # closing brace
```

Properties require a specially formatted syntax block with the following rules:

1. The property keyword must be on the same line as the command that calls it and be followed by a left brace ({). For example:

   ```
   setlayer derived = measure_cd .... property {
   ```

2. After the left brace, specify one or more property function calls. The first call can occupy the same line as the property keyword, but each additional optional property function must be on a new line. Use a right brace (}) to close the list.

   Correct:

   ```
   setlayer derived = measure_cd .... property { max
   min
   average
   }
   ```

Incorrect:

```
setlayer derived = measure_cd .... property { max min
}
```


Adding Limits

Calibre MPCverify commands can return large amounts of results, especially for bigger, more complex chip designs. In such cases, you can optionally add a limit restriction to return only the worst subset of error polygons to the SVRF rule file. You use limits as a safety net to avoid excessive error generation times due to variations in the layout. A limit is based off of one property function, a number of errors to return, and a selection criterion (smallest or largest value for the property).

Note

Error counts before limiting are available in the log file for reference; a text search on the status line LIMITING FOR LAYER will show how many errors were filtered out for each setlayer statement that includes a limit block.

The full functional syntax notation for a property block with included limit block is as follows:

```
setlayer derived = function_name .... [property { propfunction1
[propfunction2]
[propfunction3]
} [limit {
[score [property propfunctionx] [smallest|largest]]
[worst value [keep bin | exact]]
}]]
```

Limit block syntax uses the following rules:

1. Must be associated with a property keyword.
2. Must start the limit block on the same line as the end of the property keyword.
3. On a new line, limit the amount of error bins returned with the worst keyword and a closing brace (a bin contains errors with the same property value):

```
setlayer derived = measure_cd .... property { max
min
average
} limit {
worst 25
}
```

4. The optional keyword score can be specified with either ‘smallest’ or ‘largest’ to indicate which end of the data set is considered to be the worst.

```
...
} limit {
score largest
worst 25
}
```

Tip

The values “largest” and “smallest” must be selected to be relevant to your expected data and represent a non-absolute value selection criterion. For example, in order to find the worst line end pullback values (a negative EPE value), you would choose “smallest”. For line end extension values (a positive EPE value), you would choose “largest”.

When measuring the magnitude of the EPE (absolute value of over/under the target), ‘largest’ is the recommended selection.

5. If multiple types of property keyword have been specified, the score keyword can be modified with the property *propertyname* arguments between “score” and “largest/ smallest”:

```
...
} limit {
score property average largest
worst 25
}
```

   - Do not confuse the score property keyword inside the limit block with the initial property block keyword declaration.
   - The score property keyword must specify one of the property arguments previously given in the property block. For example, in the code above, score property can only use a min, max, or average argument.
   - If a score property is not specified, the first property block keyword is used (max in the example code above).

6. By default, the number of errors output is limited to the value specified in the worst argument, but also includes all the errors in the last score bin. This means that the number of errors returned may greatly exceed the value for “worst” if the last score bin contains a large number of errors.

Note that the modifier “exact” can be added to return exactly the number of errors specified in worst (the default behavior can be explicitly specified with the keep_bin modifier):

```
...
} limit {
score property average largest
worst 25 exact
}
```

Errors are sorted in priority order by the ‘worst’ value, then by smallest cell_id, then by smallest x coordinate, then separately by smallest y coordinate.

Writing Properties to an RDB File

A standard call to Calibre MPCverify returns a derived layer. However, DRC CHECK MAP calls do not write the Calibre MPCverify property information returned in the derived layer to the output design file. Instead, you must take the derived layer and use it as the argument to a DFM RDB call:

```
rulename { DFM RDB mpcv_derived_layer "filename" NOEMPTY }
```

where:

- ***rulename***

  Is a new rule identifier that will appear in the error browser. You should choose a name that helps identify the properties you have isolated.

- ***mpcv_derived_layer***

  Is the name of a derived layer resulting from an earlier Calibre MPCverify call.

- ***filename***

  Is the name for the RDB file to be created. This file is stored in the current working directory by default.

- NOEMPTY is an option used by the DFM RDB command. This command is documented in the *Standard Verification Rule Format (SVRF) Manual*.

Examples

Example 1 (No Limits)

If your SVRF rule file contains the following Calibre MPCverify call:

```
cdr = LITHO MPCVERIFY FILE "setup/check.in" popc target MAP cdr
cdr { copy cdr } DRC CHECK MAP cdr 111 0
```

and your Calibre MPCverify setup file contains the example code:

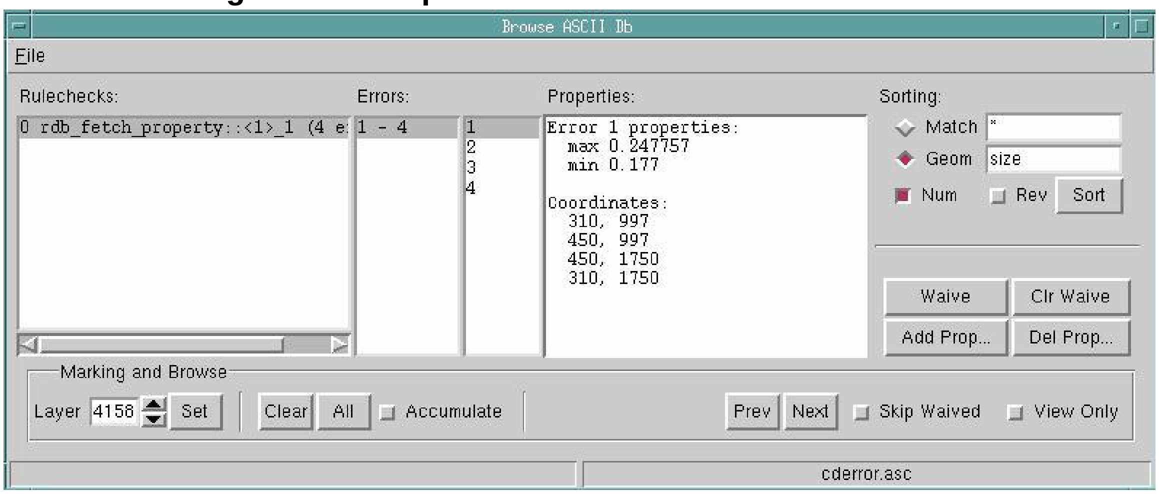
```
setlayer cdr     = measure_cd contour target internal cd_min 0.11 \
                   cd_max 0.16 cd_min_length 0.01 max_search 0.06 \
                   tol not < 1.1 ratio \
                   property {max
                             min}
```

Assuming that the derived layer “cdr” had properties returned, the following additional code creates the corresponding RDB database file with the properties information. (You should write the results only to the RDB file and not the GDS file.)

```
cdr_max { DFM RDB cdr “cdr_max.asc” MAXIMUM ALL ALL CELLS NOEMPTY }
```

Figure 4-1 shows the results (4) from a simple design file in the Browse ASCII DB tool (access this tool from Verification Center using the **Tools** > **RDB Browser** menu item).

Figure 4-1. Properties in the Browse Ascii DB Tool



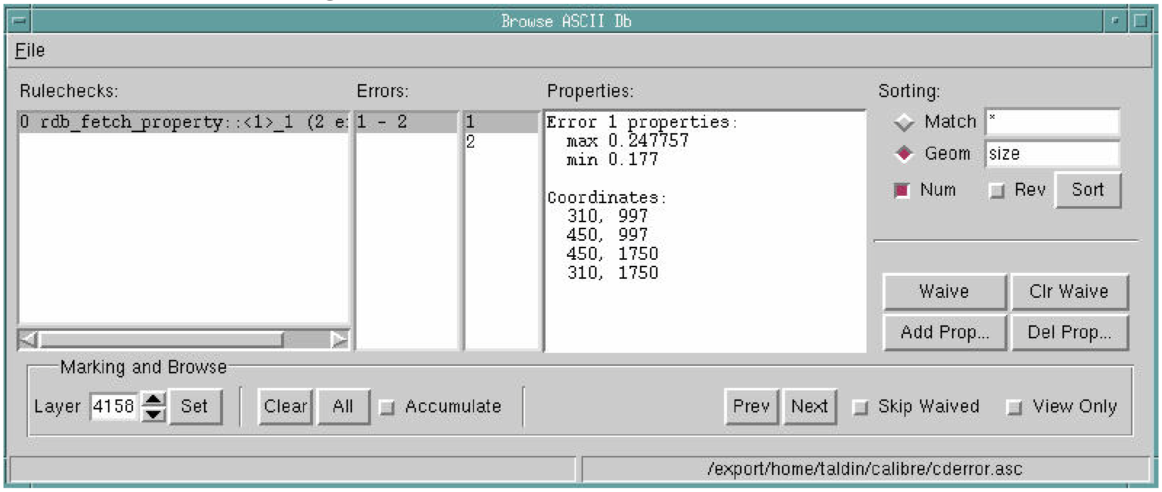
Example 2 (Limits)

Using the same example SVRF rule file as Example 1, but adding limits to the Calibre MPCverify rule check as follows:

```
setlayer cdr = measure_cd contour target internal cd_min 0.11 cd_max 0.16\
               cd_min_length 0.01 max_search 0.06 tol not < 1.1 ratio \
               property {max
                         min
               } limit {
                  score property min smallest
                  worst 2
               }
```

This results in only two error bins being returned in the database file (Figure 4-2):

Figure 4-2. RDB Browser, Limit Result



Note

This is a simple example. All errors in each bin have the same property value; in a larger sampling, there might be 5000 errors in the first bin and 6000 in the second bin, for example.

Classification Blocks

Rule File Construct

A Calibre MPCverify classification block is an extension to some of the Calibre MPCverify commands, allowing you to classify the filtered results of the command. It can be used in conjunction with a property function block, but is mutually exclusive with histogram and limit blocks.

Usage

setlayer *command*
...
[property ‘**{**’ *property_block*
‘**}**’] **classify** [*classify_name*]
‘**{**’
context *layer1* [*layer2* ... *layerN*]
halo *halo_microns* [around {center | extent}]
[coarse_match *cm_microns*]
[unique_unclassified_ids {off | on}]
[maxsize *ms_microns*]
[maximum_error_number *max_errors*]
[{suppress | keep | keep_no_context} duplicates]
[anchor *alayer* [anchor_max_snap *ams_microns*] [anchor_halo *ah_microns*]]
[approximate_context *pos_width_tol alayer1* [*alayer2*]...]
[approximate_stopping [min_compound_duplicate_rate *mcd_rate*]
[iterations *iter_constraint*]]
[[score [bin_size *score_delta*] [property *propname*] [smallest | largest]
[worst [unique | total] *count* [keep_bin | exact]
[{duplicates | per_class} *dup_count*]]]
[pm_classify max_search *value* max_length *value* [orig_runlength]]
‘**}**’

Description

Classification blocks can be specified for most setlayer checks (refer to the documentation for each setlayer check).

Note

The command “setlayer x = copy y classify {..}” syntax allows you to classify any setlayer in the deck.

A centerpoint-style classification is used for errors; Calibre MPCverify finds the center of the error and uses that as the center for the haloed context. Adding the “around extent” to the **halo** argument uses the halo area around the error marker as the extent.

When the classification process is complete, the result information is attached to output polygons as the properties CLASS_FLAT_COUNT and CLASS_ID.

- For unique errors, CLASS_FLAT_COUNT encodes the flat count of instances of this particular error including the duplicates and the unique itself.
- For duplicate errors, CLASS_FLAT_COUNT is set to 0.
- For unclassified errors, CLASS_FLAT_COUNT is set to the total placement count of the pseudo cell that unclassified errors are in.

If the keep or keep_no_context duplicates option was specified, CLASS_ID contains the IDs of errors.

- Duplicate errors have their CLASS_ID set to the ID of the corresponding unique error.
- Unclassified errors have their CLASS_ID set to 0.

MTFlex and TURBOflex May Return Different Classification Results

If you are working with both Calibre® MTflex™ and Calibre TURBOflex solutions, you may notice slightly different error counts and outputs when run on the same design. These are a result of slight differences between the way MTflex and TURBOflex handle tiling, and have the following known effects:

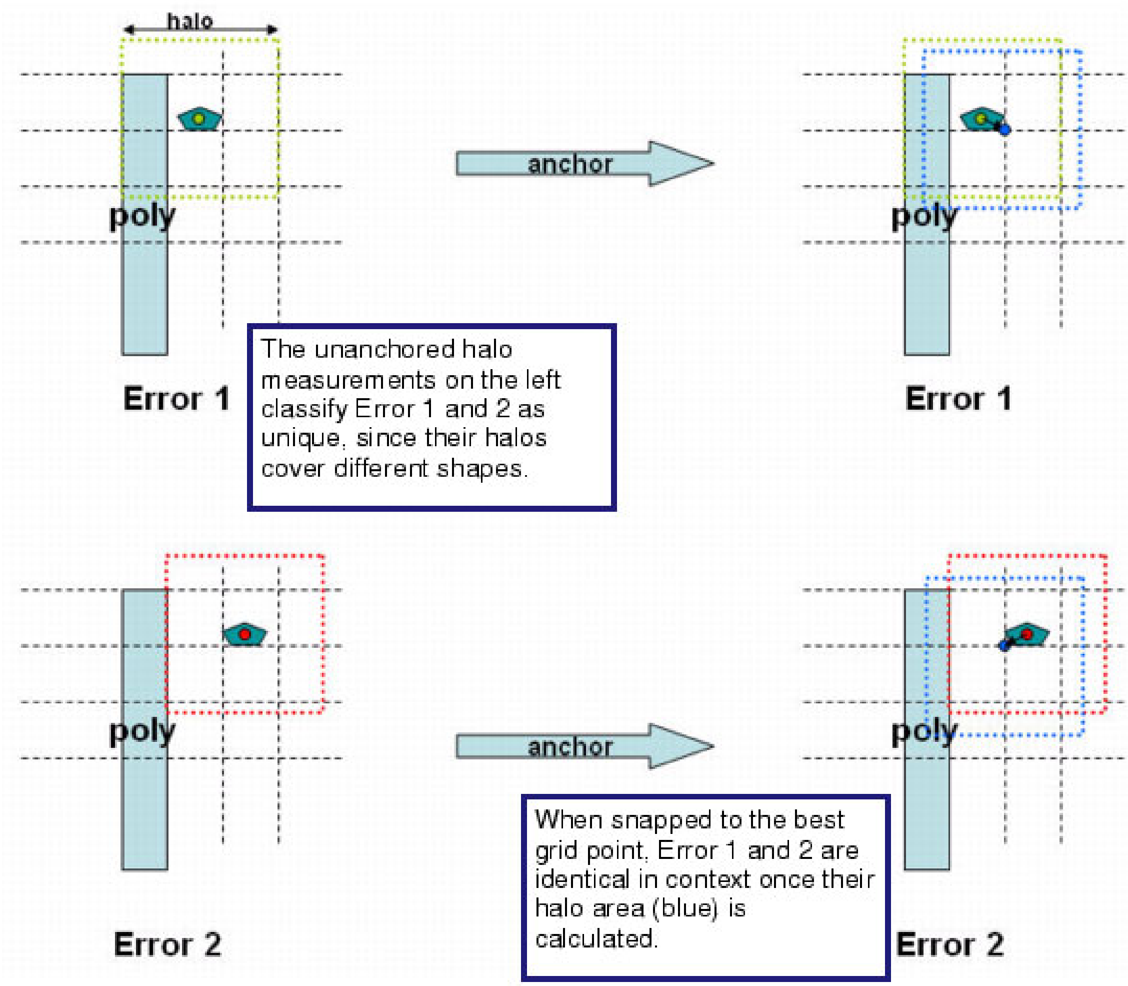
- **Limiting (when using the score keyword)** — The score threshold is updated as each tile is processed. The score threshold (and the screening done based on the threshold) depends which tile finishes first. The tile processing order can vary from run to run in MTFlex. The tile processing order will also change due to TURBOflex operations.
- **Different locations of unique classified returned results** — All unique classified results are found in order of the deepest cell in hierarchy in the lowest X,Y order. TURBOflex alters the cell hierarchy, which can cause duplicate classified shapes to be located in different cells. If the lowest (cell,X,Y) unique shape is moved into a cell higher in the cell hierarchy, a different unique-shape location from another cell will be returned instead. This means that the locations returned by a non-TURBOflex run may be different from the locations returned by a TURBOflex run.
- **Different flat counts of unique classified shapes** — Because TURBOflex alters the cell hierarchy, the cell placement count may be different. A change in the cell placement count changes the cell flat count.
- **Subtle contour differences due to different cell tiling and tile processing order** — MPC results for tiles are dependent on the processing order of the adjacent tiles. The MPC'd tile result is influenced by adjacent MPC'd tile results. Because TURBOflex can change the MPC tile processing order, formerly adjacent tiles can be either pushed up or down the cell hierarchy, thereby changing the contours. In addition, the TURBOflex restructured cells will have different tile boundaries.

Anchoring Mode

Anchoring is a process where each error's center is first snapped to a grid before the uniqueness comparisons are made. The grid is drawn based on the closest polygon vertex in context; the size of the grid is equal to 2*anchor_max_snap option.

Anchoring mode makes classification less sensitive to the location of an error's center point and reduces the number of unique errors (see Figure 4-3). Use the drawn (target) layer as the anchor layer, and the anchor_layer keyword to activate this mode.

Figure 4-3. Anchoring



Usage Notes

- A setlayer operation with a limit or classify block can be used as input to another **setlayer** operation. However, it is pre-limited and pre-classified geometry that is fed into another **setlayer** operation.
- At most 4 billion errors can be handled by the classification code.

- Calibre MPCverify classify produces slightly different results even when there are no unclassified errors in Calibre MPCverify (due to the maxsize limit being reached) and anchoring is not used. The reason is a difference in handling of errors whose extents are odd in 1 or 2 dimensions.
- In a hierarchical Calibre run, a unique error is the one with the smallest ordered triplet of (Calibre-internal cell id (the 'bottom-most' cell), the error's smallest x coordinate, and the error's smallest y coordinate). However, due to hierarchy-injection, cell numbers might not be consistent from run to run. Use the following command:

  ```
  setenv CALIBRE_DISABLE_MT_INJECTION on
  ```

  This command deactivates this source of unique error inconsistency (and gets longer HDB constructor run time).
- If *classify_name* is specified, there must be a classify_options statement in the setup section defined with that name. A classify_options statement also allows you to skip required arguments if they are defined in the classify_options block. Arguments set in an individual classify block overwrite the classify_options block setting.

Arguments

- **classify** [*classify_name*]

  All arguments must be contained within a **classify** or **classify** *classify_name* block, enclosed by braces ({ }).
- **context** ***layer1*** [*layer2* ... *layerN*]

  An argument specifying at least one context layer to use to measure nearby polygons for contextual similarities. Layers must be Calibre MPCverify input layers.

  - The context argument is required for a classify block, unless the classify block is used with a command listed in Table 4-2.
  - For the commands listed in Table 4-2, the context argument is optional.

    The context *layer* values used are taken from specific layer arguments to the command when no context argument is specified in the classification block.

    For example, specifying a classify block without a context argument for the measure_cd command uses the input ***ref_layer*** argument as its **context** ***layer1*** argument.

Table 4-2. Default Context Layers for Classification

Command	Context Layer(s) Used
measure_cd	***ref_layer***
measure_epe	***layer_target***
nilscheck	***layer_target***

Table 4-2. Default Context Layers for Classification (cont.)

Command	Context Layer(s) Used
copy	***layer***

- **halo** ***halo_microns*** [around {center | extent}]

> **Note**
> The halo keyword must appear on a new line in order for it not to be interpreted as a context layer argument.

  A required argument that specifies a halo distance around an error used as the classification region for each error. A square of size 2****halo_microns*** is clipped out around each error center by default.

  Specifying “around extent” sets the halo to instead be the shape of the error polygon, sized up by ***halo_microns***. Error shapes must have their extents matching to be considered duplicates. The “around extent” argument is incompatible with the anchor argument.

- coarse_match *cm_microns*

  An optional argument specifying that if a pattern is larger than or equal to 3 dbu, patterns whose context layers differ by less than *cm_microns* in *layer2* may be classified as duplicates. This is an approximate matching method.

- unique_unclassified_ids {off | on}

  An optional argument that, when set to “on”, specifies that unclassified errors be issued unique CLASS_IDs. When set to “off” (the default setting), this keyword specifies that unclassified errors are assigned as CLASS_ID = 0.

- maxsize *ms_microns*

  An optional argument specifying a maximum error size. Errors with an extent exceeding *ms_microns* in an X or Y direction will be considered unclassified. Default: 0.5 microns.

- maximum_error_number *max_errors*

  An optional argument specifying a limit to the number of total errors recorded. If specified, it *turns off classification* after *max_errors* have been found, and marks errors in the remaining tiles as unclassified. The granularity of this option is per tile; it has no effect for flat runs or runs with a single tile.

Caution

In a Calibre MTFlex run with multiple remotes, the maximum_error_number keyword allows all remotes to finish classifying the current tile before turning off classification for later tiles. The exact subset of errors that are classified when this argument is in use is arbitrary. You should use maximum_error_number *ONLY* as a safety measure against extremely long classification times on unexpectedly large numbers of errors.

Siemens EDA also recommends against the use of a maximum_error_number that is too similar in size to the "worst *count*" argument, as under certain conditions classification can be turned off too early to capture significant errors. The suggested value for maximum_error_number is *at least* ten times relative to worst *count*.

- {<u>suppress</u> | keep | keep_no_context} duplicates

  An optional argument that specifies whether or not duplicates are suppressed. Default: Duplicates are suppressed. If suppress duplicates is specified (the default), a CLASS_ID property is not attached to the output, and the command will output only unique and unclassified errors.

  - keep duplicates — Outputs context contours for both unique errors and duplicates.
  - keep_no_context — Duplicates outputs context contours only for unique errors, not for duplicates.

  Note

  CLASS_ID properties are arbitrary assignment numbers, and may differ between a keep versus keep_no_context duplicates run. This is expected behavior.

- anchor *alayer*

  An optional argument that specifies an anchor layer (see "Anchoring Mode") that will be used for anchoring the extent centers. It must be a Calibre MPCverify input layer; we recommend that you use the target layer. This option is particularly useful with measure_distance. Default: <u>no anchoring</u>. Thois is incompatible with the **halo** around extent argument.

- anchor_max_snap *ams_microns*

  An optional argument that specifies the maximum distance the centerpoint of an error extent can move to snap to a grid induced by the closest (in L_inf norm) vertex on the anchor layer that is within the halo region from the error's centerpoint. Larger values yield fewer unique errors. Default: <u>0.2</u> microns.

- anchor_halo *ah_microns*

  An optional argument used only in anchored mode. When this mode is active, the code looks for anchor vertices around the error center in the anchor_halo region. You use this option to search using a halo distance larger than the standard halo argument. If an anchor point is not found within the halo radius, no anchoring is performed.

Default is to use the value specified for the halo argument.

- approximate_context *pos_width_tol alayer1* [*alayer2*]...

  An optional argument that toggles a two-phase "approximate classification" test when post-MPC layers are used as the classification context. The test checks context shapes on the specified *alayer*s to reduce the number of unique errors. Tolerance is a multiple of dbu, expressed in microns, and roughly corresponds to the maximum distance between contexts classified as duplicates. For example, a +/- 2 dbu requirement translates to a 4 dbu setting for *pos_width_tol*.

  The criteria for an approximate duplicate between two possible contexts C1 and C2 is as follows:

  They have an exact content match on all layers specified with the **context** keyword.

  One of the following two conditions must also hold:

  - Let all approximate contexts with the same exact context as C1 and C2 form an approximate context group AC, consisting of {C1, C2, ...C*n*}.

    Let C_outer = OR of AC, and C_inner = AND of AC.

    {C_outer size by -*pos_width_tol*} NOT C_inner == C_outer NOT {SIZE C_inner by *pos_width_tol*}== empty

    This is the condition set where approximate patterns are within tolerance and are combined into a single duplicate set.

  - Alternatively, if there exists a context C of the same exact context as C1 and C2 such that C_band = (SIZE C by ceil(*pos_width_tol*/2)) NOT (SIZE C by -floor(ceil(*pos_width_tol*/2))) and all edges of C1 and C2 are not outside C_band.

    This is the condition where approximate contexts with the same exact context are divided into multiple duplicate sets. Each duplicate set consists of all approximate contexts fitting into a tolerance band created around one of the approximate contexts.

  In other words, "approximate duplicates" must match exactly on context layers, and also be either a part of the approximate context group whose variation band is smaller than *pos_width_tol*, or within *pos_width_tol* tolerance of another exact match context.

  There are several caveats when using approximate matches:

  - Contexts with skew (not a multiple of 45 degrees) edges are not considered in approximate duplicates calculation.

  - *pos_width_tol* is in microns, and must be small compared to the main context feature size. Otherwise, the sizing of line ends and space ends smaller than *pos_width_tol* may produce unexpected results.

- The approximate_context keyword cannot be used with the “score” clause.
- Using MPC layers as *alayer* arguments can be slower, use more memory, and produce more unique errors compared to exact classification using target layers.

- approximate_stopping [min_compound_duplicate_rate *mcd_rate*] [iterations *iter_constraint*]

  Specifies the stopping criteria for the iterative approximate classification process for a group of contexts on approximate_context *alayers* that correspond to the same context on context layers when criteria 2a is not applicable.

  mcd_rate should be set to a positive number between 0 and 1. Smaller values lead to smaller unique percentage and longer runtime. (Default = 0.02)

  This parameter is used in the following algorithm sequence: Suppose there is a group of S1 approximate contexts corresponding to the same exact context on context layers. The code attempts to find duplicates among this group of S1 contexts iteratively. If the code finds 1 unique and D_1 corresponding duplicates during iteration 1, the duplicate rate for iteration 1 is D_1/S1. Iteration 2 will start off with S2 = (S1 - (1 + D_1)) contexts. By fixing the duplicate rate at *mcd_rate*, the code computes the theoretical starting number of contexts for iteration i to be S_i_mcd_rate if the duplicate rate was mcd_rate at every iteration. If the actual number of contexts at the start of iteration i is larger than the S_i_mcd_rate, we stop looking for duplicates among the current set of approximate contexts.

  Example: Suppose S1 = 100 and *mcd_rate* is 0.02. Then:

  - S_2_mcd_rate = 100 - (1 + 0.02*100) = 97
  - S_3_mcd_rate = 97 - (1 + 0.02* 97) = 94.06
  - S_4_mcd_rate = 94.06 - (1 + 0.02* 94.06) = 91.18

  Each iteration is guaranteed to find one unique and 0 or more duplicates. The code does not attempt iteration 2 if iteration 1 finds less than 1 unique and 2 corresponding duplicates, leaving more than 97 approximate contours for next iteration.

  The code will not attempt iteration 3 if after iteration 2 there are more than 94.06 approximate contexts left to process (meaning iteration 1 and 2 managed to find less than 2 unique errors and 4 duplicates collectively).

  The code will not attempt iteration 4 if after iteration 3 there are more than 91.18 approximate contexts left to process (meaning iteration 1, 2 and 3 managed to find less than 3 unique errors and 6 duplicates collectively).

  The iterations argument should be a valid, integer-valued constraint. Larger maximum values lead to a smaller unique percentage and longer runtime. (Default = iterations >= 3; at least 3 iterations are performed for each approximate context group.)

  Both iterations and min_compound_duplicate_rate can be specified. At least one must be specified. If a lower bound is specified by the iterations argument, at least that many iterations are performed before a min_compound_duplicate_rate stopping criteria is evaluated.

- score [bin_size *score_delta*][property *propname*] [smallest | largest]

  An optional argument that indicates the start of a score block. Calibre MPCverify considers any difference in property values between two shapes that is less than *score_delta* to be a duplicate if the geometric contexts match.

  - If the bin_size *score_delta* argument is omitted, then classification uses only geometric contexts in classification, but selects a unique error with the worst score from among a set of duplicates. If the "worst" argument (see below) is also specified in this mode, the "worst ... exact" mode is always used. Explicitly specifying "keep_bin" results in a parsing error.

    Conceptually, when score is specified without "bin_size", one large score bin is used, and exact limiting is always performed if "worst" is specified.

  - For performance reasons, "bin_size" is required when duplicates are kept and the "worst" argument is specified. Also note that classification with limiting uses less memory and time when a reasonable value of bin_size is specified, so bin_size specification is recommended.

    Note

    The use of a score block requires a property block defined outside the classify block.

  The following optional sub-keywords can be specified on the same line as the score argument:

  - property *propname*

    If specified, the named property (which must appear in the property list specified in the property block outside of the classify block) is used as the selection criteria. If not specified, the first property in the list is used.

  - smallest | largest

    Sorts the values for the requested property as smallest first or largest first so that the "worst" value returns the top of the list. The default is smallest.

    Note

    "smallest" and "largest", when used, must appear on the same code line as the "score" argument.

- worst [unique | total] *count* [keep_bin | exact] [{duplicates | per_class} *dup_count*]

  An optional argument to restrict the number of errors. "worst" must be the start of a new line, and also requires that the "score" argument appear in the limit block.

  - By default, the error list is based on either the smallest or largest values in the bins with unique errors. "unique" also includes unique, unclassified errors totaling *count* errors plus the last score bin ("keep_bin"), unless the "score" argument is used without bin_size, in which case 'exact' is used as the default instead).

In “total” mode, all unique and duplicate errors are output, limited by *count*.

- In “exact” mode, worst returns the exact number of errors requested, picking from the last bin in priority order: worst score, then by smallest cell_id, then by smallest x coordinate, then separately by smallest y coordinate. This is the recommended mode.

  Duplicate errors are not counted towards the exact criteria.

- The “duplicates” and “per_class” options can only be used when the “unique” limiting mode is explicitly specified. The options are not compatible with the “total” limiting mode and “suppress_duplicates” option.

  - **duplicates** — After unique limiting, the total number of duplicates is limited to *dup_count*. Duplicates are sorted in order, by unique (from worst), then by property, then by cell ID, then by x and y coordinates. The duplicate keyword is not compatible with total limiting mode and suppress duplicates.

  - **per_class** — after unique limiting, duplicates for each class of unique errors are limited to *dup_count*. Duplicates are sorted by property, then by cell ID, then by x and y coordinates.

- pm_classify max_search *value* max_length *value* [orig_runlength]

  An optional argument that instructs Calibre MPCverify to perform a second phase of error classification after regular classification completes. It significantly reduces the final error counts. This argument invokes the Calibre MPCverify Classify Plus feature, which requires a separate license.

  - pm_classify uses the context_layer, halo, {suppress | keep | keep_no_context} duplicates, reflections_rotations_match settings, and all features of the score and worst options from the classify block. All other classify block parameters do not affect pm_classify and are used only by the regular classification step preceding pm_classify.

  - The use of pm_classify is transparent with regard to its results. The only difference between the output layer and properties when using pm_classify is that it might have fewer unique errors.

  - The CLASS_FLAT_COUNT results for the output markers are the final cumulative value after running pm_classify; they include the hierarchical placement count of the cell where the error resides, plus any duplicate reduction by standard classification, plus duplicate reductions by pm_classify. Output with “keep duplicates” includes duplicates found in both steps of classification.

  - The pm_classify command is implemented as a post-processing flow after the main MPCverify run. The post-processing step is reported in the transcript as a LITHO PM_CLASSIFY run. The final classified error layer can be used in the output_window command and also mapped back to SVRF. The error layer can be used as input to subsequent setlayer statements, but the layer sent to those commands is unclassified and unlimited, which is the same behavior as a standard classify block output.

- The pm_classify command is supported only for the Calibre hierarchical engine (calibre -hier).
- Due to licensing, a check that uses pm_classify must have at least one input contour derived from a setlayer image command initiated in the same Calibre MPCverify run.

pm_classify has one known limitation in the current release:

- In some cases, pm_classify can cause the final CLASS_FLAT_COUNT values to be overestimated. This happens if Classify Plus performs hierarchical promotion of a Calibre MPCverify error. In that case, each promoted copy inherits the original Calibre MPCverify CLASS_FLAT_COUNT, and some double counting occurs.
- Histogram output is based on the error results after Calibre MPCverify classify completes, but before pm_classify reduction.

All parameters for pm_classify must be on the same line:

- max_search *value*

  Specifies the maximum distance from the error marker center that will be searched to find target vertices that can consistently anchor the error to the target geometry. The value must be in microns. Setting this parameter excessively large is likely to cause over-aggressive classification. For this reason, max_search must be smaller than 2 * ***halo_microns***. A value of about 1.0 * ***halo_microns*** is recommended.

- max_length *value*

  Specifies the largest permitted size for an input error marker in microns. Markers larger than max_length are split before Classify Plus classification into sizes less than or equal to max_length. Classification is applied to the split pieces. The splitting of large errors is one important reason why Classify Plus is capable of large reductions in unique error counts.

  max_length must be at least ***halo_microns*** and may not exceed 20 * ***halo_microns***. There is a trade-off in setting this parameter:

  - Smaller values may split relatively small errors, and the cut pieces, if not classified away, may cause larger final error counts.
  - Larger values usually produce smaller final error counts and less confusing final markers, but at the risk of discarding important information. The pm_classify algorithm matches geometry only within a “peephole” of diameter 2 * ***halo_microns*** around a point within max_search of the input marker center. Hence, a large max_length could result in errors with important differences outside this peephole being incorrectly considered identical.

Overall, setting max_length in the range 3 to 10 * ***halo_microns*** is recommended.

Tip

To learn about the results of error splitting by max_length, run classification twice, the second time with “keep duplicates”. Then use the with-duplicates result as the context when viewing the unique-only results in Calibre WORKbench.

- orig_runlength

  If this optional keyword is present, pm_classify generates an output property ORIG_RUNLENGTH for all errors. The property is 0.0 for all errors that were not cut up by pm_classify. For errors that were cut by pm_classify, ORIG_RUNLENGTH specifies the run length, in microns, of the original error marker before cutting.

Examples

Example 1

The following examples add a classification block to a measure_cd command.

Without a property block (score bins cannot be used):

```
setlayer cderror = measure_cd contour target internal cd_min 0.11 cd_max
0.16 cd_min_length 0.01 max_search 0.06 tol not < 1.1 ratio classify {
context target
halo 0.1 }
```

With a property block:

```
setlayer cderror = measure_cd contour target internal cd_min 0.11 cd_max
0.16 cd_min_length 0.01 max_search 0.06 tol not < 1.1 ratio property {
min
max
} classify
{ context target
halo 0.1 }
```

Example 2 (Scoring)

With a property block and a scoring bin block as shown in the following code, the value of the first item in the property list (min) is used in addition to the geometric context to decide if two errors match. Each score bin in this example has a size of 0.01 microns and starts from 0. If two errors fall into different score bins, they are considered as non-matching errors even if they are less than 0.01 microns in difference (0.0195 and 0.0201, for example). Setting a very large value (10) will act if no score was specified (since the errors will land in the same bin); setting a very small value (0.001) will make classification highly sensitive to differences.

```
setlayer cderror = measure_cd contour target internal cd_min 0.11 cd_max
0.16 cd_min_length 0.01 max_search 0.06 tol not < 1.1 ratio property {
min
max
} classify { context target
halo 0.1
score bin_size 0.01}
```

Example 3 (Scoring and Worst)

With a property, scoring, and ‘worst *count* keep_bin’ block, the number of errors returned is limited to display approximately the *count* number of errors. This is not an exact number, however, since the number of errors returned is actually determined by the following criteria:

- Entire bins are returned.
- Bins are added together until the total count is equal to or greater than *count*.

  For example, if bin 0-0.01 contains four unique errors and bin 0.01-0.02 contains five unique errors, worst 3 and worst 4 returns all of bin 0-0.01 (four errors), and worst 5 returns both bins (nine errors).
- Duplicate errors are not counted.

  For example, given the following bins:

  a. bin 0.0-0.01 contains one unique and five duplicate errors
  b. bin 0.0-0.01 also contains one large unique error
  c. bin 0.01-0.02 contains one unique error and two duplicate errors
  d. bin 0.01-0.02 also contains one large unique error
  e. bin 0.02-0.03 contains 1 unique and 99 corresponding duplicates
  f. bin 0.02-0.03 also contains 10 large errors

  Setting a worst value of 3 returns the first two bins (a and b: 2 errors and five duplicates in 0.0-0.01 and c and d: 2 errors and two duplicates in 0.01-0.02). If this were a limit block, only the first bin (a and b) would be returned because the duplicates would be counted (totalling seven errors) until 3 or more errors are totaled.

```
setlayer cderror = measure_cd contour target internal cd_min 0.11 \
                   cd_max 0.16 cd_min_length 0.01 max_search 0.06 \
                   tol not < 1.1 ratio property {
                               min
                               max
                   } classify { context target
                                  halo 0.1
                                  score bin_size 0.01 property max
                                  worst 3 keep_bin
                     }
```

Histogram Blocks

Rule File Construct

A histogram block adds the ability to save tabular data from properties to a file. It can be added to any command that also has a properties block. Histogram blocks are mutually exclusive with classify and limit blocks.

Note

The number of data points stored internally is limited to the number of hierarchical shapes processed (no zero-entries are kept internally). The main performance degradation will occur only if the zero_counts argument is set to keep and the number of bins computed from the ***bin_sz*** argument is very large.

Usage

setlayer x = *command*

...

property '**{**'

property_block

'**}**' histogram '**{**'

file *output_file* property *prop_name* bin_size *bin_sz* [zero_counts {suppress | keep}]

[file *output_file2* property *prop_name*2 bin_size *bin_sz2* [zero_counts {suppress | keep}]]

'**}**'

Description

Calibre MPCverify supports histogram output to a file. A histogram output file has two tab separated columns:

- First column: centers of the bins.
- Second column: integer number of flat occurrences of error polygons on the histogram-enabled Calibre MPCverify layer with a property value in that bin.

Use of a histogram block requires a property block in the same setlayer command; only properties previously named in the property block can have histogram files created for them.

Note

A histogram-enabled Calibre MPCverify layer must also be an output layer in the SVRF rule file for the histogram to be generated.

Arguments

- histogram **{**

  Arguments in this section must be inside a histogram block and enclosed by braces ({}).

- **file *output_file***

  A required argument that specifies the filename to save the histogram data to. File name write permissions are checked at compile time and must be unique among all histograms; old histogram files are overwritten.

  The remaining histogram arguments (property, bin_size, and zero_counts) must appear on a single line with the file keyword.

- **property *prop_name***

  Specifies which of the properties in the previously-declared property block to write information for.

- **bin_size *bin_sz***

  A required argument that specifies the bin size. Bin ranges are computed as follows:

  [-0.5****bin_sz***, 0.5****bin_sz***), [0.5****bin_sz***, 1.5****bin_sz***), [1.5****bin_sz***, 2.5****bin_sz***)

  > **Note**
  > The notation [a,b) is intended to be interpreted as a<= *value* < b; for example, [0.5****bin_sz***, 1.5****bin_sz***) represents a range of 0.5 to 1.49999999.

  bin_sz must be larger that 10^-7. The range of *bin_sz* is limited by the equation: abs(max(property-value)/bin_sz) <= INT_MAX.

  Computed bin indices are integer values.

- zero_counts {<u>suppress</u> | keep}

  An optional parameter that specifies the behavior when the polygon count of a bin is zero. If set to keep, output file will have bins with zero polygon counts; suppress (the default) skips the bin.

Examples

Given the following code:

```
setlayer e2 = area_overlay contact poly_contour not > 0.80 max_extent 0.6\
shift .05 output_type worst property {
  min
} histogram {
  file histogram_e2.txt property min bin_size 0.1
}
setlayer e3 = area_compute contact < 1 max_extent 0.6   property {
  area
} histogram {
  file histogram_e3.txt property area bin_size 0.01
}
setlayer e4 = area_compute contact < 1 max_extent 0.6 property {
  area
}
```

The following information is printed to the transcript:

```
HISTOGRAM MEAN 0.236801 STANDARD DEVIATION 0.213629 FOR LAYER e2:
HISTOGRAM COUNT 1, FLAT GEOMETRY COUNT 1388
TIME FOR LAYER aov_e2 HISTOGRAMMING: CPU TIME = 0  REAL TIME = 0
TIMESTAMP 6
...
HISTOGRAM MEAN 0.050607 STANDARD DEVIATION 0.016282 FOR LAYER e3:
HISTOGRAM COUNT 1, FLAT GEOMETRY COUNT 1917
TIME FOR LAYER aov_e3 HISTOGRAMMING: CPU TIME = 0  REAL TIME = 0
TIMESTAMP 6
....
aov_e3 (HIER TYP=1 CFG=1 HGC=0 FGC=0 HEC=0 FEC=0 VHC=F VPC=F)
aov_e4 (HIER-FMF TYP=1 CFG=1 HGC=1354 FGC=1917 HEC=127681 FEC=178196 VHC=F
VPC=F)
```

The mean and standard deviation is reported for all properties in the property block of the histogram layer. The algorithm for computing standard deviation is ‘biased’ in that the (sum-of-squares minus square-of-sum) term is divided by n, where n is the number of data samples. The calculation of the standard deviation and mean is performed incrementally (adding a single data point at a time) to reduce loss of precision errors that occur when computing the difference of large-almost-equal values.

Mean/standard deviation in the presence of the ‘exception also’ keyword may include exception property values (negative values for the area_ration property). Be careful when using exception also with histograms.

In this example, notice that the histogram FLAT GEOMETRY COUNT (FGC) for histogram-enabled layer e3 of 1917 agrees with the count in the SVRF transcript for an identical setlayer operation without the histogram block (aov_e4) of FGC=1917.

Note
For the histogram SVRF layer aov_e3, the Flat Geometry Count (FGC) is 0.

The histogram output file histogram_e3.txt contains information similar to the following:

```
0.04            1060
0.05               2
0.06             462
0.07             261
0.09             132
```

Specifies the derived layer to extract from the Calibre MPCverify command setup file.

Using LITHO MPCverify

Use these steps to create a Calibre MPCverify command file and one or more LITHO MPCVERIFY calls for use in an SVRF rule file.

Prerequisites

- An OASIS or GDS design file with layers to be used as input for Calibre MPCverify

Procedure

1. First, create an Calibre MPCverify command setup file, using the following sequence:

   a. Setup commands that include Calibre MPCverify layer definitions

      Setup commands are described in the section "Calibre MPCverify Setup File Configuration Commands."

   b. Setlayer commands that operate on Calibre MPCverify layers

      Setlayer commands are described starting with the section "Setlayer Operations Reference."

2. In a separate SVRF rule file, define the mapping between the design file layer numbers and the Calibre MPCverify layers you defined inside the Calibre MPCverify command setup file.

3. Run Calibre MPCverify by defining one or more LITHO MPCVERIFY output layer definition calls from the SVRF rule file to the Calibre MPCverify command setup file, using the following syntax:

```
L1 = LITHO MPCVERIFY input_layer1 ... input_layerN FILE cmd_file \
   MAP MPCverify_layer1
L2 = LITHO MPCVERIFY input_layer1 ... input_layerN FILE cmd_file \
   MAP MPCverify_layer2
...
Ln = LITHO MPCVERIFY input_layer1 ... input_layerN FILE cmd_file \
   MAP MPCverify_layerN
```

4. Optionally, you can output the results of MPCVERIFY to the design file, using the following syntax:

```
Ln {COPY MPCverify_layerN} DRC CHECK MAP MPCverify_layerN\
   layer_number
```

5. Run Calibre DRC on the SVRF file:

```
calibre -drc -hier -turbo -turbo_litho filename.svrf
```

Results

The output of the SVRF rule file is written to the file designated in the SVRF rule file.

LITHO MPCVERIFY

SVRF Command

Runs the Litho MPCverify tool on the specified command file. Litho MPCverify is a Calibre batch command. It performs a number of dense simulations on a simulation grid at various process conditions. It then performs a set of user-defined check operations on those dense simulation results.

Usage

LITHO MPCVERIFY *input_layer1* [*input_layerN*] ... **FILE *cmd_file* MAP *MPCverify_layer*** [INSIDE OF LAYER *region*]

Arguments

- ***input_layer1*** ... *input_layerN*

  Identifies one or more layers defined with LAYER commands in the SVRF rule file to be input to the Calibre MPCverify command file.

- ***cmd_file***

  Specifies the name of the MPCverify command file to run.

- ***MPCverify_layer***

  Specifies the name of the output layer from the Calibre MPCverify command file. A setlayer command inside the command file that results in a layer name matching ***MPCverify_layer*** is copied to the SVRF layer for this LITHO MPCVERIFY command.

- INSIDE OF LAYER *region*

  Optionally invokes the region iterator. The region iterator allows efficient execution of Calibre MPCverify inside a set of small user-defined regions. All regions that are smaller than the tilemicrons setting are processed as one tile, which can improve the quality of the results.

  Using INSIDE OF LAYER forces the use of these Calibre MPCverify features and commands even if they are not explicitly specified:

  - LAYOUT ULTRA FLEX YES
  - HDB construction mode
  - processing_mode flat

  The region iterator is equivalent to the following operations:

  ```
  regs_plus = SIZE regions BY interaction_distance
  clip_l1 = l1 AND regs_plus
  clip_l2 = l2 AND regs_plus
  out_plus = LITHO MPCVERIFY clip_l1 clip_l2 FILE mpcv_file MAP out
  out = out_plus AND regions
  ```

The *interaction_distance* is the Calibre MPCverify distance sufficient to ensure that the results inside the region are the same as a full-chip run. This interaction distance is calculated automatically, and can be found in the Calibre MPCverify run transcript as a line starting with "SE: INTERACTION DISTANCE."

Limitations

- When using Calibre MPCverify with the region iterator, there are some usage limitations.
    - Images computed with the region iterator are very close to but not exactly the same as the images computed in a full-chip run, because simulation frames are placed differently. For well-behaved models, the difference should be less than 0.1nm for 1D regions and slightly larger for 2D regions.
    - Even tiny changes in images can lead to some errors disappearing or new errors appearing. For example, if a check is "pinch target contour < 0.018" and the full-chip run finds a pinch with a "min" property of 0.01795, then the contour jitter of the region iterator of 0.1nm can lead to a new "min" value of 0.01805, causing the entire error to disappear.
    - If an error is only partially covered by the filter, its region iterator properties can differ significantly from full-chip run properties.

Examples

Example 1

A standard Calibre MPCverify run command that takes the layer POLY_MPC as input and returns the result of the command "setlayer image1 = ..." as output, copied into the SVRF layer L1.

```
L1 = LITHO MPCVERIFY POLY_MPC FILE mpcverify.in MAP image1
```

Example 3

This example uses the region iterator with a set of hotspots on the layer "regions".

```
out = LITHO MPCVERIFY l1 l2 ... FILE mpcv_file MAP out INSIDE OF \
      LAYER regions
```

Calibre MPCverify Setup File Configuration Commands

The *cmdfile* argument to LITHO MPCVERIFY specifies a file (or an inline file) containing a *case-sensitive* command initialization block consisting of commands, specified one per line.

Table 4-3. Calibre MPCverify Setup File Configuration Commands

Command	Description	Instances Allowed
ebeam_model_load	Loads an e-beam model.	1+
etch_imagegrid	Sets the etch model grid independently of imagegrid.	1
etch_model_load	Loads an etch model.	0+ (optional)
setlayer	Derives a Calibre MPCverify layer, depending on the subcommand/option specified.	1+

Tip

Setlayer commands are described starting with the section "Setlayer Operations Reference."

There are a number of Calibre OPCverify setup file commands that are supported by Calibre MPCverify. These are documented in the *Calibre OPCverify User's and Reference Manual* .

Table 4-4. Calibre OPCverify Setup File Commands Supported by Calibre MPCverify

Command	Description
modelpath	A colon (":") separated list of directories to search for optical and resist models.
optical_model_load	Loads an optical model. At least one optical_model_load command must be present unless a litho model is loaded.

Table 4-4. Calibre OPCverify Setup File Commands Supported by Calibre MPCverify (cont.)

Command	Description
background	Defines the background transmission values of the mask for each exposure. **Note:** The background command is required for syntax compliance only and is not otherwise used by Calibre MPCverify. However, you must always define background opposite to the layer (for example, background set to dark and layer set to clear), otherwise the tool issues a syntax error.
resist_model_load	Loads a VT5 resist model.
etch_model_load	Loads an etch model.
ddm_model_load	Loads a Domain Decomposition Model (DDM).
flare_longrange	Loads long range flare information from a litho model.
flare_model_load	Loads a flare model.
euv_slit_x_center	Sets the x-coordinate for the center of an EUV through-slit model extension.
shadow_bias_model_load	Loads a shadow bias model.
topo_model_load	Loads a topo model if a Litho Model is not being used for them.
processing_mode	Switches the processing mode for Calibre MPCverify.
classify_options	Defines default classification block settings.
critical_dimension	Sets default CD values for multiple commands.
dynamic_output	Sets Calibre MPCverify to output interim layers during simulation.
layer	Defines the input layers and optionally, how they behave in optical simulations. This command must appear before the first setlayer command.
log_options	Toggles display of various log messages.
collect_frame_stats	Logs frame processing statistics.

Table 4-4. Calibre OPCverify Setup File Commands Supported by Calibre MPCverify (cont.)

Command	Description
promote_subframe_slivers	Promotes small slivers geometries out of frames.
contour_options	Sets anchored contour mode.
optical_transform_size	Manually sets the optical transform size.
filter	Selects a user-defined *input_layer* (which must be one of the input layers to Calibre MPCverify) as a filter.
gauge_set	Creates a named block containing gauge definition rules.
image_options	Creates a named block containing preset options for the image command.
image_set	Creates a group of image contours varied by specified criteria.
pw_annotate	Adds properties to error marker layers for process window commands.
pw_annotate_options	Sets options for pw_annotate.
imagegrid	Specifies the simulation grid in microns.
mask_sample_grid	Specifies the mask sampling grid in microns. Use only to activate RSM mode.
save_error_center_points	Writes error center coordinates to a file.
staristep	Toggles planar interpolation versus discrete pixels to represent the final contour. (This option is only used for aerial contour models, and is ignored for VT5 models.)
svrf_var_import	Imports one or more variables previously defined in the SVRF rule file.
clone_transformed_cells	Sets how Calibre MPCverify handles rotated geometries.
simulation_deangle	Passes the output of all image commands through the deangler.
summary_report	Outputs an HTML report page with snapshots.

Table 4-4. Calibre OPCverify Setup File Commands Supported by Calibre MPCverify (cont.)

Command	Description
tilemicrons	Specifies the size of a Calibre MPCverify tile.
progress_meter	Toggles the reporting of cell processing information in the output log.
setlayer	Derives a Calibre MPCverify layer, depending on the subcommand/option specified.
output_window	Outputs specified layers with context and size boundaries.

ebeam_model_load

Calibre MPCverify setup command

Defines and loads the e-beam model.

Usage

ebeam_model_load *ebeam_model_name* [***filepath*** **| ‘{’ *inline* ‘}’**] **‘{’**
  model_type simulation
  diff_length *diffusion_length...*
  [eta (*eta_i*)...]
  [kernel_type {gaussian | exponential}...]
  [sample_spacing *sample_space_val*]
  [max_segment *segment_length*]
  [log_level 0-100]
  [sequence LUMPED]

‘}’

Arguments

- ***ebeam_model_name***

  A required argument that specifies the e-beam model name. One of either ***filepath*** or {***inline***} must be specified.

- ***filepath***

  A required argument that specifies a path to the etch model.

- **‘{’ *inline*‘}’**

  A required argument that specifies the e-beam model specifications inline; the braces ({ }) are required and must surround the inline model definition.

- **model_type simulation**

  A required keyword set that specifies lithographic simulations using geometry on the input layer by convolving the input image with a Gaussian kernel of the specified diffusion length.

- **diff_length *diffusion_length...***

  A required argument set that specifies the gaussian diffusion length sigma in user units (usually um). This value is sqrt(2) larger than in the “standard deviation” form of the gaussian function. There must be at least one “very short” range term, with a diffusion length less than 0.1 user unit. The maximum allowed diffusion length is 1.0 user unit.

- eta (*eta_i*)...

  An optional argument that specifies the weight of the terms in the multigaussian kernel. The terms must be positive and add to 1.0. The number of arguments must match the number of arguments to diff_length and be in the same order. This keyword may be omitted if only one kernel is used.

If you are using multiple kernels, the eta values for the different kernels must add up to 1.0. The sequence of sigma values, eta values, and kernel types must match. For example, when using three kernels, you specify:

```
model_type   simulation
diff_length  0.025    0.065    1.60
eta          0.70     0.24     0.06
kernel_type  gaussian gaussian exponential
```

In the previous example, the third kernel is an exponential kernel with a 1.6um sigma and contributes 6% to the model signal.

- kernel_type {gaussian | exponential}...

  An optional argument set that specifies the kernel type. The number of arguments must match the number of arguments to diff_length and have the same order. This keyword may be omitted if only one kernel is used.

- sample_spacing *sample_space_val*

  An optional argument that specifies sampling spacing, which defaults to *diffusion_length*/3. Larger values tend to improve performance, and smaller values improve accuracy. In addition, sample_spacing influences the effect of max_iter_movement: the effective value of max_iter_movement cannot be less than that of sample_spacing. The option sample_spacing applies to simulation models only.

- max_segment *segment_length*

  An optional argument that triggers a high resolution contour generation mode that breaks down pixels into smaller pixels where necessary. High resolution mode is triggered when the max_segment is less than sample_spacing * sqrt(2); otherwise, it is not in effect.

- log_level 0-100

  An optional keyword set that specifies the amount of information printed to stdout: 0 = nothing, 10 = start/stop, 20 = errors, 30 = warnings, 40 = info, 50 = debug, 60 = prints. The default setting is 0.

- sequence LUMPED

  An optional keyword set that enables a “lumped” model mode. In this mode, instead of applying the e-beam and VEB (etch) models in sequence (requiring saving the intermediate results of both models at the expense of run time), both models are treated as a single model and the signal of both is calculated in a single step.

  Note

  When switching to “lumped” model mode, the model must be re-calibrated since the simulated signal using identical parameters changes slightly.

Description

The ebeam_model_load command defines an E-beam model. This is required if the **ebeam_model** form of the ebeam_simulate command is used. The setlayer ebeam_simulate command performs pseudo-lithographic simulations using geometry on the input layer.

The E-beam model simulates a special class of short-range effects, known as E-beam effects, are typically triggered by forward scattering and beam blur.

Examples

```
ebeam_model_load ebeam_25nm {
   diff_length 0.024
   model_type simulation
   sample_spacing 0.008
   }
```

etch_imagegrid

Calibre MPCverify setup command

Describes the grid size use for etch model simulation purposes.

Usage

etch_imagegrid *etch_grid_microns*

Arguments

- ***etch_grid_microns***

  A required argument that specifies the etch grid size.

Description

Defines the grid size to use for etch simulations only.

- If the imagegrid option is explicitly specified, **etch_imagegrid** defaults to the value specified for imagegrid.
- If **etch_imagegrid** is specified, however, it overrides the supplied value for **imagegrid** for etch simulation (see veb_simulate) only.
- If neither **etch_imagegrid** nor **imagegrid** are specified, the grid value used for etch simulation is derived from the etch model's parameters.

Tip

Performance Warning - Smaller values of etch_imagegrid will slow down performance.

etch_model_load

Calibre MPCverify setup command

Loads the specified etch model in Calibre MPCverify.

Usage

etch_model_load ***etch_model_name*** [***filepath*** **|** ‘**{**’ ***inline*** ‘**}**’]

Arguments

- ***etch_model_name***

  A required argument specifying a name that you will later use to refer to the etch model in the image and veb_simulate setlayer commands.

  One of either ***filepath*** or {***inline***} must be specified.

- ***filepath***

  Specifies a path to the etch model.

- ‘**{**’ ***inline***‘**}**’

  Specifies the etch model specifications inline; the braces ({ }) are required and must surround the inline model definition. For the definition of VEB etch models, see the “Variable Etch Bias (VEB Model File Format)” reference page in the *Calibre WORKbench User’s and Reference Manual*.

Description

Loads the specified etch model for visible etch bias (VEB) calculations.

setlayer

Calibre MPCverify setup command

Creates and derives layers, depending on the options used. You create Calibre MPCverify layers using the design layers you defined with the layer command, and you will eventually use one or more of them as output in your LITHO MPCVERIFY calls in your SVRF file.

Usage

setlayer *output_layer_name* = ***setlayer_operation*** [*setlayer_command_options*]

Arguments

- ***output_layer_name***

  A required option specifying the name of the new output layer. The new layer is created and the results of the operation are written to the layer.

- ***setlayer_operation***

  One of the following operations:

  - Image Operations

    An image operation performs optical and resist simulation using geometry on visible layers. In the case of multiple exposures, different options can be specified for each exposure, allowing you to simulate a wider array of lithographic conditions.

  - DRC-Type Operations

    Similar to Calibre DRC, you can use setlayer to work with the polygons on the source layers, outputting the result to the output layer.

  - Verification Control Operations

    Calibre MPCverify includes a number of special operations to perform activities on layers.

- *setlayer_command_options*

  The typical setlayer operation requires one or more options, which may be layers, values, constraints, flags, or other switches depending on the operation.

> **Tip**
> The setlayer operations are all arguments to the setlayer command, but are described separately, starting with the section “Setlayer Operations Reference.”

Setlayer Operations Reference

All commands that create or modify layers are known as setlayer operations.

The following sections describe setlayer operations and list all setlayer-related commands.

Using Setlayer Options as Operations

Calibre MPCverify uses setlayer options (sometimes referred to as setlayer operations, setlayer subcommands, or setlayer commands) to create and derive layers.

All setlayer operations consist of a single *case-sensitive* line that uses the syntax:

```
setlayer output_layer_name = setlayer_operation
```

A *setlayer_operation* is one of the following types:

- Image Operations
- DRC-type Operations
- Verification Control Operations

Calibre MPCverify-Specific Setlayer Operations

The following table lists the setlayer operations utilized specifically for Calibre MPCverify.

Table 4-5. Calibre MPCverify Setlayer Operations

Operation	Description
ebeam_simulate	Simulates E-beam effects on an input layer.
veb_simulate	Simulates Variable Etch Bias (VEB) effects on an input layer.

Calibre OPCVerify Setlayer Operations Supported by Calibre MPCverify

Calibre MPCverify and Calibre OPCverify share the majority of their setlayer options. These are documented in the *Calibre OPCverify User's and Reference Manual*

An image operation performs optical and resist simulation using geometry on visible layers defined by the layer command, subject to the command options. Image operations are described in the image syntax page described in the *Calibre OPCverify User's and Reference Manual*.

The DRC-type operations are implemented for use with Calibre OPCverify setlayer commands. They must be entered in case-sensitive text.

Table 4-6. Calibre OPCverify Setlayer Ops in Calibre MPCverify (DRC-type)

Operation	Description
and	Performs multilayer AND on inputs.
copy	Copies the layer to a new layer.
enclosure	Performs an enclosure check within the specified region.
external	Performs an external check within the specified region.
internal	Performs an internal check within the specified region.
not	Performs a multilayer NOT of all inputs.
or	Performs a multilayer OR of all inputs.
size	Performs the sizing operation on the input layer, expanding or shrinking polygons on the input layer by the specified amount.
width	Performs a width check on the specified region.
xor	Performs a multilayer XOR of all inputs.

The customized verification control operations are implemented for use with setlayer commands. They must be entered in case-sensitive text.

Table 4-7. Calibre OPCverify Setlayer Ops in Calibre MPCverify (Verification Control)

Operation	Description
annotate	Annotates an input layer with shapes matching the selection criteria.
area_compute	Checks for shapes that match a certain area constraint.
area_overlay	Performs a contact / via alignment check.
area_ratio	Checks for shapes that meet a certain area ratio constraint.
bandcheck	Checks CD accuracy for inner and outer tolerance band violations.
bridge	Checks for bridging problems.
center_shift	Measures projections of contacts and vias from a target.
circularity_compute	Measures the circularity of simulated contours.
contour_diff	Calculates the error factor between two contours, similar to meefcheck.
empty_layer	Creates an empty layer.

Table 4-7. Calibre OPCverify Setlayer Ops in Calibre MPCverify (Verification Control) (cont.)

Operation	Description
end_cap	Checks endcap coverage.
filter_generate	Generates a filter layer for the measure_cd command.
gate_stats	Computes statistics for a gate area.
gauges	Adds properties to an output layer based on user-defined gauges.
holes	Returns holes inside layer shapes.
identify_corner	Identifies line fragments of edges near corners.
identify_edge	Identifies line ends and jogs.
imax_check	Analyzes the aerial image intensity for an image contour against an Imax constraint.
imin_check	Analyzes the aerial image intensity for an image contour against an Imin constraint.
interact	Checks for shapes that interact with each other.
measure_cd	Checks width and space CD accuracy.
measure_cdv	Checks CD between two contours versus a target CD.
measure_cross_section	Checks the cross section of a contour versus a target shape.
measure_distance	Checks the distance between an edge and the nearest edge.
measure_epe	Checks if a layer's EPE fails a specified constraint.
nilscheck	Checks the Normalized Image Log-Slope (NILS) for two or more contours for a specified range constraint.
not_printing	Checks for features that do not print.
notchfill	Fills in notches on polygons.
pinch	Checks for pinching problems.
pinch_tolerance	Checks for inner tolerance band violations.
polygon_extent	Checks the area of polygons in x and y dimensions.
pvband	Generates a process variation band from multiple contours.
pwcheck	Checks the common process window versus a process window.
shadow_bias	Shifts a layer according to a shadow bias model.
shift	Shifts the polygons on a layer a distance in x and y.
show_annotation	Writes annotation markers to an output layer.

Table 4-7. Calibre OPCverify Setlayer Ops in Calibre MPCverify (Verification Control) (cont.)

Operation	Description
window	Outputs context clips based on an error layer.

ebeam_simulate

Calibre MPCverify setlayer command

Simulates E-beam effects on an input layer.

Usage

setlayer ***output_layer_name*** = **ebeam_simulate** {
{***input_layer_name*** ebeam_model ***ebeam_model_name*** [dose *dose_value*]}...
[etch_model *etch_model_name*] | *image_options_name*}

Arguments

- ***output_layer_name***

  A required keyword specifying the name of the new output target layer. The new layer is created and the results of the operation are written to the layer. This layer must be a valid Calibre layer containing polygon data.

- ***input_layer_name***

  Specifies the name of the input layer. The layer must be a valid Calibre layer containing polygon data.

- {***input_layer_name*** **ebeam_model** *ebeam_model_name*[dose *dose_value*]}...

  Specifies an E-beam model as created by ebeam_model_load. The **ebeam_model** syntax requires ***ebeam_model_name*** be defined before **ebeam_simulate**. Each input layer block includes the layer name, the applied E-beam model, and the dose value (where the dose is optional) and the default value is 1. Currently, all input layers must share the same E-beam model. Multiple input layers can be specified.

- etch_model *etch_model_name*

  Specifies an optional etch model.

- *image_options_name*

  Specifies an image_options structure. The image_options structure is defined as follows:

```
image_options <variable> {
   layer <layer_name> {visible | hidden}
litho_model <dir>
}
```

Description

The ebeam_simulate command performs pseudo-lithographic simulations using geometry on the input layer. It differs from the image command in that instead of performing a convolution for the entire contour, it computes explicitly a number of key points on the contour and connects them with edges that approximate the desired contour.

Examples

Multiple Input Layers

The following example shows a simulation model with multiple input layers.

```
ebeam_model_load "ebeam_25nm" {
   model_type simulation
   diff_length 0.025
}
setlayer eb_contour = ebeam_simulate poly0 ebeam_model\
   "ebeam_25nm" dose 0.8 poly1 ebeam_model “ebeam_25nm” dose 1.2\
   etch_model etch
```

Default Sample Spacing

The following example shows a simulation model with default sample spacing.

```
ebeam_model_load "ebeam_25nm" {
   model_type simulation
   diff_length 0.025
}
setlayer eb_contour = ebeam_simulate ebeam_model "ebeam_25nm"
```

identify_edge

Verification control for Calibre MPCverify

Identifies line ends and jogs.

Usage

identify_edge *layer* length *length_constraint*
[length1 *constraint*]
[length2 *constraint*]
[corner1 {convex | concave | *angle_constr*}]
[corner2 {convex | concave | *angle_constr*}]
[complement]
[trim_ends *trim* [*min_length*]]
[extend [by_factor] *ext_value*]
expand {***value*** | ***value_in value_out***}

Description

This operation can be used to identify line ends or jogs. It is analogous to the DRC operation CONVEX_EDGE. It then performs end trimming and span complementation, followed by edge expansion in the normal direction and the parallel direction to create the final output shape.

Arguments

- ***layer***

  A required argument, specifying the layer containing the shapes to analyze.

- **length *length_constraint***

  A required argument specifying a length constraint for the edge to test.

  > **Note**
  > Avoid using large ***length_constraint*** values, which increase interaction distance. This can result in flattening of data and slower run times.

- length1 *constraint*

  An optional argument, specifying a minimum length constraint for the first edge adjacent to the edge being tested. If an edge being tested does not pass the given constraint, the edge fails and is not included in the result set.

  If only length1 is specified, the edge passes if either adjacent edge to the edge being tested passes the constraint.

- length2 *constraint*

  An optional argument, specifying a minimum length constraint for the second edge adjacent to the edge being tested. Both adjacent edges to the edge being tested must meet their constraints for the edge to pass. Note that the algorithm tests the adjacent edges twice (given the endpoints A, B of an edge, the tool tests A/length1, B/length2, and A/length2, B/length1). If either permutation passes both constraints, the edge passes.

Tip

See "Constraints" on page 45 for more information on constraint syntax.

- corner1 {convex | concave | *angle_constr*}

  An optional argument, specifying a constraint on the corner for the adjacent edge to the edge being examined.

  If only corner1 is specified and either corner meets the constraint, the edge passes.

  If length1 is also specified, an adjacent edge must pass the combined length1 and corner1 constraints to pass.

- corner2 {convex | concave | *angle_constr*}

  An optional argument specifying a constraint for the corner for the second adjacent edge. Both adjacent corners must meet their constraints for the edge to pass. The tool tests the adjacent corners using both permutations (endpoint A/corner1 with endpoint B/corner2, and endpoint A/corner2 with endpoint B/corner1). If either permutation passes both constraints, the edge passes.

  If length2 is also specified, both adjacent edges must pass the set of combined constraints (length1 (and if specified, corner1) and length2, corner2) for this edge to pass.

- complement

  An optional argument that identifies edges that do not meet the specified constraints.

- trim_ends *trim* [*min_length*]

  An optional argument that enables edge trimming. The parameter *trim* specifies the length (in microns) that will be cut off each end of the output edges. The parameter *min_length* specifies the minimum edge length (in microns) after trimming.

  For short edges, the trim length is adjusted to preserve the central portion of min_length in ums. If min_length is not specified, the default value is 3 dbu. The trim_ends option cannot be specified together with the extend keyword.

- extend [by_factor] *ext_value*

  An optional argument, specifying how far along the identified edge to extend or shrink the marker. Without the optional by_factor keyword, *ext_value* specifies a positive or negative extension in microns.

  If the optional by_factor flag is specified, then *ext_value* must be greater than -0.5 and cannot be zero. The extension length is computed individually as the product of an edge's length and *ext_value*.

  - If the extension length is positive, the edge is extended from its endpoints by the computed value.

- If the extension length is negative, the edge is trimmed by the absolute value of the extension length on both ends. However, the edge trimming is adjusted if needed to preserve a minimum edge length of 3 dbu.

- **expand *value* | *value_in value_out***

  A required argument, specifying the amount to expand the marked edges by.

  - Specifying a single value expands the corner inside and outside of the polygon.
  - You can specify different values for the expansion inside and outside the polygon using the second argument format (***value_in value_out***).

Examples

The following code selects any line edges less than .11 um, expanding them by .01 um:

```
setlayer ide1 = identify_edge TARGET length < .11 expand .01
```

The following more restrictive version of the code example detects mainly line ends and some jogs:

```
setlayer ide2 = identify_edge TARGET length < .11 length1 > .1 \
   length2 > .1 expand .01
```

The following even more restrictive version will detect only line ends:

```
setlayer ide3 = identify_edge TARGET length < .11 length1 > .1 \
   length2 > .1 corner1 convex corner2 convex expand .01
```

The following alternate version detects space ends:

```
setlayer ide4 = identify_edge TARGET length < .11 length1 > .1 \
   length2 > .1 corner1 concave corner2 concave expand .01
```

For fragments corrected with the SEM Box correction method in Calibre nmMPC (refer to the section "SEM Box Correction" in the *Calibre nmMPC and Calibre nmCLMPC User's and Reference Manual* for a description of SEM Box correction), use a combination of identify_edge with the by_factor *factor* keyword and the measure_epe command with the gauges_per_fragment *count* keyword. In the following example, the *factor* value is the fraction of an edge not covered by the SEM Box, divided by two.

```
setlayer lineEndsSemBox  = identify_edge MpcTarget\
  length <= <semBoxLimit> length1 > <semBoxAdjLen> length2 >\
  <semBoxAdjLen> corner1 convex corner2 convex\
  extend by_factor -0.25 expand 0.005
setlayer epeSemBox  = measure_epe sim MpcTarget inside\
  semBox gauges_per_fragent <count> function  average\
  min_featsize 0.030 max_edgelen 0.25 fragment only\
  not > -0.001 < 0.001 output_expanded_edges 0.001 \
  trim_ends 0.002 property {average}
```

For example, if the SEM Box size in correction is specified as 0.6 (the SEM Box covers the center 60% of the edge), the factor in Calibre MPCverify should be 0.2. The gauges_per_fragment count must be the same number as used in Calibre nmMPC. In this example, also note that average is specified as the function in measure_epe as the correction engine also minimizes the average EPE across the SEM Box in Calibre nmMPC.

veb_simulate

Calibre MPCverify setlayer command

Performs etch bias simulation.

Usage

setlayer *output_layer_name* = veb_simulate *layer* etch_model *veb_model*

Arguments

- ***output_layer_name***

  A required keyword specifying the name of the new output target layer. The new layer is created and the results of the operation are written to the layer. This layer must be a valid Calibre layer containing polygon data.

- ***layer***

  A required argument. This layer should contain the resist contour before etch biasing.

- ***veb_model***

  A required argument specifying the name of the VEB etch model to use for the simulation.

Description

Performs an etch bias simulation on an input contour layer. Used for model-based retargeting verification and VEB model verification purposes. The etch model being used should have been previously loaded with an etch_model_load command.

Examples

```
setlayer etch_contour = veb_simulate resist_contour etch_model etch_mod
```

Index

Third-Party Information

Details on open source and third-party software that may be included with this product are available in the *<your_software_installation_location>/legal* directory.