Contents

Preface ............................................................................................................ 13
  Related Documents ..................................................................................... 13
  Typographic and Syntax Conventions ....................................................... 14
    SKILL Syntax Examples ............................................................................. 15

1 Introducing Diva Verification ................................................................... 17
  Diva Verification Tool Set ........................................................................ 17
  Diva Verification Programs ....................................................................... 19
  Diva Verification Product Flow ................................................................ 19

Prerequisites for Diva Verification ............................................................... 20
  Design Framework II ................................................................................. 21
  Simulation Environment ............................................................................ 21
  Verification Rule File ................................................................................ 21

Using Diva Verification ................................................................................. 21
  About the Mouse ....................................................................................... 22
  Using the Verify Menu .............................................................................. 23
  Using Forms ................................................................................................ 24

2 Running Diva Verification ......................................................................... 28
  Interactive Mode ....................................................................................... 30
  Batch Mode ................................................................................................ 33
  Remote Mode .............................................................................................. 34

Controlling Diva Verification ....................................................................... 38
  Controlling Run Modes Using Properties ............................................... 39
  Creating a Log File .................................................................................... 41
  Using Checkpoints ..................................................................................... 43
  Conditional Execution of DRC and Extraction Rules (ivIf) ....................... 45
  Default ivIf Switch Values ....................................................................... 50
  Rules File Example ................................................................................... 51
  Controlling SKILL Access ........................................................................ 54
  Special Functions ....................................................................................... 57
3 Executing Diva Verification Using SKILL Functions

- ivConcICe .............................................. 68
- ivCreatePCells ...................................... 71
- ivDRC .................................................. 73
- ivERC .................................................. 77
- ivExtract ............................................. 79
- ivLVS .................................................. 83
- ivRestart ............................................. 87

Executing Diva Verification using UNIX ........................................ 88
- ivVerify .............................................. 89
### 5

#### Layer Processing Concepts

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Layers</td>
<td>131</td>
</tr>
<tr>
<td>Shapes</td>
<td>133</td>
</tr>
<tr>
<td>Edge Boolean Functions</td>
<td>135</td>
</tr>
<tr>
<td>Invalid Graphics Data</td>
<td>137</td>
</tr>
<tr>
<td>Exclusive Selection</td>
<td>138</td>
</tr>
<tr>
<td>Data Integrity Checks</td>
<td>139</td>
</tr>
<tr>
<td>dubiousData</td>
<td>140</td>
</tr>
<tr>
<td>offGrid</td>
<td>142</td>
</tr>
<tr>
<td>Logical Functions</td>
<td>143</td>
</tr>
<tr>
<td>geomAnd</td>
<td>144</td>
</tr>
<tr>
<td>geomAndNot</td>
<td>146</td>
</tr>
<tr>
<td>geomCat</td>
<td>148</td>
</tr>
<tr>
<td>geomNot</td>
<td>149</td>
</tr>
<tr>
<td>geomOr</td>
<td>150</td>
</tr>
<tr>
<td>geomXor</td>
<td>152</td>
</tr>
<tr>
<td>Relational Selection Functions</td>
<td>153</td>
</tr>
<tr>
<td>geomAvoiding</td>
<td>154</td>
</tr>
<tr>
<td>geomButting</td>
<td>155</td>
</tr>
<tr>
<td>geomButtOnly</td>
<td>158</td>
</tr>
<tr>
<td>geomButtOrCoin</td>
<td>161</td>
</tr>
<tr>
<td>geomButtOrOver</td>
<td>164</td>
</tr>
<tr>
<td>geomCoincident</td>
<td>167</td>
</tr>
<tr>
<td>geomCoinOnly</td>
<td>170</td>
</tr>
<tr>
<td>geomEnclose</td>
<td>173</td>
</tr>
<tr>
<td>geomInside</td>
<td>176</td>
</tr>
<tr>
<td>geomLineEnd</td>
<td>178</td>
</tr>
<tr>
<td>geomOutside</td>
<td>183</td>
</tr>
<tr>
<td>geomOverlap</td>
<td>184</td>
</tr>
<tr>
<td>geomStraddle</td>
<td>187</td>
</tr>
<tr>
<td>Sizing Functions</td>
<td>189</td>
</tr>
<tr>
<td>geomSize</td>
<td>190</td>
</tr>
</tbody>
</table>
7
DRC Overview ......................................................... 257
   Modes of Operation ........................................ 257
   Area Halo Processing ........................................ 261
   DRC Limitations ............................................. 262
How DRC Performs Checks ........................................ 266
   Looking Through the Wall .................................. 270
   Check Conjunction .......................................... 271
   Alignment and Registration ................................ 272
   Cell-Based Options ......................................... 277
Checking Commands ................................................ 278
   checkAllLayers ............................................... 279
   checkLayer ................................................... 281
   drc .......................................................... 283
   drcAntenna .................................................. 286
Functions ......................................................... 291
   area .......................................................... 292
   enc .......................................................... 293
   notch ......................................................... 295
   ovlp ........................................................ 296
   sep .......................................................... 298
   width ......................................................... 300
Function Modifiers ............................................... 301
   app .......................................................... 302
   diffNet ....................................................... 304
   length, lengtha, lengthb .................................. 305
   notParallel .................................................. 307
   only_perp .................................................... 308
   opposite ..................................................... 309
   parallel ....................................................... 311
   sameNet ...................................................... 312
   shielded, shieldedA, shieldedB .......................... 313
   with_perp .................................................... 315
Output Modifiers ................................................ 316
   edge, edgea, edgeb ......................................... 317
   fig, figa, figb ............................................... 320
Alignment and Registration Modifiers ......................... 321
   normalGrow ................................................ 322
   squareGrow ................................................ 323
Other Modifiers ..................................................... 323
8 Models of Device Extraction

Flat Mode ................................................................. 326
Macro Cell Mode ......................................................... 326
Full Hierarchical Mode ................................................. 327
Incremental Hierarchical Mode .................................... 330
Device Recognition ..................................................... 331
Using Device Recognition ........................................... 331
Device Recognition Polygons ....................................... 332
Processing Pins ........................................................ 334
Device Extraction Commands ...................................... 336
extractDevice ............................................................ 337
extractMOS ................................................................. 341
Data Storage ............................................................. 344
saveInterconnect ......................................................... 345
saveProperty .............................................................. 346
saveRecognition ......................................................... 347

9 Extracting Parameters (iLPE) ...................................... 348
Introduction .............................................................. 348
Measurement Process ................................................ 349
Measurement Functions ............................................. 349
NULL Measurements .................................................. 351
Measurement Optimization ........................................ 352
Parameter Measurement Reference Commands ............. 353
calculateParameter ..................................................... 354
measureParameter ....................................................... 358
Data Storage ............................................................. 363
saveParameter ........................................................... 364

10 Extracting Parasitics (iLPE) ...................................... 365
Introduction .............................................................. 365
## Extracting Parasitic Resistance (iPRE)

### Introduction

- What Is Parasitic Resistance Extraction? 420
- How Do I Invoke It? 420
- What Are the Results? 420

### The Extraction Process

- Preconnect 423
- Postconnect 423

### Resistor Formation

- Contact Resistance 423
- Current Flow 425
- Edge Resistance 425
- Area Resistance 426

### Device Terminals

- Bend Resistance 429
- Transition Resistance 430

### Network Reduction

- Cross Coupling and Fringe Capacitance 431
- Resistor Body Capacitance 435
- R-C Models 435
- Netlisting R-C Models 436

### Multiple Layer Extraction

- 438
12

Checking Electrical Rules (iERC) ................................................. 450

Introduction ................................................................. 450
   Prerequisites .......................................................... 451
   Program Functionality .............................................. 451
   Program Output ....................................................... 453
ERC Functions ............................................................. 454
   Reducing Networks .................................................. 454
   Consolidating and Testing Parameters ......................... 460
ERC Commands ............................................................ 466
   oneWayPath .......................................................... 467
   setGround ............................................................. 468
   setInput ............................................................... 469
   setOutput ............................................................. 470
   setPower ............................................................... 471
   twoWayPath ........................................................... 472
Tracing and Connectivity Commands ..................................... 472
   checkConnected ..................................................... 473
   checkDeviceNetCount .............................................. 474
   checkFanOut .......................................................... 476
   checkFloatingDevices .............................................. 479
   checkFloatingNets ................................................... 480
   checkNotConnected .................................................. 481
   checkOneNetDevices ............................................... 482
   checkOneTerminalNets ............................................. 483
   checkPullDown ....................................................... 484
   checkPullUp .......................................................... 485
   checkPullUpAndDown ............................................... 486
Gates and Parameters Commands ............................................ 486
   displayFeedBackLoops .............................................. 487
   displayGateErrors ................................................... 488
   reduceDevice ......................................................... 489
13 Comparing Layout to Schematic (iLVS) ........................................... 496

Introduction .................................................................................. 496
  Prerequisites ............................................................................... 497
  Capabilities ............................................................................... 497
  Backannotation .......................................................................... 498
Verification Rules Functions .......................................................... 498
  Permutating Devices .................................................................... 498
  Ignoring Device Terminals ........................................................... 504
  Ignoring Extraneous Devices ......................................................... 505
  Removing Unwanted Devices ........................................................ 507
  Parameter Analysis ....................................................................... 507
  Device Fixing ............................................................................... 515
  Correspondence Points .................................................................. 516
  Network Terminal Processing ......................................................... 518
Comparing Macro Cell ................................................................... 520

Output Files .................................................................................... 520
  Error Files .................................................................................. 521
  Displaying Errors ......................................................................... 523
LVS Verification Rules Commands .................................................. 525
  compareDeviceProperty ................................................................. 526
  ignoreTerminal ............................................................................. 527
  parameterMatchType .................................................................... 528
  permuteDevice .............................................................................. 530
  pruneDevice ................................................................................ 533
  removeDevice ............................................................................... 535
Backannotation (lvsbx) .................................................................. 536

14 Verify Menu Commands ............................................................... 539

Introduction .................................................................................... 539
DRC Command ................................................................................. 539
Extract Command ............................................................................. 545
ERC Command ................................................................................ 549
LVS Command ............................................................................... 556
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Inherited Connections for LVS in Diva Verification</td>
<td>572</td>
</tr>
<tr>
<td>Probe Command</td>
<td>573</td>
</tr>
<tr>
<td>Markers Command</td>
<td>578</td>
</tr>
<tr>
<td><strong>15</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Short Locator</strong></td>
<td>580</td>
</tr>
<tr>
<td>Shorts</td>
<td>580</td>
</tr>
<tr>
<td>Prerequisites</td>
<td>580</td>
</tr>
<tr>
<td>How Short Location Works</td>
<td>580</td>
</tr>
<tr>
<td>Short Locator Form</td>
<td>584</td>
</tr>
<tr>
<td>Using Shorts</td>
<td>585</td>
</tr>
<tr>
<td><strong>A</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Translating PDV Files to Diva Verification Files</strong></td>
<td>588</td>
</tr>
<tr>
<td>The pdvtodiva Command</td>
<td>588</td>
</tr>
<tr>
<td>pdvtodiva</td>
<td>589</td>
</tr>
<tr>
<td><strong>B</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Translating Dracula Files</strong></td>
<td>591</td>
</tr>
<tr>
<td>Dracula To Diva Verification Translator: DraculaToDiva</td>
<td>591</td>
</tr>
<tr>
<td>Running DraculaToDiva</td>
<td>591</td>
</tr>
<tr>
<td>What Happens During Translation?</td>
<td>592</td>
</tr>
<tr>
<td>Dracula To Diva Verification Command Correspondence</td>
<td>592</td>
</tr>
<tr>
<td>SQUARE Check</td>
<td>604</td>
</tr>
<tr>
<td>Checking Philosophy</td>
<td>604</td>
</tr>
<tr>
<td>Translation Pitfalls</td>
<td>606</td>
</tr>
<tr>
<td><strong>C</strong></td>
<td></td>
</tr>
<tr>
<td><strong>Improving Hierarchical DRC Performance</strong></td>
<td>608</td>
</tr>
<tr>
<td>Assigning the hdrc boundary Layer</td>
<td>608</td>
</tr>
<tr>
<td>How the hdrc boundary Layer Works</td>
<td>608</td>
</tr>
<tr>
<td>Hierarchical DRC Switch Names</td>
<td>609</td>
</tr>
<tr>
<td>Rules File Example</td>
<td>610</td>
</tr>
</tbody>
</table>
Diva Reference

D
Simulation and Environment Control .......................... 613
  Directories and Files ........................................ 613
  Variables ...................................................... 615
  Control Input .................................................. 616
  Setting Up the Netlister ...................................... 616

E
Application Techniques ........................................ 627
  Bipolar Processing ............................................. 627
    General Technique .......................................... 627
    npn Transistor ............................................... 631
    pnp Transistor ............................................... 632
  Resistor Terminal Recognition ............................... 635
  Extraction Device Library Entries ............................ 638
  Substrate, Background, and Ground ........................... 640
    Concepts ..................................................... 640
    Relationships ............................................... 640

Index ..................................................................... 642
Preface

This manual assumes that you are familiar with the development and design of integrated circuits. It contains reference information about the following Assura™ Diva® verification products which belong to the Assura family of physical verification products:

- Design Rule Checker (iDRC)
- Layout Parasitic Extractor (iLPE)
- Parasitic Resistance Extractor (iPRE)
- Electrical Rules Checker (iERC)
- Layout Versus Schematic program (iLVS)

In addition, the Assura Diva verification tool has a remote verification function (Assura™ Diva® remote verification) that lets you run jobs on a remote, central server. This renders the local stations free to perform other tasks.

The preface discusses the following:

- Related Documents
- Typographic and Syntax Conventions

Related Documents

The Assura Diva verification software is often used with other Cadence products to find and correct design errors. The following manuals give you more information about the tools associated with the Assura Diva verification software.

- To learn more about the Open Simulation System, read the *Open Simulation System Reference Manual*.
- To learn more information about editing and loading technology files, read the *Technology File and Display Resource File User Guide*.
- To learn more about parameterized cells, see *Virtuoso Layout Editor User Guide*.
- To learn more about other menus on the menu banner, read the *Virtuoso Layout Editor User Guide*. 
To learn more about other PDRACULA commands, read the *Dracula Standalone Verification Reference Manual*.

The *Inherited Connections Flow Guide* describes how to use inherited connections and net expressions with various Cadence® tools in the design flow.

The *Virtuoso Schematic Composer User Guide* describes connectivity and naming conventions for inherited connections and how to add and edit net expressions in a schematic or symbol cellview.

The *Cadence Installation Guide* tells you how to install the product.

### Typographic and Syntax Conventions

Here are some conventions used to describe menu commands.

Each form shows you the system defaults:

- Filled buttons are the default selections.
- Filled-in values are the default values.

This list describes the syntax conventions used for this product in this manual.

<table>
<thead>
<tr>
<th><strong>literal (LITERAL)</strong></th>
<th>Nonitalic (UPPERCASE) words indicate keywords that you must enter literally. These keywords represent command (function, routine) or option names.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>argument (z_argument)</strong></td>
<td>Words in italics indicate user-defined arguments for which you must substitute a name or a value. (The characters before the underscore (<em>) in the word indicate the data types that this argument can take. Names are case sensitive. Do not type the underscore (z</em>) before your arguments.)</td>
</tr>
<tr>
<td>**</td>
<td>Vertical bars (OR-bars) separate possible choices for a single argument. They take precedence over any other character.</td>
</tr>
<tr>
<td>**</td>
<td>Brackets denote optional arguments. When used with OR-bars, they enclose a list of choices. You can choose one argument from the list.</td>
</tr>
<tr>
<td>**</td>
<td>Braces are used with OR-bars and enclose a list of choices. You must choose one argument from the list.</td>
</tr>
</tbody>
</table>
Three dots (…) indicate that you can repeat the previous argument. If you use them with brackets, you can specify zero or more arguments. If they are used without brackets, you must specify at least one argument, but you can specify more.

argument...: specify at least one, but more are possible
[argument]...: you can specify zero or more

,...

A comma and three dots together indicate that if you specify more than one argument, you must separate those arguments by commas.

⇒

A right arrow precedes the possible values that a SKILL function can return. This character is created with a special symbol font in the publishing tool. It is represented in ASCII by an equal sign and a greater than sign (=>).

/

A slash separates the possible values that can be returned by a SKILL function.

Important
The language requires many characters not included in the list above. You must enter required characters literally.

SKILL Syntax Examples
The following examples show typical syntax characters used in SKILL.

Example 1
list (  
g_arg1  
[g_arg2] ...  
)  
⇒ l_result

Example 1 illustrates the following syntax characters:

list

Plain type indicates words that you must enter literally.

g_arg1

Words in italics indicate arguments that you must substitute a name or a value.
Parentheses separate names of functions from their arguments.

_ An underscore separates an argument type (left) from an argument name (right).

[ ] Brackets indicate that the enclosed argument is optional.

⇒ A right arrow points to the description of the return value of the function. Also used in code examples in SKILL manuals.

... Three dots indicate that the preceding item can appear any number of times.

Example 2

```
needNCells(
    s_cellType | st_userType
    x_cellCount
)
⇒ t/nil
```

Example 2 illustrates two additional syntax characters.

| Vertical bars separate a choice of required options.

/ Slashes separate possible return values.
Introducing Diva Verification

The Assura™ Diva® verification product is a set of verification tools belonging to the Assura family of physical verification products that let you find and correct design errors. Using layer processing to prepare data this set of verification tools checks physical design and electrical functionality, and performs layout versus schematic comparisons. This tool helps you find errors early in the design process and lets you view them interactively to help speed error diagnosis and correction. This product also allows you to perform incremental checks on areas that you change.

Diva Verification Tool Set

The Diva verification tool set has five interactive products:

- Design Rule Checker (iDRC)
- Layout Parasitic Extractor (iLPE)
- Parasitic Resistance Extractor (iPRE)
- Electrical Rules Checker (iERC)
- Layout Versus Schematic program (iLVSc)

iDRC

The Design Rule Checker, idRC, helps you find deviations from VLSI/ULSI design constraints. The DRC program works with any technology and handles all layout methodologies, including full custom, structured custom, standard cell, macro cell, gate array, and automated layout.

DRC's interactive tools give such broad design rule coverage that it almost eliminates false errors. You can use DRC to check material spacing, enclosure, and overlap. Its full hierarchical operation, incremental checking, and unique pattern-recognition capabilities reduce run times.
iLPE

The Layout Parasitic Extractor, iLPE, extracts physical devices. It can extract complex parameters and parasitics from most technologies. The LPE program operates in flat or hierarchical mode to produce a complete network database that can be used by other products, such as performance analyzers, simulators, and netlisters.

iPRE

The Parasitic Resistor Extractor, iPRE, extracts parasitic resistance and capacitance and converts the circuit interconnect into a complete resistor-capacitor (R-C) network. The PRE program handles all layout topologies, including bend and transition compensation, contact area and edge resistance, and user-defined coefficients. You can also select among various representations for the extracted R-C network.

iERC

The Electrical Rules Checker, iERC, checks network connectivity. The ERC program highlights electrical problems, such as floating interconnect and devices, and abnormal connections in physical or schematic designs. The ERC program uses networks generated from either the layout or schematic.

ERC performs conventional checks, such as verifying pull up/pull down and isolating inactive devices. It can also convert a MOS transistor-level network into a gate-level network with gate-level parameters. This lets the network be processed by gate-level simulators.

iLVS

The comparison product Layout Versus Schematic, iLVS, checks for matching nets, devices, and parameters in circuit networks. The LVS program compares a layout and schematic, two layouts, or two schematics. The LVS program has advanced capabilities, including full device permutability. To aid in error correction, you can view the results interactively using error probing and correspondence cross-probing.
Diva Verification Programs

The five interactive products are integrated into three programs that let you run verification checks and display errors graphically. The programs, the processes they perform, and the products they use, are shown below.

<table>
<thead>
<tr>
<th>Program</th>
<th>Process</th>
<th>Product</th>
</tr>
</thead>
<tbody>
<tr>
<td>DRC/Extract</td>
<td>Layer Processing</td>
<td>iDRC</td>
</tr>
<tr>
<td></td>
<td>Design Rule Checking</td>
<td>iDRC</td>
</tr>
<tr>
<td></td>
<td>Connectivity Extraction</td>
<td>iDRC</td>
</tr>
<tr>
<td></td>
<td>Device Extraction</td>
<td>iLPE</td>
</tr>
<tr>
<td></td>
<td>Parameter Extraction</td>
<td>iLPE</td>
</tr>
<tr>
<td></td>
<td>Parasitic Extraction</td>
<td>iLPE</td>
</tr>
<tr>
<td></td>
<td>Resistance Extraction</td>
<td>iPRE</td>
</tr>
<tr>
<td>ERC</td>
<td>Electrical Rule Checking</td>
<td>iERC</td>
</tr>
<tr>
<td>LVS</td>
<td>Layout Versus Schematic Comparison</td>
<td>iLVS</td>
</tr>
<tr>
<td></td>
<td>Correspondence Point Generation</td>
<td>iLVS</td>
</tr>
</tbody>
</table>

Diva Verification Product Flow

The Diva verification products are interrelated and, in some cases, dependent on each other. For example, to do an Electrical Rules Check (ERC) or a Layout Versus Schematic (LVS) comparison, you must first do an extraction on the layout. However, you can run other processes independently. You can do a Design Rule Check (DRC) or an extraction without reference to any other product. This is a typical verification sequence.

- Do a DRC to check the layout for physical design constraints and correct construction flaws.
- Do an extraction to prepare the layout design for ERC or LVS verification checks. You can also use Extract (iLPE) to prepare a circuit for post-layout simulation.
- Do an ERC to check basic electrical functions for circuit consistency. You can use ERC to check both layout and schematic data.
- Do an LVS to compare layout and schematic designs.
This figure shows the flow of products that are used for interactive verification.

Prerequisites for Diva Verification

To run this tool in interactive mode, you must have

- The Cadence® design framework II software
- The Open Simulation System (for ERC and LVS)
- A set of verification rules

To run this tool in batch mode from the UNIX prompt, you must have

- The Open Simulation System (for ERC and LVS)
A set of verification rules

**Design Framework II**

Cadence design framework II is an integrated design environment in which you can run the Cadence design tools. It has a unified database that provides a consistent way of storing design information, a way for tools to communicate with each other, and a way to distribute design data. You must have Cadence design framework II to run the Diva verification tool in interactive, graphics mode. For more information, refer to the *Design Framework II Help*.

**Simulation Environment**

The Open Simulation System (OSS) contains the netlisters and simulators supported by your system. The ERC and LVS programs both run under its control. When you start an ERC or LVS job, the OSS takes control and performs a number of operations. These include running the system netlister, if required, running the ERC or LVS program, and translating the output so that you can do probing and cross-probing. For more information about OSS, refer to the *Open Simulation System Reference Manual*.

**Verification Rule File**

Before you can run a verification check, you must define the rules for the routine you want to run: DRC, extraction, ERC, or LVS. You use a text editor to write these rules as a series of SKILL functions in a verification rules file.

You can store this file anywhere in your file system and reference it when you run Diva verification. A common location for these rules is in a technology library. By default, for DRC and extraction, this tool looks for the files `divaDRC.rul` and `divaEXT.rul` in the technology library. There is no default location for the LVS or ERC rules files. If you store the file anywhere in a technology library, you must include a dot in the file name to distinguish it from other library entries. If you use a standard convention of adding a `.rul` suffix to your file names, you can always identify the Diva verification rules files.

See Chapter 1, “Introduction” for more information about writing rules files.

**Using Diva Verification**

Mouse button assignments vary slightly from application to application. The *Design Framework II Help* describes button assignments for the CIW and other windows in the
Cadence design framework II environment. The information that follows describes how the mouse works in Diva verification.

**About the Mouse**

This manual uses the following terms to tell you how to use the mouse.

<table>
<thead>
<tr>
<th>Manual Term</th>
<th>What You Do</th>
</tr>
</thead>
<tbody>
<tr>
<td>Click</td>
<td>When you are asked to <em>click</em>, press and release the mouse button. Which button you press depends on the location of the pointer. You use the left button most of the time. Sometimes this manual uses the terms <em>click left</em>, <em>click middle</em>, and <em>click right</em>. If no button is mentioned, click the left button.</td>
</tr>
<tr>
<td>Double click</td>
<td>When you are asked to <em>double click</em>, click the specified mouse button twice, rapidly.</td>
</tr>
<tr>
<td>Select</td>
<td>When you are asked to <em>select</em>, usually you move the mouse pointer to an object and click the left mouse button. Sometimes you perform another action, such as holding the left button down, moving the mouse, then releasing the button.</td>
</tr>
<tr>
<td>Pop up</td>
<td>When you are asked to <em>pop up</em> a menu, press the middle mouse button.</td>
</tr>
<tr>
<td>Hold</td>
<td>Press and hold down the mouse button.</td>
</tr>
<tr>
<td>Drag</td>
<td>Hold down the mouse button while you move the mouse.</td>
</tr>
<tr>
<td>Draw through</td>
<td>Draw a rectangle by pressing the mouse button at one corner of the rectangle, moving to the opposite corner of the rectangle with the button held down, and releasing the button.</td>
</tr>
</tbody>
</table>
Using the Mouse
Buttons

The following figure shows how to use the mouse.

To choose commands
To select options on forms
To select objects in cells
To draw objects

To pop up menus
To pop up options forms by double clicking

To repeat a command
To delete a point in a cell

Using the Verify Menu

The Diva verification commands are located on the Verify menu at the top of either a layout or a schematic window. This illustration shows parts of the layout menu and the Verify menu at the top of a layout window.
To display the Verify menu, click on the name of the menu. To select a command from the Verify menu, move the pointer to the item you want and click the left mouse button. The item you select displays a form or another menu.

- When you click on an item followed by three dots (...), you see a form for choosing options before you execute the command.

- When you click on the Markers command, which is followed by an arrow (→), you see a “slider” menu with more commands. To select a command, you hold down the left mouse button and slide the pointer to the item you want, then release the button.

See Chapter 1, “Introduction” for more information about the Verify menu.

Using Forms

This section describes how to use the forms that appear as you use the Diva verification commands. For more details about using forms, see Design Framework II Help.

The buttons across the top of forms are suited to the work you are doing at the moment.

- “Standard” forms appear when you select a command followed by three dots (...).

- “Options” forms appear when you use a command that requires repeated input.

Standard Forms

Standard form values take effect when you execute the command by clicking OK or Apply. A standard form looks like this.
The buttons at the top of a standard form function as follows.

- **OK**: Executes the command and removes the form.
- **Cancel**: Removes the form without executing the command.
- **Defaults**: Displays default values for options on the form.
- **Apply**: Executes the command and keeps the form on the screen.
- **Help**: Provides information about the form.

Pressing Return has the same effect as clicking on **OK**.

**Options Forms**

The values you select or enter on options forms take effect immediately. For example, you can use the options form for the *Ellipse* command to change settings while you are drawing an ellipse.
The buttons on an options form are different from standard form buttons. Here’s how they work.

Pressing Return has the same effect as clicking on *Hide*.

**Displaying or Suppressing Options Forms**

When your company first receives the Cadence software, options forms display automatically. After you are comfortable using the Cadence software, you might not need to use the options forms. You can turn off this feature from the User Preferences form which controls how the Cadence software behaves.

You can turn off the options forms setting as follows.

1. Select *User Preferences* from the Set Options menu in the Command Interpreter Window (CIW).
   
   The User Preferences form appears.

2. Click the *Options Displayed When Commands Start* button.

3. Click *OK* to close the form.

**Removing and Retrieving Options Forms**

To remove options forms from the screen and retrieve them, use the *Hide* button.

1. To make an options form disappear temporarily, click on the *Hide* button. The options form disappears.

2. To redisplay the options form, double click the middle mouse button.
Running Diva Verification

You can run Diva verification tool from the Cadence or UNIX environment, or you can run this tool remotely (only if you have an Diva remote verification license).

- To use the Diva interactive graphics mode, you must work in the Cadence environment. When you do a DRC or extraction, this tool runs in the foreground. When you run an ERC or an LVS, the program runs in background mode and lets you do other work in the Cadence environment.

There are three ways you can run this product in the Cadence environment.

- Open a design window, and select the Verify menu commands.
- Open the Cadence design framework II, and type SKILL functions in the Command Interpreter Window (CIW). See Chapter 3, Executing Diva Verification Using SKILL Functions for more information on SKILL functions.
- Open Cadence design framework II in nongraphics (-nograph option) mode, and type SKILL functions at the prompt. See Chapter 3, Executing Diva Verification Using SKILL Functions for more information on SKILL functions.

During interactive DRC and extraction runs, the rules being executed are normally echoed in the CIW. There are two conditions that impact this display.

- Rule optimization changes the order in which rules are executed and forces multiple rules to combine into a single execution step. For combined rules, all the combined rules echo as components of the single execution step.
- If you run in hierarchical mode, the execution of the rules is not echoed in the CIW. The complete rule set is executed for every cell being processed, and an echo of this in the CIW and in the run log generates unacceptably large quantities of repetitive data.

- When you use the UNIX environment, the Diva verification tool is not interactive. You do not need a graphics terminal. Errors are listed in the standard output file displayed on the screen. The results of the run are saved in a new version of the design. You can graphically display the new version and the errors by running interactive Diva verification. This means you can run a verification check in
background mode while using Cadence software on another design, and then later display the results of the check. See Chapter 3, Executing Diva Verification Using SKILL Functions for more information on UNIX functions.

■ When you run Diva® remote verification, you can run verification on one design while you work on another. You can run verification on a remote server or workstation, or on the local machine as a batch job. When the server completes the remote verification, you can display the results on the local workstation and correct your design.
Interactive Mode

Running this tool interactively lets you run a verification check and then display and probe errors on the screen. This graphics capability can significantly reduce the time you spend finding and correcting errors.

You can run this product interactively in one of the following ways:

■ Open a design window in the Cadence design framework II environment and select the Verify menu commands.

■ Open the Cadence design framework II environment and type SKILL functions in the CIW.

Using Verify Menu Commands

The Cadence software displays designs in layout or schematic windows. You can access Diva verification commands from the Verify menu at the top of either a layout or a schematic window. For information about other menus on the menu banner, refer to Virtuoso Layout Editor User Guide. For information about using the Verify menu, forms, and options windows, see Chapter 1, “Introducing Diva Verification.”

Using Commands from a Layout Window

To access this tool’s commands from a layout window, perform these steps.

1. Select Verify from the menu banner.
2. Select the Diva verification command you want to use. A brief description of each command follows the menu. For a detailed description of each command, see Chapter 14, “Verify Menu Commands.”

<table>
<thead>
<tr>
<th>Command</th>
</tr>
</thead>
<tbody>
<tr>
<td>MSPS Check Pins</td>
</tr>
<tr>
<td>DRC</td>
</tr>
<tr>
<td>Extract</td>
</tr>
<tr>
<td>Substrate Coupling Analysis</td>
</tr>
<tr>
<td>ConclCe</td>
</tr>
<tr>
<td>ERC</td>
</tr>
<tr>
<td>LVS</td>
</tr>
<tr>
<td>Shorts</td>
</tr>
<tr>
<td>Probe</td>
</tr>
<tr>
<td>Markers</td>
</tr>
<tr>
<td>Explain</td>
</tr>
<tr>
<td>Find</td>
</tr>
<tr>
<td>Delete</td>
</tr>
<tr>
<td>Delete All</td>
</tr>
</tbody>
</table>

**DRC** checks a layout for design rule violations. DRC puts any errors found on the marker layer. You can highlight and display information about errors using the **Markers** commands.

**Extract** extracts devices and connectivity to prepare a layout for ERC and LVS verification checks. Extract puts any errors found on the marker layer. You can highlight and display information about errors using the **Markers** commands.

**ERC** checks an extracted cellview of a layout or schematic for electrical violations. ERC lets you highlight and display information about any errors found.

**LVS** compares two versions of a circuit. You can compare two layouts, two schematics, or a layout and a schematic. You can highlight and display information about any differences found.

**Probe** highlights and displays information about devices and nets in a layout following an ERC or LVS run. Single-probing lets you highlight a net or device in a single cellview. Cross-probing lets you highlight a net or device in two different cellviews that have been compared using LVS.

**Markers** displays the Markers menu, which contains commands that let you find error or warning markers in a layout.

**Explain** displays a window showing the reason for the error or warning marker.
**Find** searches for and highlights each error and warning marker.

**Delete** permanently removes an error or warning marker.

**Delete All** permanently removes all error and warning markers for the current cellview or the complete hierarchy.

**Using Commands from a Schematic Window**

To access this software’s commands from a schematic window, perform the following steps.

1. Select **Tools - Diva** from the banner menu.
   - The menu **Verify** gets added to the menu banner.
2. Select **Verify** from the banner menu.

Select the Diva verification command you want to use from the **Verify** menu. A brief description of each command follows the menu. For a detailed description of each command, see Chapter 14, “Verify Menu Commands.”

**ERC** checks an extracted cellview of a schematic or layout for electrical violations. ERC lets you highlight and display information about any errors found.

**LVS** compares two versions of a circuit. You can compare two layouts, two schematics, or a layout and a schematic. You can highlight and display information about any differences found.

**Probe** highlights and displays information about devices and nets in a layout following an ERC or LVS run. Single-probing lets you highlight a net or device in a single cellview. Cross-probing lets you highlight a net or device in two different cellviews that have been compared using LVS.

**Using SKILL Functions**

In addition to using the pulldown menus, you can type SKILL functions in the CIW. For detailed descriptions, syntax, and examples of these commands, see Chapter 3, Executing Diva Verification Using SKILL Functions.
Batch Mode

Running Diva verification in batch mode lets you run a verification check either in another window or on another machine while you continue to work on a different application. When you run this tool in batch mode, you do not need a graphics terminal. You can run a verification check on a machine without graphics capability and then later display and probe the errors on a machine with graphics capability.

You can run this product in batch mode by using either the Cadence design framework II environment or the UNIX environment.

- Open the Cadence design Framework II environment in the nongraphics (no-graph option) mode and execute the SKILL functions with the necessary arguments. For detailed descriptions, syntax, and examples of these commands, see Chapter 3, Executing Diva Verification Using SKILL Functions.

- Type `ivVerify` at the UNIX prompt. For a detailed description, syntax, and examples of this command, see the `ivVerify` section in Chapter 3.
Remote Mode

With Diva remote verification, you can run verification on one design while you work on a different machine. You can run verification on a remote server or workstation, or on the local machine as a batch job. When the server completes the remote verification, you can display the results on your local workstation and correct the design.

When you run a job on a remote workstation, Diva remote verification uses the NFS file system to access the data you are working with on the local machine or any other machine.

To run this tool remotely

1. Open a design window in the Cadence design framework II environment.
2. Select the Verify menu command DRC, Extract, ERC, or LVS.
3. Select the remote option for Machine and enter the machine name in the field provided. A blank entry in this field tells the Diva verification tool to use the local machine.

Prerequisites

Before running Diva remote verification, you must

- Have this software (DRC, Extract, ERC, or LVS) and licenses available on the remote machine.
- Set up the local and remote machines to run this remote tool as described in the following section.

Setting Up Machines for Diva Remote Verification

After the Cadence tools and licenses are installed on the remote and local machines, you must set up the machines to run Diva remote verification by performing the following steps.

1. Modify the rhosts file on the local machine to gain access to the remote machine.
2. Make sure you have not defined restricted access to your local machine by using an xhost command in your initialization file. The remote machine must be able to talk to the local machine.
3. Note whether your remote machine setup causes messages to be returned when you use rsh. For example, an entry in the .cshrc of the remote machine stty erase ^H causes a message to be echoed when the rsh command executes that .cshrc. This echo might cause a problem in some cases.
4. Using *mount* or *automount*, make the data you are working with on the local machine accessible in read and write mode from the remote machine.

5. Check the access you have to the software on the remote machine. You can do this by typing the following command on the local machine:

   ```
   rsh server ivVerify -V
   ```

   This should return with a line of the form

   ```
   ivVerify version 4.4 Thu Nov 17 11:04:26 PST 194 (cds9448)
   ```

   On some machines, you need to use *remsh* instead of *rsh*.

   Check the version number to ensure you are using the correct program. This is the version number of the Diva verification software that is running on the remote machine. If the version number is incorrect, you need to change the .cshrc file on the server.

   If *rsh* fails and prints “Permission Denied,” you do not have permission to run this program on the remote machine. See your local system administrator to correct this. You might need to update the .rhosts file on the server.

   If the *rsh* returns with the message “ivVerify: Command not found” the .cshrc file on the server does not define the path containing this tool.

6. You must decide whether you want to use manual or automatic file mapping. If you choose manual file mapping, you need to set up a map file. See the following section on “File Mapping” for more information.

7. To improve the performance of the remote processing, try to keep network traffic to a minimum. The *ivVerify* program creates temporary files of the form 1234.pdv. Set up your .cshrc file on the remote machine so these temporary files remain on the server. You can do this by setting the DRCTEMPDIR environment variable. In the .cshrc file add the line:

   ```
   setenv DRCTEMPDIR working_directory_on_server
   ```

   This eases network traffic and lets *ivVerify* run faster. When you set this variable, you must take into account the size capabilities of the remote machine.

### File Mapping

The Diva remote verification program needs to access data from machines other than the server it is running on. Therefore, the file paths on the machines must be mapped.

File mapping is the process of determining the name of a file as it is seen from the server. For example, a file on the local machine called `/usr/ucb/vi` might be referenced from the server with the name `/net/ziggy/usr/ucb/vi`. 
There are two ways to do this: manually and automatically.

- The manual methodology requires you to create a mapping file, but is independent of the automount methodology or the complexity of the automount configuration file.

- The automatic methodology requires no special setup, but has a small start-up run-time overhead. Because the automount methodology relies on simple automount usage, it might not work with AFS or on Apollo, and it might not work with complex automounts.

If this remote tool interface program does not find a manual mapping file on the local machine, this software runs in automatic mode.

See the following sections for more information about manual and automatic file mapping.

Manual File Mapping

Manual mapping is used when the Diva verification tool finds a map file. The map file must be in one of the following locations on the host machine:

- Current directory
- Home directory
- `<your_install_dir>/tools/dfII/local/divamaps`
  
  (Replace `your_install_dir` with the location your Cadence software was installed in.)

The map file name must be of the form:

```
diva.local.host.map
```

where

- `local` is the name of the machine you are running on.
- `host` is the target machine for Diva remote verification execution (server).

The machine name used is the canonicalized machine name returned when you type the SKILL function `ivGetHostName` (machine) in the CIW.

Map files have a two-column format that contains directory definitions. The columns can be separated by spaces or tabs, and each line consists of one map entry.

The first column in the map file is the name of the directory as it appears when referenced from the local machine. The second column is the name of the directory as it appears from the host (server) machine. The directories can reside on any machine. For example,
Map file name:
```
diva.ice.snow.map
```

Map file contents:
```
/               /net/ice
/usr            /net/ice/usr
/net/snow       /
/net/snow/usr   /usr
/net            /net
```

The first two entries refer to directories on the local machine (*ice*) so the first column has direct references and the second column uses automount addresses.

Conversely, the next two directories reside on the server machine (*snow*) so the second column has direct addresses and the first column uses automount.

The last entry allows any other machine to be accessed through automount. The address of other machines is the same from both the local and server machines.

The map file is sorted by length of name of the local directory so that longer paths match before shorter paths. The first entry that matches determines the mapped name.

**Automatic File Mapping**

Automatic file mapping uses two daemons, *cdsd* and *remoteDivad*. The *cdsd* daemon must be running on all computers that have design data or run Cadence tools. The *remoteDivad* daemon is started automatically by the Cadence graphic editor.

Once the *cdsd* daemon is started, it runs forever if not manually terminated. The *remoteDivad* daemon terminates when you exit the graphics editor. If you change servers, the daemon is terminated on the original server and started on the new server. The *remoteDivad* daemon is not used if you choose the local machine as the server.

The first time file mapping is required for a given server there is a slight delay as the *remoteDivad* daemon is started on the remote server. Network problems might increase this delay. Subsequent mapping is not delayed.

**Limitations**

The *remoteDivad* daemon scans through the `/etc/fstab` file on the server to determine what a file name should map to. It can also work with automount. However, it might be confused if you use automount with complex configuration files. To ensure the daemon works correctly,
keep your mounted NFS files mounted with either the mount command, such as defined by the `/etc/fstab/` file, or with simple automount, such as `automount /net -hosts`.

## Controlling Diva Verification

Diva verification lets you control certain aspects of the verification run. This section explains different commands you can use to

- Override run modes on a cell-by-cell basis
- Create a log file of all marker shapes created during the Diva verification runs
- Save the status of a verification run at any point
- Execute DRC and Extraction rules under conditions you specify
- Control SKILL access to the DFII database
- Set environmental variables
- Specify where this tool stores temporary files
- Control what cells are referenced in your circuit hierarchy
- Convert cells in the hierarchy into instances of devices
Controlling Run Modes Using Properties

For the ivDRC and ivVerify commands, the run modes can be overridden on a cell-by-cell basis by adding a property to the cell. The property overrides the option on the command line for the cell containing the property and all cells below it in the hierarchy.

Syntax

hdrc = mode

Fields

hdrc

The property name.

mode

The value of the property enclosed in double quotation marks (" "). The value can be any one of these single characters, or a string starting with that character:

F

Flat. Equivalent to the ?hier nil option.

N

Nonoptimized hierarchical. Equivalent to ?hier t plus ?optimize nil.

E

Optimized hierarchical with expanded mosaics. Equivalent to ?hier plus ?optimize, except mosaics are expanded.

U


References to the expansion of mosaics identifies how they are processed in the pattern recognition optimization. In expanded mode, the mosaic planes are treated as contiguous data without interplane boundaries or repetition. In unexpanded mode, the interfaces between the planes are optimized the same as interfaces between cells.

Example

In this example, the cell is checked hierarchically with optimization, while mosaics remain unexpanded.

    hdrc = "U"
If you leave the inclusion limit to its default of 1,000, all cells are processed. For more information on specifying the inclusion limit, see the `?inc` argument in the `ivDRC`, `ivExtract`, or `ivVerify` command.
Creating a Log File

The `drcCreateLogFile` command lets you create a log file that lists all marker shapes created during an Diva verification run. You can use the coordinates in this list to reference the errors and then cross them off the list once you have fixed them. You can use the `drcCreateLogFile` command in the CIW or in a SKILL program.

Syntax

```
drcCreateLogFile( topCellId [keyed arguments] )
```

Fields

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>topCellId</code></td>
<td>The ID of the top-level cell. You get the top cell ID by typing the command <code>cv=dbOpenCellViewByType( &quot;lib&quot; &quot;cell&quot; &quot;view&quot; )</code> in the CIW.</td>
</tr>
<tr>
<td><code>keyed arguments</code></td>
<td>Options used for keyed arguments. You can specify the following options:</td>
</tr>
<tr>
<td><code>?name</code></td>
<td>Name of the log file. The existing data in this file is overwritten when any new text is written to it. Diva verification writes the log file to the CIW by default.</td>
</tr>
<tr>
<td><code>?key</code></td>
<td>The name of the property to filter, such as “drcWhy” or “extractWhy.” The default is no filtering.</td>
</tr>
<tr>
<td><code>?hier</code></td>
<td>If <code>t</code>, a full hierarchical summary is generated. Otherwise, only a log file for the top-level cell is generated.</td>
</tr>
<tr>
<td><code>?precision</code></td>
<td>Number of digits to the right of the decimal point to be used when the numbers are printed. For example, a value of 2 might cause coordinates to be printed such as 5.25, while a value of 3 might produce 5.252. The default value is 2.</td>
</tr>
<tr>
<td><code>?page_width</code></td>
<td>The default page width is 80 characters. Lines that exceed 80 characters are broken into multiple lines.</td>
</tr>
<tr>
<td><code>?page_length</code></td>
<td>The default page width is 0, which disables this feature. If you use this argument, the system inserts a page header at the top of each page.</td>
</tr>
</tbody>
</table>
?line_limit

Specifies the limit to the number of lines output to the log file. After this indicated number of lines are output to the log, no more will be output. The default is no limit.

Example

drcCreateLogFile( )
Using Checkpoints

You can use the `checkPoint` command to save the status of a verification run at any point in its execution. You then use this run status to start another run from the final checkpoint, rather than recreating data from the beginning.

To create checkpoints, you enter the `checkPoint` command in the verification rules file. The `checkPoint` command creates a directory, if it does not already exist, and stores run data in that directory. You specify the directory name with the `checkPoint` command.

You can use multiple `checkPoint` commands in the rules file. You can insert as many checkpoint commands as you want. At each checkpoint, the data resulting from the previous checkpoint is overwritten by data from the most recent checkpoint. Only the last checkpoint data is saved. If the run finishes successfully, all the stored data is removed.

Adding a checkpoint at the beginning or end of the rules has no effect. At the beginning, this tool has not yet created any data so there is nothing it can checkpoint. At the end, the Diva verification is complete and results are saved, so there is no data to save.

⚠️ **Caution**

Because Diva verification optimizes command execution, there is not always a one-to-one correlation between command lines and functions performed. A command after a checkpoint might be combined with, and executed with, a command prior to the same checkpoint. This means you cannot be sure which commands are processed before or after any checkpoint.

The checkpoint process creates many files in the checkpoint directory. These files are used when you restart a run. Do not alter the checkpoint files or use the checkpoint directories to store other files, or you might jeopardize the integrity of the run.

If you are doing an extraction, or the DRC checks generate errors, the `checkPoint` command creates a library called `cpLib` in the directory you specify.

You cannot use the `checkPoint` command in hierarchical processing.

To restart the program, you use the `ivRestart` command. See the “`ivVerify`” on page 89 for more information about the `ivRestart` command.

**Syntax**

```plaintext
checkPoint( name )
```
Arguments

name

Defines the directory under which the checkpoint files are created. If this text string (in quotation marks) matches an environmental variable, the contents of that variable are used as the string.

You can enter a full or relative path to the checkpoint directory.

If the named directory exists, it is used along with any existing files. If it does not exist, it is created. If for any reason the directory cannot be created, the run terminates with a warning message.

Examples

The following example creates the directory my_checkpoint for the temporary layers:

    checkPoint( "my_checkpoint" )

In the following example, the first command in the UNIX environment sets the environmental variable CHECK_DIR to the value of my_checkpoint. The second command in the verification rules file creates a directory that references the CHECK_DIR environmental variable.

    setenv CHECK_DIR my_checkpoint
    checkPoint( "CHECK_DIR" )
Conditional Execution of DRC and Extraction Rules (ivIf)

You can choose whether to use DRC extraction rules when you compile the rules. After grouping the rules, you use the if-then-else capability of the ivIf command to define the conditions under which each group is run.

You can define the condition as a “switch” name or logical combination of switches. A switch name can be any name you choose or the default switch names like drc? and extract?. If you choose your own switch name, you must specify that name at run time. If you use only the default names, you do not need to specify a switch name at run time.

You can specify multiple geomConnect commands in the rules file, but only one can be active at a time. This software prints an error message if two or more geomConnect commands become active due to switch settings.

Syntax

ivIf(
    condition ...  
    then
    then_rules ... 
[else 
    else_rules ...])

Description

Divides the verification rules into groups and makes the execution of those groups dependent on conditions determined at rule compilation time.

When this tool evaluates the if condition, if the result is t (true), the then group of rules is compiled. If the result is nil (false), the else group of rules is compiled.

If there is only one rule in each group, the then and else keywords are unnecessary.

Caution

You should not use derived layers with ivIf(empty). You should only use raw graphical layers.
Arguments

**condition**

Determines which of the subsequent groups of rules is compiled. The result of the condition is always t (true) or nil (false).

The condition can be a single expression or a complex combination of expressions. You can use these variables for your condition.

```
switch( t_switch_name )
```

Tests for the switch name you define. You must enclose the switch name in quotation marks.

```
empty( t_layer_name )
```

Tests for the absence of data on the specified original graphics layer. Derived layers are not allowed since the value of the function is determined when the rules are compiled.

The use of derived layers is no longer supported due to changes in the task execution engine which preclude run time evaluation of the function.

```
&&
```

Combines condition functions in an AND operation. You can concatenate any number of functions. For example

```
switch( "A" ) && switch( "B" )
```

If all functions return t, the result is t. If one of the functions returns nil, the result is nil.

```
||
```

Combines condition functions in an OR operation. You can concatenate any number of functions. For example,

```
switch( "A" ) || switch( "B" ) || switch( "C" )
```

If either function returns t, then the final result is t. If both of the functions return nil, then the result is nil.

```
!
```

Reverses the meaning of one of the functions or concatenated group of functions. If you place “!” in front of the function, it behaves as a NOT operator. For example, if the function is `switch`, the `!` variable reverses the meaning to `not_switch`. 
To negate a concatenated group of operations, you must enclose them in parentheses with the operator preceding the opening parenthesis. For example,

```plaintext
!(switch( "A" ) || switch( "B" ))
```

The following examples show how you can combine the functions and operators.

```plaintext
( switch( "A" ) && switch( "B" ) ) || !switch( "C" )
```

```plaintext
( switch( "A" ) && switch( "B" ) ) ||
( switch( "C" )
 && switch( "D" ))
```

**Note:** Use parentheses to clarify the meaning of groups of concatenated expressions.

- **then**
  Optional keyword for the group of expressions to be executed if the result of the `if` condition is `t`. If only one expression in the group is to be executed, you can omit the keyword.

- **then_rules**
  Commands to be executed if the result of the `if` condition is `t`. These can be verification commands or SKILL functions.

- **else**
  Optional keyword for the group of expressions to be executed if the result of the `if` condition is `nil`. If no expressions are to be executed, you can omit the `then` keyword. If both the `then` and `else` groups have only one expression each, you can omit the `then` keyword.

- **else_expression**
  Commands to be executed if the result of the `if` condition is `nil`. These can be verification commands or SKILL functions.

**Examples**

The following are some examples of the command.

```plaintext
ivIf( switch( "do_drc" )
then
drc( gate "cut" sep < 3 )
drc( "poly" width < 2 )
)
```
If the switch `do_drc` is set, the two `drc` commands run. All subsequent commands run no matter how the switch is evaluated.

```plaintext
ivIf( !empty( "poly" ) && !empty( "diff" )
gate = geomAnd( "poly" "diff" )
)
```

If the original graphics layers `poly` and `diff` contain data, the subsequent layer processing command runs.

```plaintext
ivIf( switch ( "do_simple" )
then
length = measureParameter( length( gate inside poly )
0.5 )
else
temp = measureParameter( length( gate inside poly )
0.47 )
width = measureParameter( length( gate coincident poly
;0.52 )
bends = measureParameter( bends_all( gate inside poly
)
length = calculateParameter( temp - ( width * bends *
0.5 )
)
```

If `do_simple` is set, the one-line measurement method is used. If `do_simple` is not set, a more complex form of measurement that considers bends is used.

If you use the `ivIf` command and switches, you can have more than one `geomConnect` command in a rules file, as shown in the following example:

```plaintext
drcExtractRules(
 /* layer definitions go here */
ivIf( switch("EXTRACT") then
geomConnect /* connect for extract */
    via( cut poly1 diff metall1 )
    via( ncontact ntub diff )
    via( pcontact ptub diff )
    label( "polylabel" poly1 )
    label( "metallabel" metall1 )
)
extractDevice( ngate (poly1 "G") (diff "S" "D"
(ptub "B") "nfet ivcell" )
extractDevice( pgate (poly1 "C") (diff "S" "D"
(ntub "B") "pfet ivcell" )
saveInterconnect( poly1 diff metall1 cut ncontact
    pcontact ntub )
```
When you run a job, you can use only one switch containing a `geomConnect` command. The Diva verification tool issues an error message if you specify more than one switch containing a `geomConnect` command.
Default ivlf Switch Values

The Diva verification tool has a number of predefined switch names you can use with the ivlf command.

DRC and Extraction

You can switch the DRC and Extraction sections of the rules file automatically using the default switches drc? and extract?.

- **drc?**
  - When you run DRC, the drc? switch is automatically enabled.
  - The switch extract? is de-activated.

- **extract?**
  - When you run Extraction, the extract? switch is automatically enabled. The switch drc? is de-activated.

**Note:** The use of these switches is not necessary because the DRC and Extraction rules are now stored in separate files by default. The switches are still supported for cases where you select a nondefault rules file and want to keep both sets of rules in the same file.

Hierarchy Control

IDRC has three switches that are set during certain phases of execution. These switches let IDRC perform a specific class of design rule checks.

- **hier?**
  - This switch lets you use a different set of rules in hierarchical mode than in flat mode. In hierarchical mode, the hier? switch is automatically on.

- **currentCell?**
  - This feature lets you write rules that are cell based and not truly hierarchical. The currentCell? switch is toggled on and off as the design is checked. In hierarchical mode, checking occurs in two passes. The first pass uses only the polygons in the current cell. The second pass runs in flat mode, from the current cell down, rechecking small areas where two instances overlap or where an instance and a polygon overlap in the current cell. The currentCell? switch is set on in the first pass and set off in the second pass.

- **topCell?**
  - This switch, when present, instructs this tool to perform a flat DRC analysis on the top cell after all hierarchical checking is done. You can use this feature to check sparse data, for example pads, to improve execution speed.
In the final pass, you can run a rules check in flat mode on data that is best checked in flat mode. For example, pad checks have limited data and do not check efficiently in hierarchical mode.

Rules File Example

To understand how the switches can be used, consider four simple checks on a design.

- Flag any metal spacing less than 0.5 microns.
- Flag any pad spacing that is less than 10 microns. The pad layer is a sparse layer and is better checked in flat mode. This check can be done in hierarchical mode, but it increases the size of the halo, which makes the program run slow even when there is no pad layer in the vicinity.
- Check that any cell containing layer Q also contains layer R and that layer R is exactly 0.4 micron larger than layer Q. This can be checked by using the `currentCell?` switch.
- Check that no part of the design is more than 1 micron from a piece of metal. This check is easily done in flat mode.

**Note:** This check is the most difficult of the four to implement.

```plaintext
mlg = geomSize( "m1" 0.5 )
err = geomAndNot( geomBkgnd() mlg )
saveDerived( err "m1 coverage error" )
```

The problem in hierarchical mode is that instances form an area where no polygonal data exists and a large error is generated. It is necessary to find errors that are not over instances, yet errors must also be found that are between instance polygons and cell polygons.

This tool solves this problem by introducing a new layer in the data base, `m1 boundary`. As the design is processed, a boundary shape for `m1` is created. The check is done by looking at shapes at the current level and `m1` boundaries from instances that have already been processed.

The rules for the check follow. The rules apply only to hierarchical mode.

```plaintext
drcExtractRules(
    ivIf( switch( "drc?" ) then
        ivIf( switch( "hier?" ) then
            ivIf( switch( "topCell?" ) then
                ; check 2
                pad = geomOr( "pad" )
                drc( pad sep < 10 "pad spacing less than 10" )
            else
```
m1 = geomOr( "m1" )
; check 1
drc( m1 sep < 0.5 "m1 spacing less than 0.5" )

ivIf( switch( "currentCell?" ) then
; check 3
check3 = geomXor( geomSize( "Q" 0.4 ) "R" )
saveDerived( check3 "R and Q check" )
; check 4
; Remove any m1 boundary shapes because they are
; going to be rederived
geomErase( "m1" "boundary" )
; Collect m1 in the current cell with m1 boundary
; polygons from one level down in the hierarchy.
mlb = geomOr( "m1" geomGetPurpose( "m1" "boundary" 1 1 ) )
; Grow the m1 shapes by 0.5 microns to find areas
; that are not covered.
mlbg = geomSize( mlb 0.5 )
; Get the cell boundary to
; (1) determine where a coverage error should be
flagged.
; This gets rid of errors on the boundary of the cells.
; (2) build the m1 boundary for the current cell.
; The hdrc boundary shapes were calculated just prior
; to rules execution in the current cell.
cell_bnd = geomGetPurpose( "hdrc" "boundary" 0 0 )
cell_bnd_shrunk = geomSize( cell_bnd -0.5 )
check4 = geomAndNot( cell_bnd_shrunk mlbg )
saveDerived( check4 "m1 coverage error" )
; Derive the m1 boundary for the next level of
hierarchy.
ml_bnd_next = geomOr( "m1" cell_bnd_shrunk )
saveDerived( ml_bnd_next ( "m1" "boundary" ) )
)
)
else
/*
 * Rules to run in flat mode.
*/
ml = geomOr( "m1" )
drc( ml sep < 0.5 )
/*
   ... other flat checks ...
*/

)
Controlling SKILL Access

You can have SKILL access to the DFII database during Diva verification execution prior to saving the data to disk, which provides a simple interface and alleviates the read-write overhead. The *ivCallProc* command provides access to the DFII database only. There is no SKILL access to the this tool's internal data structures or edge files.

**Caution**

*You can destroy or corrupt data by using SKILL access to the database. The SKILL Access command does not determine whether a requested change is detrimental to the database. Therefore, you must make sure your changes do not harm the database.*

During hierarchical extraction, SKILL Access is applied for each pass over the rules, which corresponds to each cell being processed.

**Syntax**

```
ivCallProc( function )
```

**Description**

You can place this command anywhere in the rules file. SKILL Access is applied during extraction and abstract generation (abgen), and is ignored during DRC. Depending on where you place this command, it has the following results:

- If you place the command at the beginning of the rules file, the extracted view is available for processing but is empty because the tool has not done any work yet. You can add cell properties at this point.
- If you place the command after an *extractDevice* command in the rules file, the extracted view contains the instances of the devices. The devices are connected, but they do not have parameters such as length and width attached.
- If you place the command after the *saveParameter* command in the rules file, the device instances have properties attached.
- If you place the command after *saveParasitic* in the rules file, the extracted cell contains parasitic devices.
- If you place the command at the end of the rules file, the extracted cell is complete except for the *schematicLastExtracted* property.
Arguments

function

The name of the SKILL lambda function you specify.

The Diva verification tool passes a disembodied property list to the function. This property list contains the cell environment at the point in the execution of commands (rules file) where the ivCallProc command is referenced. There are four possible property values, which are database IDs of cellviews being processed or created. For hierarchical processing they refer to the current cell.

The property values are

layout The id of the cell being extracted. This is usually the layout view.

extracted The id of the extracted cell. It is nil when extraction is not active.

excell The id of the excell cell defined during hierarchical extraction. It is nil during flat extraction.

abstract The id of the abstract cell valid when abgen is active. If abgen is not active, the abstract is nil.

Examples

The following example places the name of the user who created the extracted cell on the extracted cell.

ivCallProc( lambda( (ids)
let( ( myName )
if( ids->extracted then
myName = getShellEnvVar( "LOGNAME" )
dbReplaceProp( ids->extracted "creator" "string" myName
)
)
)
)

The following example adds a property to each nfet in the extracted cell.

x = lambda( (ids)
let( (nfet nfets q)
if( ids->extracted then
nfets = setof(q ids->extracted~>instances q~>cellName == "nfet")
foreach( nfet nfets
    dbReplaceProp( nfet "myProp" "int" 3 )
)
)
The following example adds a property to each `nfile` in the extracted cell with a procedure.

```plaintext
procedure( addNfetProp(ids)
  let( (nfet nfets q)
    if( ids->extracted then
      nfets = setof(q ids->extracted->instances q->cellName == "nfile")
      foreach( nfet nfets
        dbReplaceProp( nfet "myProp" "int" 3 )
      )
    )
  )
)
ivCallProc( getd( 'addNfetProp) )
```
Special Functions

Syntax

ivGetCellView ()

Description

From the rules file, this function gets the database ID of the layout cellview being processed, from which the cell, library, and techfile can be accessed. Properties associated with these are available for use in the rules.

When this function is called, the optimized rules are marked as not being reusable. Subsequent runs of Diva verification will compile and optimize the rules, even if they have not been changed. This is necessary because the rules compiler cannot detect that a property used as a parameter in the rules has been changed.

Syntax

ivCompilingFor (string...)

Description

From the rules file, this function returns 1 if any of the strings passed as arguments contain the constant diva. It is expected that the results of calling the function will be used in a SKILL if statement. You can use the SKILL statement to select the Diva rules in a rule deck containing both Diva and Assura rules.

Example

The following example shows the use of this function to implement a check in a rule deck which supports both Diva and Assura.

if(ivCompilingFor("diva")
then a=geomSize(geomSize(m1 -0.5) 0.5)
else a=geomWidth(m1 keep > 1.0)
)
Controlling Properties and Variables

When you use the DRC and extraction tools, you can use environmental variables to control the behavior of various aspects of the program.

To set up the environmental variables automatically when you log in, place them in your .cshrc or .login file under csh or in your .profile file under sh.
Storing Temporary Files

By default, any temporary layer files that this tool generates are placed in the directory in which you started this verification.

If you want this tool to store temporary files in other directories, use the DRCTEMPDIR environmental variable. The directories you specify need not be on the same disk as the Diva verification tool. Each directory can be on a different disk.

To specify where you want to store the files, define the environmental variable as follows:

```
setenv DRCTEMPDIR "directory"
setenv DRCTEMPDIR "directory1 directory2 directory3 ... "
```

Temporary layer files created by this tool are named according to this format:

```
machid_pid_lyrid_fileno.pdv
```

- **machid**: The machine host name
- **pid**: The process id
- **lyrid**: The layer id
- **fileno**: The file number
- **pdv**: The layer file extension

The following is an example of a temporary layer filename:

```
machineA_101_20_1.pdv
```

The maximum size of any single temporary layer file created by this tool is 256 megabytes. Layers that exceed this file size limit are split into multiple files. The first file has a file number 1, the second file has a file number 2, and so on. As an example, a 3-gigabyte layer resides in 12 files.

When one or more disks is filled, this tool continues to allocate files to the remaining directories. If a file is too large for a directory, that file is automatically split, and the remainder of the file is placed in the next directory or disk with free space. If this tool fills all the disks that contain directories in the DRCTEMPDIR list, and there are still more files to store, the Diva verification terminates and the files are removed.
Including and Excluding Cells

By default, Diva verification processes all cells in the hierarchy of a circuit. To specify that a cell be ignored during the processing, you provide a non-negative integer value using one or more of the following methods:

- Provide a user-defined SKILL function to dynamically generate a value.
- Define a value with an `ivIncludeValue` property placed on the cell instance.
- Define a value with an `ivIncludeValue` property placed on the cell master.

This tool evaluates these methods in the order given until one provides a valid value that is then compared to the Include Limit specified when the run was started. During hierarchical processing, only method 3 is available because the instances are not present at the time the master is processed.

The user can define the SKILL function `ivUserIncludeValueFilter` to dynamically generate the value used in place of the `ivIncludeValue` property. This function can be defined in the run specific file. This function is called with the full instance path through the hierarchy to the instance currently being tested. The function can return the value to use for `ivIncludeValue` or an empty string to tell this tool it does not know the value and the rest of the above list is to be evaluated. A sample of this function is shown next.

```skill
; function to emulate the ivIncludeValue instance
; property using the
; user defined property myIncludeValue
procedure( ivUserIncludeValueFilter( instPath )
    let( ( instId result )

        ; get the leaf instance from the path
        instId = car( instPath )

        ; check for a mosaic stored as ( instId row column )
        when( listp( instId ) instId = car( instId ) )

        ; check if we know what the value should be
        result = cond(
            ( instId->myIncludeValue instId >myIncludeValue )
            ( t "" ) ;indicate that we don’t know the value
        ); cond

        ; send the answer back
        result
    )
)
```
When you start this tool, you can define an inclusion value as shown in the table below. For every cell to be processed, this tool compares this inclusion value with the value of the \textit{ivIncludeValue} property:

- If the property value is larger than the inclusion value, the cell is ignored.
- If the property value is equal to or less than the inclusion value, the cell is processed.

A cell without an \textit{ivIncludeValue} property is treated as if it has a property value of 0 (zero). If you do not specify an inclusion value when you start this tool, then it defaults to a value of 1,000.

<table>
<thead>
<tr>
<th>If you start Diva verification...</th>
<th>You set the inclusion value...</th>
</tr>
</thead>
<tbody>
<tr>
<td>By using the \textit{ivVerify} command</td>
<td>By specifying the \textit{-inc} option</td>
</tr>
<tr>
<td>By using the SKILL commands \textit{ivDRC} or \textit{ivExtract}</td>
<td>By specifying the \textit{?inc} option</td>
</tr>
<tr>
<td>Interactively</td>
<td>In the start-up form with the entry \textit{Inclusion Limit}</td>
</tr>
</tbody>
</table>

Consider the following example:

Cell \textit{A} has no property.

Cell \textit{B} has the property \textit{ivIncludeValue} = 3.

Cell \textit{C} has the property \textit{ivIncludeValue} = 5.

Cell \textit{D} has no property and references cells \textit{A}, \textit{B}, and \textit{C}.

If you set the inclusion limit to 0 on startup, cells \textit{B} and \textit{C} are ignored because their property values are greater than the start-up inclusion value.
Macro Cell Mode

Normally, for the extraction of hierarchical structured layouts, this tool expands the contents of each instance of a cell in the hierarchy into a single flat representation. The flattening process continues until all cell instances are gone, leaving only shapes on layers.

Macro cell processing modifies this process so that specific cells in the hierarchy are not expanded into a flat representation, but are converted into instances of devices. When you look at the extracted cellview, instances of macro cells appear the same as normal cell instances.

Using MacroCell Processing

You can use macro cell processing to significantly reduce extraction run time when you use a cell more than once. The Diva verification tool can check each cell before it is added to the system library. Using that cell as a macro cell causes this tool to treat it as a device in the layout. This software replaces all cell internal processing by the recognition of the external connections to the cell.

You can also use macro cell processing when you have not yet designed the contents of a cell or when you don’t have access to the contents of a cell. You must define the pin connections to the cell before this tool can process it.

Defining a Macro Cell

If you specify macro cell mode, a cell becomes a macro cell based on the following requirements:

■ The cell must contain one or more pins on connected layers. A layer is connected if it appears in a `geomConnect` command. The pins form the external connections to the cell.

■ If Diva verification is not in hierarchical mode and the SKILL function `ivUserCellTypeFilter` is defined by the user, `ivUserCellType` is called with the full instance path through the hierarchy to the instance currently being tested. This function can be defined in the run specific file. If the value returned is a non-empty string, the cell is processed according to that value.

The value can be:

- `macro` Forces the cell to be a macro cell providing the cell has pins.
If Diva verification is not in hierarchical mode and the SKILL function is not defined or returns an empty value and the instance has an \textit{ivCellType} property, the property value is used to define the processing mode.

The \textit{ivCellType} property value can be:

- \textit{macro} Forces the cell to be a macro cell providing the cell has pins.
- \textit{graphic} Forces the cell to not be a macro cell.
- \textit{none} Ignores the \textit{macroCellFile} and \textit{ivCellType} property and lets the cell become a macro or not, based on its pins and \textit{prCellType} property.
If the `ivCellType` is not found on the instance, and the `macroCellFile` is provided in the run-specific rules, and the cell being processed matches an entry in the file, the cell is processed according to the value defined for that cell in the file.

The `macroCellFile` value can be:

- **macro** Forces the cell to be a macro cell providing the cell has pins.
- **graphic** Forces the cell not to be a macro cell.
- **none** Ignores the `ivCellType` property and lets the cell become a macro or not based on its pins and `prCellType` property.

If the `macroCellFile` is not provided in the run-specific rules, or the cell does not exist in the file, this tool uses the `ivCellType` property on the cell to define the processing mode.

The `ivCellType` property value can be

- **macro** Forces the cell to be a macro cell providing the cell has pins.
- **graphic** Forces the cell not to be a macro cell.

If a cell does not have an `ivCellType` property, it becomes a macro cell if it has a `prCellType` of:

- Standard
- IO
- Corner

**Note:** The top level of hierarchy cannot be a macro cell. Even if the top cell conforms to the requirements of a macro cell, the program treats it as a normal cell. If you switch to hierarchical processing mode, the hierarchical cell control takes precedence over the macro cell interpretation unless the `ivCellType` property explicitly defines the cell as a macro cell.
The *macroCellFile* and *ivCellType* values are

<table>
<thead>
<tr>
<th></th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>macro</strong></td>
<td>Forces the cell to be a macro cell providing the cell has pins.</td>
</tr>
<tr>
<td><strong>graphic</strong></td>
<td>Forces the cell not to be a macro cell.</td>
</tr>
<tr>
<td><strong>none</strong></td>
<td>Ignores the <em>ivCellType</em> property and lets the cell become a macro or not based on its pins and <em>prCellType</em> property.</td>
</tr>
</tbody>
</table>

**Controlling Macro Cell Hierarchy**

When traversing the hierarchy, Diva verification descends into the cellview defined by the instance of the current cell unless the property *originalViewName* exists on the cell.

If the property *originalViewName* is specified for the cell, this tool switches into the cellview defined by this property providing

- The cellview defined by *originalViewName* exists.
- The cell has no *ivCellType* property either attached to the cell or defined in the *macroCellFile*.
- The *prCellType* of the cell is *macro*.

**Starting Macro Cell Processing**

To start macro cell extraction for interactive processing from a layout window, select the Extract command from the Verify menu, then select *macro cell* for the extract method in the Extract form.

To start macro cell extraction using SKILL, use the *ivExtract* command to specify a t value for the *?macro* argument.

**Note**: The command stream for macro cell processing must contain a *geomConnect* command and can contain any other DRC and extraction commands for layer processing and device recognition. Normally, you can mix extracted devices with macro cell devices.
How Diva Verification Processes Macro Cells

When this tool recognizes a macro cell, it does not recognize or expand cell instances within that macro cell. This software product expands shapes only on layers whose purpose is *boundary* or shapes associated with *pins*.

Although the program copies the cell boundary from a macro cell (if it is on purpose *boundary*), the boundary is not used for connectivity processing. However, you can use the boundary to generate “keep-out” layers. Keep-out layers represent regions on the macro cell that you don’t want other cells’ layers to overlap.

This tool places every macro cell as a device instance in the extracted version. At the topmost level, it gives that instance the same name as the original cell instance in the layout. At lower levels of hierarchy, the program gives instances the names +1, +2, +3, and so on. Make sure these names do not conflict with your own cell names.
Executing Diva Verification Using SKILL Functions

In addition to using menu commands as described in Chapter 14, “Verify Menu Commands,” you can use SKILL functions and UNIX commands to run Diva verification. You use SKILL functions in the Cadence design framework II environment. When using UNIX commands, you are running this tool in batch mode. This chapter describes the commands, shows you the syntax for each command, and provides examples.
ivConICe

ivConICe(
  ?cell
  ?resModelData
  ?capModelData
  [?ConICeName]
  [?coupling]
  [?accuracy]
) => t / nil

Description

Starts the ConICe RC network reduction section of the layout verification program. Each of
the arguments is a keyword preceded by a question mark (?). You must give each keyword a
value, as indicated below. The ?cell, ?resModelData, and ?capModelData
keywords are required, the other keywords are optional.

Arguments

?cell

Object ID of the cell to be processed.

When using the command from the CIW, you can specify the object ID as

?cell geGetWindowCellView()

When using the command in the no-graph mode, you can specify the object ID as

?cell dbOpenCellViewByType( "libname"
"cellname" "view" nil "r" )

?resModelData

A quoted string with the names of the cellviews to be recognized as resistors and the name of the property on each cellview which holds the resistance value. The cellview name and property name must exist in pairs with spaces between the cellview and property names. If all cellview and property information is based on parasitic extraction performed by Diva verification, the string can be specified by referencing the extracted view property resModelData as
You can specify the string directly as

?resModelData "pres r presistor res"

?capModelData A quoted string with the names of the cellviews to be recognized as capacitors and the name of the property on each cellview which holds the capacitance value. The cellview name and property name must exist in pairs with spaces between the cellview and property names. If all cellview and property information is based on parasitic extraction performed by Diva verification, the string can be specified by referencing the extracted view property capModelData as

?capModelData cell~>capModelData

You can specify the string directly as

?capModelData "pcap c pcapacitor cap"

?ConICeName A quoted string which is the name of the cell view for the concICe view. The default name is "concice".

?coupling Set to a number which is the scale factor to be applied to the value of coupling capacitors. The default value is 1.

?accuracy Set to a fixed point number, greater than zero, which is the level of accuracy desired in the resulting RC chains. The default value is 2.

Values Returned

\( t \) Returns \( t \) if the execution succeeds.

\( \text{nil} \) Returns \( \text{nil} \) if the execution fails.

Examples

The following example runs an RC reduction on the current cellview.

```clojure
(cell = geGetEditCellView() ivConICe( ?cell cell ?res cell~>resModelData ?capModelData)
```
This example illustrates how the values of the options can be preset in symbols.

```plaintext
cell = geGetEditCellView()
rd = cell~>resModelData
cd = cell~>capModelData
name = "concie"
cf = 0.95
acc = 3
ivConcICe( ?cell cell ?res rd ?capModelData cd ?ConcICeName name
    ?coupling cf ?accuracy acc )
```
ivCreatePCells

ivCreatePCells( filename ) => t/nil

Description

This function creates parameterized versions of existing cells for the display of devices created by the device extraction commands.

Prerequisites

The technology file for the library to which the original cells belong must contain a definition of the view name being created with any desired view type. This definition must have the same value of DBUPerUU as other view names having layout view types.

Arguments

filename Name of a file containing the list of cells to be processed.

The contents of the file consist of one line for each cell to be processed. The format of that line is as follows:

    libName cellName viewName-in viewName-out magnification

libName is the name of the library from which the original cell is read and the new cell is written.

cellName is the name of the cell from which the parameterized cell is derived and the name of the resultant parameterized cell.

viewName-in is the name of the view of the cell from which the parameterized cell is derived.

    The input cellview is opened for read only and is not changed by the ivCreatePCells procedure unless the output cellview is the same as the input cellview, in which case it is overwritten.

viewName-out is the name of the view of the resultant parameterized cell. The resultant cellview type is the one you defined for the view name in the technology file.
If a cell already exists with the requested output cellview, it is deleted. The exception to this deletion is if the output cellview is the same as the input cellview, in which case the input cellview is extended to convert it into a parameterized cell.

The view name for predefined parameterized cells in the system library is `ivpcell`.

`magnification` is the floating-point number which is used to change the size of the shapes in the resultant cellview. The number is applied as a factor to the size of any shape of the input cellview as it is copied to the output cellview.

This magnification can be used to make the display of the cell (which is normally derived from a symbol view) more acceptable in the layout environment.

This parameter is ignored if the output cellview is the same as the input cellview, but it must still be present.

### Values Returned

- `t` Returns `t` if the execution succeeds.
- `nil` Returns `nil` if the execution fails.

### Examples

The following is an example of the command.

```plaintext
ivCreatePCells( cell_file )
```

The following is an example of the contents of the file.

```plaintext
mylib nfet symbol ivpcell 5
mylib pfet symbol ivpcell 5
mylib njfet symbol ivpcell 1
mylib resistor symbol ivpcell 1
```
ivDRC

ivDRC(
  ?cell
  [?echo]
  [?full]
  [?hier]
  [?optimize]
  [?inc]
  [?join]
  [?set]
  [?rsf]
  [?area]
  [?rulesFromUnix]
  [?rulesLibName]
  [?rulesFilePath
) => t / nil

Description

Starts the DRC section of the layout verification program. Each of the arguments is a keyword preceded by a question mark (?). You must give each keyword a value or a symbolic name set to the value. The ?cell argument is required; the other arguments are optional.

Arguments

?cell

Object ID of the cell to be processed. The cell must be opened in append mode.

When using the command from the CIW, you can specify the object ID as

?cell geGetWindowCellView( )

When using the command in the no-graph mode, you can specify the object ID as

?cell dbOpenCellViewByType( "libname" "cellname" "view" nil "a" )

?echo

Echoes the rules as they are executed when set to t. The default is nil.
Diva Reference
Executing Diva Verification Using SKILL Functions

?full
Analyzes the complete design when set to t. Set to nil to perform an incremental analysis and check only those areas that have changed since the last DRC run. The default is nil.

?hier
Performs a hierarchical analysis when set to t. Set to nil to perform a flat analysis. The default is nil.

?optimize
Applies pattern recognition optimization to the hierarchical analysis when set to t. This option only applies if ?hier is set to t. The default is t.

?inc
Set to a fixed point number to determine the cell inclusion-exclusion limit. The default is 1,000.

?join
Merges nets having the same name into a single net during connectivity analysis when set to t. Set to nil to prevent this merging. A warning message is displayed if two or more nets have the same name when set to the default nil.

?set
Sets the names of the switches to be used in the conditional rules execution. If you set two or more switches, you must enclose the list in quotation marks with a space between the switch names. You must also put a space between the keyword and the opening quote. If this option is not used, no user-defined switches are set.

?set "docap"
?set "docap dores"

?rsf
Name of the file containing run-specific commands. The default does not use a run-specific file.

?area
Defines the area of the circuit to be processed by DRC. The area can be one or more rectangles with rectangular holes in them. This option takes precedence over a verifyArea command in the run-specific command file. You specify the area by using two pairs of x and y coordinates, enclosed in parentheses, that define the bottom left and top right of a rectangular area. You must put a colon between the x and y and at least one space between each pair of coordinates. For example

?area list( 10:100 5000:2500 )
You can get further syntax details from the DRC run-specific command `verifyArea`, which has the same format as this option. The checking halo around the area can be set to zero using the `drcZeroHalo` command in the run-specific file or the `.cdsinit` file. This option is disabled when running in hierarchical or incremental modes. For more information on zero halo, see the “Area Halo Processing” on page 261.

```plaintext
?rulesFromUnix
```

If you set this option to `t`, the program gets the DRC rules from a text file defined by the `?rulesFilePath` option. If you do not define this option, or if you set it to `nil`, the program gets the DRC rules from the library. The `?rulesFilePath` and `?rulesLibName` options determine the library name and the rules file name.

```plaintext
?rulesLibName
```

If you specify this option and do not set the `?rulesFromUnix` option to `t`, the program gets the rules from the library name defined by this option. If you do not specify this option, or if you set it to `" "`, and do not set the `?rulesFromUnix` option, the program gets the rules from the library containing the top level cellview.

```plaintext
?rulesFilePath
```

If you use this option and set the `?rulesFromUnix` option to `t`, this option defines the name of the rules file within the Unix environment. If you use this option and do not set the `?rulesFromUnix` option to `t`, this option defines the name of the rules file relative to the library. If you do not use this option, or if you set it to `" "`, the program uses the default rules name of the `divaDRC.rul`. To have the program get the default rules from the library you specify, you must put the rules at the library directory level.

**Note:** You can reference a rules file in a library from the library directory itself, the cell directory, or the view directory.

**Values Returned**

- **t**: Returns `t` if the execution succeeds.
- **nil**: Returns `nil` if the execution fails.
Examples

The following example runs full DRC on the design in the active window and echoes the rules to the CIW.

```lisp
ivDRC( ?cell geGetWindowCellView( ) ?echo t ?full t )
```

This example runs incremental DRC with the `do_drc_connect` switch to include optional rules and also uses the commands in the `myrules` run-specific file.

```lisp
ivDRC( ?cell geGetWindowCellView( ) ?set "do_drc_connect"
  ?rsf "myrules" )
```

This example runs hierarchical DRC, uses an inclusion value of 3, enables joinable nets, and calls commands from a run-specific command file.

```lisp
ivDRC( ?cell dbOpenCellViewByType( "testLib"
  "myCircuit" "layout" nil "a" ) ?hier t ?inc 3 ?join t
  ?rsf "chip.data" )
```

This example shows how the values of the options can be preset in symbols.

```lisp
e = t
h = nil
ivDRC( ?cell geGetWindowCellView( ) ?hier h ?echo e )
```
ivERC

ivERC(
    runDir
    libName
    cellName
    viewName
    netlist
    jobPriority
    rulesFromUnix
    rulesLibName
    rulesFilePath
    machine
) => t / nil

Description

Starts the ERC program of the layout verification system. All arguments are required.

Arguments

runDir

Name of the directory that you want to contain the ERC run information. If this directory does not exist, it is created and its contents are initialized with the values defined in this command. If the directory does exist, its contents are overwritten with the values defined in this command.

The name is a text string in double quotation marks. It must be either a full path name preceded by a slash (/) or a name relative to the current working directory.

libName

A text string in quotation marks that defines the library name containing the cellview to be checked.

cellName

A text string in quotation marks that defines the cell name of the cellview to be checked.

viewName

A text string in quotation marks that defines the view name of the cellview to be checked.

netlist

Set this option to t to generate a new netlist for the cell. Otherwise, the existing netlist in the run directory is used.
<table>
<thead>
<tr>
<th><strong>jobPriority</strong></th>
<th>An integer from 0 to 20 that defines the priority of the ERC job. The highest priority is 0.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>rulesFromUnix</strong></td>
<td>Set this option to t to get the ERC rules from a text file defined by the rulesFilePath argument.</td>
</tr>
<tr>
<td></td>
<td>If you do not specify this option, or if you set it to nil, the program gets the ERC rules from the library. The rulesLibName and rulesFilePath determine the library name and rules file name.</td>
</tr>
<tr>
<td><strong>rulesLibName</strong></td>
<td>If you set rulesFromUnix to nil, the program gets the rules from the library defined by this argument, which is a text string enclosed in quotation marks.</td>
</tr>
<tr>
<td><strong>rulesFilePath</strong></td>
<td>A text string enclosed in quotation marks defining the file name or file path name of the rules file. If you set rulesFromUnix to t, this name is relative to the UNIX file system. If you set rulesFromUnix to nil, this name is relative to the library name.</td>
</tr>
<tr>
<td><strong>machine</strong></td>
<td>A text string in quotation marks defining the name of the remote machine that is running ERC. Set this value to nil if you are running ERC locally.</td>
</tr>
</tbody>
</table>

**Values Returned**

<table>
<thead>
<tr>
<th><strong>t</strong></th>
<th>Returns t if the execution succeeds.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>nil</strong></td>
<td>Returns nil if the execution fails.</td>
</tr>
</tbody>
</table>

**Example**

The following example runs ERC on the fflatch extracted cellview in the library mylib. The run directory is ercrun. A new netlist is created before checking. The job priority is set to 3, and the rules file divaERC.rul is obtained from the library mylib.

```sql
ivERC( "/usr/me/mydata/ercrun" "fflatch" "extracted"
       "mylib" t 3 nil "mylib" "divaERC.rul" nil )
```
ivExtract

ivExtract(
    ?cell
    [?echo]
    [?macro]
    [?hier]
    [?full]
    [?inc]
    [?join]
    [?set]
    [?rsf]
    [?rulesFromUnix]
    [?rulesFilePath]
    [?rulesLibName]
    [?extractedName]
    [?excellName]
) => t / nil

Description

Invokes the Extraction section of the layout verification program. The ?cell argument is required; the other arguments are optional.

Arguments

?cell
  Object ID of the cell to be processed. The cell must be opened in append mode.

When using the command from the CIW, you can specify the object ID as

?cell  geGetWindowCellView( )

When using the command in the no-graph mode, you can specify the object ID as

?cell dbOpenCellViewByType("libname" "cellname"
"view" nil "a")

?echo
  Echoes the rules as they are executed when set to t.
**Diva Reference**

**Executing Diva Verification Using SKILL Functions**

- **?macro**
  Performs a macro cell extraction. Set to *nil* for a normal extraction when set to *t*.

- **?hier**
  Performs hierarchical analysis when set to *t*. Set to *nil* for a flat analysis.

**Note:** ?hier takes precedence over ?macro.

- **?full**
  This option is only meaningful during hierarchical analysis.
  Analyzes all cells throughout the hierarchy when set to *t*. Set to *nil*, to analyze only those cells that have changed since the last extraction.
  For flat extraction, the mode of execution is as if ?full is set to *t*, even if it is actually set to *nil*.

- **?inc**
  Set to a fixed point number, to determine the cell inclusion/exclusion limit. The default value is 1,000.

- **?join**
  Merges nets having the same name into a single net during connectivity analysis when set to *t*. Set to *nil*, to prevent his merging. A warning message is issued if two or more nets have the same name.

- **?set**
  Sets the names of the switches to be used in the conditional rules execution. You can set one or more switches. If you set two or more switches you must enclose them in quotation marks and put a space between the keyword and the opening quote.

  ```
  ?set "docap"
  ?set "docap dores"
  ```
  If you do not use this option, no switches are set.

- **?rsf**
  Name of the file containing run-specific commands.

- **?rulesFromUnix**
  If you set this option to *t*, the program gets the extract rules from a text file defined by the rulesFilePath argument.
  If you do not specify this option, or if you set it to *nil*, the program gets the extract rules from the library. The ?rulesLibName and ?rulesFilePath arguments determine the library name and rules file name.
### Diva Reference

**Executing Diva Verification Using SKILL Functions**

<table>
<thead>
<tr>
<th>Option</th>
<th>Description</th>
</tr>
</thead>
</table>
| `?rulesLibName`       | If you use this option and you do not set the `?rulesFromUnix` to `t`, the program gets the rules from the library name defined by this option.  
If you do not use this option or set it to "", and you do not set the `?rulesFromUnix` option, the program gets the rules from the default technology library. |
| `?rulesFilePath`      | If you use this option and set the `?rulesFromUnix` option to `t`, this option defines the name of the rules file within the UNIX environment.  
If you use this option and do not set the `?rulesFromUnix` option to `t`, this option defines the name of the rules file relative to the library.  
If you do not use this option, or set it to "", the program uses the default rules name of the `divaDRC.run`.  
To have the program use the default rules from the default library, you must place them at the same location as the `techfile.cds` file. If the default rules are to be accessed from the library you specify, they will be assumed to be at the library directory level. |
| `?extractedName`     | The name of the cell view for the extracted view. The default name is `extracted`.                                                                                                                           |
| `?excellName`        | The name of the cell view for the excell view. The default name is `excell`.                                                                                                                                  |

**Values Returned**

- **t**: Returns `t` if the execution succeeds.
- **nil**: Returns `nil` if the execution fails.
Examples

The following example runs an extraction on the current cellview.

```s/filesystem
ivExtract( ?cell geGetEditCellView( ) )
```

This example runs a macro extraction on a cellview in a library, which can be open or closed.

```s/filesystem
ivExtract( ?cell dbOpenCellViewByType( "testLib" "testCell layout")?macro t )
```

This example runs a full hierarchical extraction on the current cellview and echoes the commands to the CIW.

```s/filesystem
ivExtract( ?cell geGetWindowCellView( ) ?echo t ?hier t ?full t )
```

This example runs a hierarchical extraction incrementally, has an inclusion value of 3, enables joinable nets, and uses commands from the run-specific file, `chip.data`.

```s/filesystem
ivExtract( ?cell dbOpenCellViewByType( "testLib" "myCircuit layout" "a")?hier t ?inc 3 ?join t ?rsf "chip.data" )
```

This example runs an incremental flat extraction on the current cellview only if the switch `parasitics` evaluates to true and uses commands from the run-specific file `myrules`.

```s/filesystem
ivExtract( ?cell geGetWindowCellView( ) ?set parasitics ?rsf "myrules" )
```

This example illustrates how the values of the options can be preset in symbols.

```s/filesystem
e = t
h = nil
ivExtract( ?cell geGetWindowCellView( ) ?hier h ?echo e )
```
ivLVS

ivLVS(
    runDir
    layoutLibName
    layoutCellName
    layoutViewName
    netlistLayout
    schematicLibName
    schematicCellName
    schematicViewName
    netlistSchematic
    deviceFixing
    noRewire
    termCorr
    useFileCorr
    corrFile
    xref
    jobPriority
    rulesFromUnix
    rulesLibName
    rulesFilePath
    machine
) => t / nil

Description

Invokes the Layout Versus Schematic (LVS) Comparison program of the layout verification system. All arguments are required.

Arguments

runDir

Name of the directory into which you want to put the LVS run information. If this directory does not exist, Diva verification creates it and initializes the contents with the values defined in this command. If the directory does exist, this tool overwrites its contents with the values defined in this command.

The name is a text string in double quotation marks. It must be either a full path name preceded by a slash (/), or a name relative to the current working directory.
### Diva Reference

**Executing Diva Verification Using SKILL Functions**

<table>
<thead>
<tr>
<th>Variable</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>layoutLibName</td>
<td>A text string in quotation marks that defines the library name containing the extracted cellview to be compared.</td>
</tr>
<tr>
<td>layoutCellName</td>
<td>A text string in quotation marks that defines the cell name of the extracted cellview to be compared.</td>
</tr>
<tr>
<td>layoutViewName</td>
<td>A text string in quotation marks that defines the view name of the extracted cellview to be compared.</td>
</tr>
<tr>
<td>netlistLayout</td>
<td>Generates a new netlist for the layout when set to t. Otherwise, the program uses the existing layout netlist in the run directory.</td>
</tr>
<tr>
<td>schematicLibName</td>
<td>A text string in quotation marks that defines the library name containing the schematic cellview to be compared.</td>
</tr>
<tr>
<td>schematicCellName</td>
<td>A text string in quotation marks that defines the cell name of the schematic cellview to be compared.</td>
</tr>
<tr>
<td>schematicViewName</td>
<td>A text string in quotation marks that defines the view name of the schematic cellview to be compared.</td>
</tr>
<tr>
<td>netlistSchematic</td>
<td>Generates a new netlist for the schematic when set to t. Otherwise, the existing schematic netlist in the run directory is used.</td>
</tr>
<tr>
<td>deviceFixing</td>
<td>Applies device fixing when set to t.</td>
</tr>
<tr>
<td>noRewire</td>
<td>Does not do automatic rewiring when set to t.</td>
</tr>
<tr>
<td>termCorr</td>
<td>Applies terminal correspondence when set to t.</td>
</tr>
<tr>
<td>useFileCorr</td>
<td>Uses a file of correspondence points when set to t.</td>
</tr>
<tr>
<td>corrFile</td>
<td>A text string in quotation marks that defines the name of the file of correspondence points requested by the useFileCorr keyword. If you do not need a correspondence point file, you must enter an empty string (&quot; &quot;).</td>
</tr>
</tbody>
</table>
xref
Generates a layout to schematic correspondence text file when set to `t`. The name given to this file is `xref.out`.

jobPriority
An integer from 0 to 20 that defines the priority of the LVS job. The highest priority is 0.

rulesFromUnix
Set this option to `t` to get the LVS rules from a text file defined by the `rulesFilePath` argument.
If you do not specify this option, or if you set it to `nil`, the program gets the LVS rules from the library. The `rulesLibName` and `rulesFilePath` arguments determine the library name and rules file name.

rulesLibName
If you set `rulesFromUnix` to `nil`, the program gets the rules from the library defined by this argument, which is a text string enclosed in quotation marks.

rulesFilePath
A text string enclosed in quotation marks which defines the file name or file path name of the rules file. If you set the `rulesFromUnix` argument to `t`, this name is relative to the UNIX file system. If you set `rulesFromUnix` to `nil`, this name is relative to the library name.

machine
A text string in quotation marks defining the name of the remote machine that is running LVS. Set this value to `nil` if you are running LVS locally.

Values Returned

*t*  
Returns `t` if the execution succeeds.

*nil*  
Returns `nil` if the execution fails.

Example
The following example runs LVS on extracted and schematic views of the same cell, specifying a run directory `lvsrun`. New netlists are generated for both cellviews. No device fixing is used, and rewiring is enabled. Terminals are not used as correspondence points, but the correspondence point file `myCorr` is used. No cross-reference file is output, and the job priority is set to 3. The rules `divaLVS.rul` are obtained from the library `mylib`.
ivLVS( "/usr/me/mydata/lvsrun"
 "mylib" "fflatch" "extracted" t
 "mylib" "fflatch" "schematic" t
 nil nil t t "mycorr" nil 3 )
 nil "mylib" "divaLVS.rul" nil )
)
ivRestart

ivRestart(?dir
    [?echo]
) => t / nil

Description

Initiates a DRC or Extraction run previously saved using the checkPoint command in the verification rules.

The program creates a checkpoint directory when it executes the checkPoint command. The checkpoint directory contains the status of the previous run at the time of the last checkpoint. The restart point is at the last checkpoint saved in the directory.

When restarting a run, you cannot change the verification rules command stream except to add additional checkpoint commands after the command that created the current checkpoint. You can find the current checkpoint by viewing the line file in the checkpoint directory. This file contains line numbers of the checkpoints in the command file.

For more information about the checkPoint command, see the “Using Checkpoints” on page 43.

Arguments

?dir Name of the directory, enclosed in quotation marks, containing the checkpoint information.

?echo Echoes the rules as they are executed when set to t. This option overrides any ?echo option used in the restarted run. The default is nil.

Values Returned

t Returns t if the execution succeeds.

nil Returns nil if the execution fails.
Examples

The following example shows the files in the checkpoint directory \texttt{cpoint\_dir} that are to be used to restart the program.

\begin{verbatim}
   ivRestart( ?dir "cpoint\_dir"
\end{verbatim}

This example shows the files in the checkpoint directory \texttt{/myroot/mydirs/cpoint} that are to be used to restart the program. Echoes all Diva verification commands to the CIW.

\begin{verbatim}
   ivRestart( ?dir "myroot/mydirs/cpoint" ?echo t
\end{verbatim}

**Executing Diva Verification using UNIX**

You can run this verification as a batch program from the UNIX prompt. This section gives you the syntax, description, and examples for the UNIX command.
ivVerify

ivVerify ( [cell] [view]
  [-lib library]
  [-dir directory_name]
  [-rsf run_specific_rules_name]
  [-set switch_name] ...
  [-full]
  [-hier]
  [-echo]
  [-macro]
  [-optimize]
  [-inc]
  [-join]
  [-area]
  [-z]
  [-rl rules_lib_name]
  [-rf rules_file_name]
  [-ru unix_file_path]
  [-extracted]
  [-excell]
) => t / nil

Description

Runs DRC and extraction from the UNIX environment.

For all optional arguments, you must use a keyword preceded by a minus. You can abbreviate all the keywords to the first letter.

Arguments

function One of three keywords.

drc Starts a DRC run.

extract Starts an extraction run.

restart Restarts a DRC or Extraction run previously saved using the checkPoint command.
You must specify the name of the checkpoint directory defined in the `checkPoint` command.

```bash
restart -dir directory_name
```

The restart point is at the last checkpoint saved in the directory.

When restarting a run, you cannot change the verification rules command stream except to add additional checkpoint commands after the command that created the current checkpoint. You can find the current checkpoint by viewing the `line` file in the checkpoint directory. This file contains the line numbers of the checkpoints in the command file.

When you use `restart`, you can only specify the restart directory name and the option `-echo`.

```bash
cell
```

Name of the cell to be processed. Do not use with `restart`.

```bash
view
```

Name of the cellview to be processed. Do not use with `restart`.

```bash
-lib
```

Name of the library of the cell to be processed (top level). If you don't specify a library, the system uses the first cellview it finds that matches the specified name by scanning all libraries specified in the `cds.lib` file.

```bash
-lib mylib
```

```bash
-dir
```

Used with `restart` to specify the name of the checkpoint directory.

```bash
-rsf
```

Name of the file that contains run-specific commands. If you do not use this option, no run-specific file is used.

```bash
-rsf myrules
```

```bash
-set
```

List of switches to be used in the conditional rules execution. You must enclose two or more switches in quotation marks and put a space between the keyword and the opening quote.

```bash
-set docap
-set "docap dores"
```
You must use this option if you have not used the drc? and extract? commands in the verification rules. Otherwise, no switches are set. If you have used drc? and extract?, you do not need this option.

- **full**
  Specifies that full analysis is to be performed for DRC. If you do not use this keyword, the program performs incremental analysis, and checks only those areas that changed since the last DRC run.

  For extraction, this optional keyword is only meaningful during hierarchical analysis. If used, all cells throughout the hierarchy are analyzed. If not, only those cells that have changed since the last extraction are analyzed.

  For flat extraction, a full analysis is always performed.

- **-hier**
  Performs hierarchical analysis. If not used, flat analysis is performed.

  **Note:** For extraction **-hier** takes precedence over **-macro**.

- **-echo**
  Echoes commands as they are executed when set to t.

- **-macro**
  Performs macro cell extraction. If not preset, normal extraction is performed.

- **-optimize**
  Applies pattern recognition optimization to the hierarchical analysis. This option only applies if the keyword **-hier** is defined.

- **-inc**
  A fixed point number that defines the cell inclusion-exclusion value. If not defined, the value is defaulted to zero.

  **-inc 3**

- **-join**
  Merges nets with the same name into a single net during connectivity analysis. If this option is not used, a warning message is generated when two or more nets have the same name.

- **-area**
  Definition of a rectangular area of the circuit to be processed. The definition has this form

  **-area "list( 0:0 10:10 )"**
The definition is enclosed in quotation marks and consists of the \textit{list} function with two pairs of coordinates in parentheses. Each value in the coordinate pairs is separated by a colon, and the two pairs are separated by a space.

The first coordinate pair defines the lower left corner of the area to be processed, and the second coordinate pair defines the upper right corner of the area to be processed.

\textbf{-z}

Sets the halo to be used in area processing to zero. The halo is the distance added around the checking area to ensure all interactions and relationships that could affect the data in the area are included.

The halo distance normally used is dependent on the dimensions in the rules file and the conjuncted relationships between rules. For some rule sets, the halo can be very large and have a serious impact on run time.

This option can cause true violations to be missed and should only be used with discretion.

This option is disabled when running in hierarchical or incremental modes, and is overwritten by the \textit{drcZeroHalo} command in the run-specific file.

\textbf{-rl}

The name of the library from which the rules are read.

If you do not use this option and you use the \textit{-rf} option, the program gets the rules from the default technology library.

\textbf{-rf}

The path name of the rules file in the library.

The rules file can be referenced in a library only from the library directory itself, the cell directory, or the view directory.

If you do not use this option and you define the library by the \textit{-rl} option, the default rules file name of \textit{divaDRC.run} or \textit{divaEXT.run} are referenced from the library directory level.

If you do not use this option and you also do not use the \textit{-rl} and \textit{-ru} options, the program gets the rules from the default rules file
name in the default technology library at the location of the `techfile.cds` file.

-**ru**

The path name of the rules file to be read from the UNIX environment.

This option is not valid in conjunction with the `-rl` and `-rf` options.

-**extracted**

The name of the cell view for the extracted view. The default name is `extracted`.

-**excell**

The name of the cell view for the Excell view. The default name is `excell`.

**Values Returned**

- **t**
  Returns `t` if the execution succeeds.

- **nil**
  Returns `nil` if the execution fails.

**Examples**

The following are examples of the `ivVerify` command.

```bash
ivVerify drc mycell layout -lib mylib -echo -full -join
```

Runs full flat DRC on `mycell layout` in the library `mylib`. Echoes all Diva verification commands to the CIW and enables joinable nets.

```bash
ivVerify restart -dir save_dir -echo
```

Restarts a DRC using the checkpoint directory `save_dir` and echoes the this tool’s commands to the CIW.

```bash
ivVerify extract mycell layout -inc 3 -set "dores docap"
```

Runs a macro Extraction on a design, using commands in the run-specific file `myrules`.

```bash
ivVerify extract mycell layout -inc 3 -set "dores docap"
```

Runs an incremental flat Extraction on the design, using an inclusion value of 3 only after evaluation of switches `dores` and `docap`.

June 2000 93 Product Version 4.4.6
Writing Diva Verification Rules

Introduction

A verification rules file contains DRC and Extraction rules, ERC rules, and LVS rules. You write these rules into the verification rules file using the SKILL functions described in this manual.

The verification rules you add to the technology file are usually constant for a technology. There are times when you might make changes for a particular verification run. You do this by using commands that apply only to that run, and you put these run-specific commands in a separate file that you call when you perform a DRC or an extraction.

The following sections describe the rules you need for a verification rules file and also the commands you need for a run-specific file.

Creating a Verification Rules File

To run Design Rule Checking (DRC) and extraction on your design, you must define your DRC and extraction rules. You use a text editor to write these rules into a verification rules file using the SKILL functions defined in this manual. You can place these rules in a technology library or anywhere else in your system. When you start DRC or Extraction, the program prompts you for the location of the rules.

You can create a verification rules file by doing the following.

- Use the SKILL functions described in this manual to create your rules and control their processing.

- Place all DRC and extraction rules in the drcExtractRules() function, even though DRC and extraction are unrelated processes you run separately. Diva verification uses SKILL functions and subroutines to perform its verification tasks.

- Group the DRC commands together in one set and the Extract commands in another set. At run time, you can indicate which set of commands you want to use.
Enter the commands in the correct order for this tool to process the commands as it encounters them.

Use the `ivIf` and `checkpoint` commands to control the processing of the verification rules.

Store the rules file in the required location. The default file names are `divaDRC.rul` and `divaEXT.rul`, and the default location is in the technology library. However, you can store the rules anywhere with any name you choose. If you store the rules in a technology library, the rules file name must contain a dot to distinguish it from other library entries. If you use a standard convention of adding `.rul` to your file names, you can always identify the Diva verification rules files.

**Grouping Commands**

When you create a verification rules file, you must group the commands defining the rules under three SKILL functions that correspond to the three Diva verification programs (DRC/Extract, ERC, and LVS). For example, you must put all rules commands pertaining to extraction under the `drcExtractRules()` function. The table shows the processes associated with each SKILL function.

<table>
<thead>
<tr>
<th>Program</th>
<th>Function</th>
<th>Process</th>
</tr>
</thead>
<tbody>
<tr>
<td>DRC/Extract</td>
<td><code>drcExtractRules()</code></td>
<td>Layer Processing</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Design Rule Checking</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Connectivity Extraction</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Device Extraction</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Parameter Extraction</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Parasitic Extraction</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Resistance Extraction</td>
</tr>
<tr>
<td>ERC</td>
<td><code>ercRules()</code></td>
<td>Electrical Rule Checking</td>
</tr>
<tr>
<td>LVS</td>
<td><code>lvsRules()</code></td>
<td>Layout Versus Schematic Comparison</td>
</tr>
</tbody>
</table>
Nesting Commands

When creating verification rules, you can save time and effort by nesting commands inside other commands. Instead of creating two or three separate commands, you can combine them. You can nest commands to any depth.

This is particularly useful when you need to specify derived layers. A derived layer is a symbolic name for the result of a verification command. When you create a rule that uses a derived layer, you can do it in two ways. The first is to write the command that creates the derived layer and then use the name of the derived layer in a second command. A second and easier method is to write only one command, substituting the command that generated the derived layer for the derived layer name.

In this example, the command `geomAnd("poly" "diff")` creates a derived layer and is nested within the `drc` command.

```plaintext
   drc( geomSize( geomAnd( "poly" "diff") 2.0 )
      "cut" sep < 2 )
```

Another way of nesting commands is to define multiple layers in a command. You can use the `geomCat` command to group the layers. In this example, `geomCat` is nested within the `geomAnd` command and groups the layers `poly1`, `poly2`, and `poly3`.

```plaintext
   gate = geomAnd( geomCat( "poly1" "poly2" "poly3" )"diff" )
```

You can also substitute the value resulting from a layer property command anywhere a dimension is specified in a DRC. For example, instead of specifying an integer as the width dimension in the command

```plaintext
   drc( "poly" width <3 )
```

you can nest the layer property command `techGetLayerProp( techId "poly" "minWidth" )` within the DRC command

```plaintext
   cvId = ivGetCellView()
   techId = techGetTechFile(cvId)
   drc( "poly" width < techGetLayerProp( techId "poly" "minWidth" ) )
```

For more information on layer properties, refer to Chapter 5, “Layer Processing Concepts.”

Compiling and Optimizing Commands

This product uses a compiled and optimized version of the rules you provide. Compilation and optimization is the first step in the execution of DRC or extraction. However, this product does not have to compile and optimize the rules for every run. Compilation and optimization is done
only when you reference a new rule set for the first time, when you change the rules, and when you change the switch (ivIF switches) settings.

Optimization reduces run time, are usually invisible to you, and have no effect on the final results of the run. In some cases, they do change the order of command execution, which can affect checkpoints.

You should be aware of the following optimizations.

- A rule is eliminated if its output layer is not referenced by any other rule.
- All DRC checks to be performed on the same input layers are combined and processed in a single pass.
- All derived layer operations to be performed on the same input layers are processed in one pass.
- All duplicate functions are extracted and performed only once.
- All parameters and parasitics on the same layer are extracted in a single pass.

Derived layers that are no longer required are deleted immediately. This tool compiles the command set only once. It does not recompile subsequent runs that use the same set of commands, reducing the time required for these runs.

Compilation and optimization are performed after you have defined the ivIF switch settings so this tool set knows which functions are to be executed and can process them accordingly.

**Using Wildcards**

You can specify names using wildcard characters for various Diva verification commands. There are two levels of wildcards:

- **Simple wildcards**
  
  You can use the character "*" as a prefix or suffix to any label.

  This form of wildcards is easy to use, but does not allow the complex substitution available in regular expressions.

- **Regular expression wildcards**

  This format is the same as the format used in UNIX and in the SKILL function *rexMatch*. This level of wildcards allows far more complex expressions than the "*" wildcards described above.
This tool can confuse special symbols in regular expressions with simple wildcards or with special symbols normally used in network names. To distinguish between simple wildcards and regular expressions, you must prefix all regular expressions with a “?” For example, gnd* uses a simple wildcard evaluation. ?gnd* uses the general expression evaluation.

The following table summarizes wildcard expressions:

<table>
<thead>
<tr>
<th>Wildcard</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>gnd*</td>
<td>The program recognizes any characters for “*”</td>
</tr>
<tr>
<td>*gnd</td>
<td>The program recognizes any characters for “*”</td>
</tr>
<tr>
<td><em>gnd</em></td>
<td>The program recognizes any characters for both “*”s</td>
</tr>
<tr>
<td>*</td>
<td>Matches any string</td>
</tr>
<tr>
<td>**</td>
<td>Matches any string</td>
</tr>
<tr>
<td>?gnd*</td>
<td>Interprets gnd* as a regular expression</td>
</tr>
</tbody>
</table>

**Limitations**

If you have a normal label containing the character “*” at the beginning or end of its name, this tool interprets it as a simple wildcard character. To avoid this, you must convert the references to each label in this tool’s functions into a regular expression by placing a “\” before the “*”. For example, reference clk* as ?^clk\*. This product recognizes this as a regular expression and translates it back to the original form. The “^” in this expression specifies that no other clk expressions match this one.

Regular expressions take up considerable CPU time. Do not use regular expression if you can avoid them.

**Sample Verification Rules File**

This is a sample verification rules file for a standard pwell CMOS process. This file contains both DRC and extraction rules and uses the default switches drc? and extract?.

drcExtractRules(
/* The first section contains the geomOr statements that associate symbols with the physical layers to create derived layers. You can use these symbols in the rest of the rule set. You must put the geomOr statements before any switches so they are executed regardless of which switch you use at run time. */

bkgnd = geomBkgnd()
poly = geomOr( "poly" )
active = geomOr( "active" )
gate = geomOr( "gate" )
contact = geomOr( "contact" )
metal1 = geomOr( "metal1" )
via = geomOr( "via" )
metal2 = geomOr( "metal2" )

/*
DRC RULES FOR PWELL CMOS
The first ivIf switch is the drc? switch. If you use this switch, the DRC part of the file automatically executes during a DRC run.
*/

ivIf( switch( "drc?" ) then

/*
* POLY RULES
*/

drc( poly width < 3 "Poly width < 3.0 " )
drc( poly sep < 3 "Poly to Poly spacing < 3.0" )
drc( poly notch < 3 "Poly to Poly spacing < 3.0" )
drc( poly active sep < 2 "Field Poly to Active spacing < 2.0" )

/* The next three lines define the gate area and its edges. */

gate = geomAnd( poly active )
gatew = geomGetEdge( gate inside poly )
gatel = geomGetEdge( gate coincident poly )

/* The next two commands check the gate extension onto the field and the source/drain enclosure of the gate. */

*/
drc(poly gatew enc < 3 opposite "Poly gate overlap onto field < 3.0")

drc(active gatel enc < 3 opposite "Source/Drain enclosure of gate < 3.0")

/*
* CONTACT RULES
*/

// The saveDerived statements output bad contact geometries to the error layer. The first command outputs contacts that overlap onto the gate; the second command outputs parts of the contact that are not covered by metal.

*/

saveDerived( geomAndNot( contact geomOr( active poly ) )
"Contact not inside Active or Poly")

saveDerived( geomAndNot( contact metall1 )
"Contacts not covered by Metal1")

drc( contact width < 3 "Contact width < 3.0")

drc( contact sep < 3 "Contact to Contact spacing < 3.0")

drc( poly contact enc < 2 "Contact inside Poly < 2.0")

drc( metall1 contact enc < 2 "Contact inside Metal1 < 2.0")

drc( active contact enc < 2 "Contact inside Active < 2.0")

/*
* METALL1 RULES
*/

/* METALL1 RULES
*/

drc( metall1 width < 3 "Metal1 width < 3.0")

drc( metall1 sep < 4 "Metal1 to Metal1 spacing < 4.0")

drc( metall1 notch < 4 "Metal1 to Metal1 spacing < 4.0")

/*
* VIA RULES
*/
drc( via width < 3 "Via width < 3.0")

drc( via sep < 3 "Via to Via spacing < 3.0")

drc( via contact sep < 3 "Via to Contact spacing < 3.0")

/* This saveDerived statement outputs vias that overlap onto contacts.
*/

saveDerived( geomOverlap( via contact ) "Via not allowed over contacts")

drc( via poly sep < 2 "Via to Poly spacing < 2.0")

/* This saveDerived statement outputs vias that overlap onto poly.
*/
saveDerived( geomOverlap( via poly ) "Via not allowed over Poly" )
drc( metal1 via enc < 2 "Via inside Metal1 < 2.0" )
drc( metal2 via enc < 2 "Via inside Metal2 < 2.0" )

/* The next two saveDerived statements output the parts of vias that are not covered by metal1 or metal2. */

saveDerived( geomAndNot( via metal1 ) "Via not inside Metal1" )
saveDerived( geomAndNot( via metal2 ) "Via not inside Metal2" )

/*
   * METAL2 RULES
   */

drc( metal2 width < 5 "Metal2 width < 5.0" )
drc( metal2 sep < 5 "Metal2 to Metal2 spacing < 5.0" )
drc( metal2 notch < 5 "Metal2 to Metal2 spacing < 5.0" )

/*
   * EXTRACT RULES FOR PWELL CMOS
   */

/* The second ivIf switch is the extract? switch. If you use this switch, the extraction part of the file automatically executes during an extract run. */

ivIf( switch( "extract?" ) then

   /* The first section of the extraction part of the rules contains the geomOr statements that change the physical layers used only for extraction to symbols that you can use in the rest of the extraction rule set.
   You must precede special characters used in layer names with a backslash, so layer “p+” is renamed “p\+” and layer “n+” is renamed “n\+”.
   */

   well = geomOr( "well" )
   ngate = geomOr( "ngate" )
   pgate = geomOr( "pgate" )
   psd = geomOr( "psd" )
```
p\+ = geomOr( "p+" )
nsd = geomOr( "nsd" )
n\+ = geomOr( "n+" )

/* This section contains boolean functions such as geomAnd and
   geomAndNot that create new layers for device recognition and
   define interconnections between layers. */

nwell = geomAndNot( bkgnd well )
pdiff = geomAnd( active p\+ )
ndiff = geomAnd( active n\+ )
pgate = geomAnd( poly pdiff )
ngate = geomAnd( poly ndiff )
psd = geomAndNot( pdiff poly )
nsd = geomAndNot( ndiff poly )
ptap = geomAnd( well psd )
ntap = geomAnd( nwell nsd )
ptie = geomOr( ptap )
ntie = geomOr( ntap )

/* The geomConnect statement defines the connections between the
   layers and establishes the network to be extracted. */

geomConnect( 
  via( via metal1 metal2 )
  via( contact psd nsd poly metall1 )
  via( ntie nsd ntap )
  via( ptie psd ptap )
)

/* The geomStamp statement transfers connectivity information from
   one layer to another. The following example transfers the well
   tie connectivity to the well itself. This prevents the well from
   acting as a connection layer forming soft connects. */

nwell = geomStamp( nwell ntap error )
well = geomStamp( well ptap error )

/* The extractDevice statements define the devices used in the
circuit. */

extractDevice( pgate (poly "G") (psd "S" "D") (nwell "B") "pfet ivpcel1" )
```
extractDevice( ngate (poly "G") (nsd "S" "D") (well "B" )
"nfet ivpcell" )

/* The measureParameter commands extract device parameters. These
parameters can then be checked when you run an LVS.

Use the coefficient 0.5 to measure the gate width and gate length
parameters because the measureParameter command measures the
length of all edges defined in the rule and you only want the
length of one edge. For example, to measure the length of a gate,
measure the length of the gate edges that are inside and not
coincident with poly. Two edges satisfy that requirement. If you
do not multiply by 0.5, you are counting the gate length twice.
*/

pgateWidth = measureParameter( length ( pgate inside poly ) 0.5 )
pgateLength = measureParameter( length ( pgate coincident poly ) 0.5 )

/* Add the parameters to the devices by using the saveParameter
command. The parameter names must be the same as the names that
are used in the LVS rules.
*/

saveParameter( pgateWidth "w" )
saveParameter( pgateLength "l" )
ngateWidth = measureParameter( length ( ngate coincident poly ) 0.5 )
ngateLength = measureParameter( length ( ngate inside poly ) 0.5 )
saveParameter( ngateWidth "w" )
saveParameter( ngateLength "l" )

/* The saveInterconnect command writes layers to the extracted
cellview. Save the layers defined in the geomConnect command so
that you can cross-probe your nets after running LVS.
*/

saveInterconnect( nsd psd poly contact metal1 via metal2 )

/* The saveRecognition command creates the device recognition
polygons in the extracted cellview. It is used in conjunction
with the Model view name ivpcell in the extractDevice or
extractMOS command.
*/

saveRecognition( ngate "ngate" )
saveRecognition( pgate "pgate" )
ERC and LVS Verification Rules

To run Electrical Rules Check (ERC) or Layout Versus Schematic (LVS) on your design, you must define your ERC and LVS rules. You write these rules into a text format verification rules file using the SKILL functions defined in this manual. You can place these rules files in a technology library or anywhere else in your system. When you start ERC or LVS, the program prompts you for the location of the rules.

To create verification rules files, you do the following:

- Use the SKILL functions described in the ERC and LVS chapter to create a separate text rules file for each program.
- Put all ERC rules in the `ercRules()` function. Diva verification uses SKILL functions (subroutines) to perform its verification tasks.
- Put all LVS rules in the `lvsRules()` function.
- Enter the commands in the correct order, because this program processes the commands as it encounters them.
- Enter SKILL functions for parameter reduction and comparison before the ERC and LVS verification commands.
- Store the rules files in the required location. There are no default names for these files, and you can store them in the technology library or anywhere else with any name. If you store the rules in a technology library, the rules file name must contain a dot to distinguish it from other library entries. If you use a standard convention of adding `.rul` to your file names, you can always identify the Diva verification rules files.

**Note:** Unlike DRC/Extract, there is no *if-then-else capability* for ERC and LVS. In addition, ERC and LVS verification commands are not optimized or rearranged.

During ERC and LVS processing, the rules are copied to the run directory you define and are stored with the pre-set names `divaLVS.rul` and `divaERC.rul`.

Sample ERC Rules File

This section contains a sample ERC verification rules file.
ercRules(
  setPower( "vdd!" )
  setGround( "gnd!" )

  /* Identifies vdd! nets as power nets and gnd! nets as ground nets. */
  twoWayPath( "pfet" "S" "D" )
  twoWayPath( "nfet" "S" "D" )

  /* Defines all pfets as having bidirectional current paths from Source to Drain and all nfets as having bidirectional current paths from Source to Drain. */
  checkPullUp( )
  checkPullDown( )

  /* Checks all nets for a path from the net through devices to nets identified as power nets. Then checks all nets for a path from the net through devices to nets identified as ground nets. */
  checkFloatingDevices( )
  checkOneNetDevices( )

  /* Flags all devices with no connection to any part of the circuit. Then flags all devices with only one connection to any part of the circuit. */
  checkFloatingNets( )
  checkOneTerminalNets( )

  /* Flags all nets with no connection to any part of the circuit. Then flags all nets with only one connection to any terminal of a device. */
  checkNotConnected( "pfet" "B" "vdd!" )
  checkConnected( "nfet" "S" "vdd!" )

  /* Flags all devices with the specified terminal not connected to the named net. Then flags all devices when the specified terminal is connected to the named net. */
  reduceDevice( MOS "pfet" )

  /* Reduces the specified device to a logically equivalent gate-level circuit. (You must use setPower and setGround to identify supplies.) */
  reduceDevice( MOS "nfet" logic )
/* Reduces the specified device to a logically equivalent gate-level circuit. Uses keyword logic to signal device types used to form the logic portion of gates. (You must use setPower and setGround to identify supplies.) */

/* The right parenthesis terminates the ercRules. */

Sample LVS Rules Files

This section contains two sample LVS verification rules files.

Sample CMOS Rules File

The following is a sample LVS verification rules file for CMOS devices.

```c
lvsRules(
procedure( compareFet( layPlist, schPlist )

/* The first SKILL procedure, compareFet, compares the width and length of active devices. The compareFet procedure uses two variables, layPlist, which contains the parameter list from the layout device being checked, and schPlist, which contains the parameter list for the schematic device being checked. */

prog( ( )
    if( layPlist->w != nil && schPlist->w != nil then
        if( layPlist->w != schPlist->w then
            sprintf( errorW, "Gate width mismatch: %gu layout to %gu schematic", float( layPlist->w ), float( schPlist->w ) )
            return( errorW )
        )
    )

/* First, compareFet checks that both lists contain the property w and, if they do, compares the values of the properties. If the values are not the same, compareFet returns an error. */

    if( layPlist->l != nil && schPlist->l != nil then
        if( layPlist->l != schPlist->l then
            sprintf( errorL, "Gate length mismatch: %gu layout to %gu schematic", float( layPlist->l ), float( schPlist->l ) )
            return( errorL )
    )
```

procedure( parallelMOS( m1Plist, m2Plist )

/* The next SKILL procedure, parallelMOS, combines the parameters of parallel MOS transistors in the netlist for the parameter comparison done by the preceding procedure. parallelMOS uses two variables, m1Plist, which contains the parameter list from the first device of the pair of devices to be consolidated, and m2Plist, which contains the parameter list for the second device of the pair of devices to be consolidated. */

prog( ( parMos )
   parMos = ncons( nil )
   if( m1Plist->l != nil && m2Plist->l != nil then
      parMos->l = ( m1Plist->l + m2Plist-> l ) / 2.0
   )

   /* First, parallelMOS checks that both lists contain the property l and, if they do, adds the values of the two properties together and divides them by 2. */
   if( m1Plist->w != nil && m2Plist->w != nil then
      parMos->w = m1Plist->w + m2Plist->w
   )

   /* Next, parallelMOS checks that both lists contain the property w and, if they do, adds the values of the two properties together. */
   return( parMos )
) ; prog
) ; parallelMOS

/* The new device created by parallel consolidation contains the properties l and w, set to the values calculated by the procedure. If the properties l or w do not exist, or have no value for the devices being consolidated, the new device does not have the l or w properties. */
Diva Reference
Writing Diva Verification Rules

/* The third part of this file contains the actual iLVS rules for device permutation and parameter comparison. */

permuteDevice( parallel "pfet" parallelMOS )
permuteDevice( parallel "nfet" parallelMOS )
permuteDevice( MOS "pfet" )
permuteDevice( MOS "nfet" )

/* The permute parallel statements reference the procedure parallelMOS to consolidate the parameters of devices being permuted. */

compareDeviceProperty( "pfet" compareFet )
compareDeviceProperty( "nfet" compareFet )
)

/* The compare statements reference the compareFet procedure and cause parameter comparison to take place. */

Sample Bipolar Rules File

The following is a sample LVS verification rules file for bipolar devices.

lvsRules(
procedure( parallelRes( r1Plist, r2Plist )

/* The SKILL procedure parallelRes combines the resistance values of parallel resistors. First, the procedure reads the list of properties from the first resistor in the parallel reduction (r1Plist) and compares that list to the list of properties from the second resistor in the parallel reduction (r2Plist). */

prog( ( parPlist myPort )
parPlist = ncons( nil )
myPort = outfile( "/user/project/LVS/Resistors.val"
"a")
if( r1Plist->r != nil && r2Plist->r != nil then
parPlist->r =
( r1Plist->r * r2Plist->r ) / ( r1Plist->r + r2Plist->r )

/* If parallelRes finds the property r in both lists, it divides the product of r1Plist x r2Plist by the sum of r1Plist + r2Plist. The result of the calculation is returned to the new parallel-resistor device as the value of its property r. */

if( myPort == nil then
printf( "Can’t open parallelRes IO port - myPort") else
fprintf( myPort,
"Parallel Resistor value is %g \n", float(parPlist->r) 
    close( myPort )
)
return( parPlist )
)
procedure( seriesRes( r1Plist, r2Plist )

/* The seriesRes procedure combines the resistance values of series resistors. First, the procedure reads the list of properties from the first resistor in the series reduction (r1Plist) and compares that list to the list of properties from the second resistor in the series reduction (r2Plist). */

prog( ( serPlist myPort )
    serPlist = ncons( nil )
    myPort = outfile( "/user/project/LVS/Resistors.val" "a" )
    if( r1Plist->r != nil && r2Plist->r != nil then
        serPlist->r = ( r1Plist->r + r2Plist->r )
    )

    /* If seriesRes finds the property r in both lists, it then adds (+) the values of the two properties. The result of the calculation is returned to the new series-resistor device as the value of its property r. */

    if( myPort == nil then
        printf( "Can't open seriesRes IO port - myPort") else
        fprintf( myPort,
            "Series Resistor value = %g \n", float( serPlist->r )
        )
        close( myPort )
    )
return( serPlist )
)

procedure( compareRes( layRes, schRes )

/* The compareRes procedure compares the resistance values calculated during the parallel and series consolidation of resistors. First, the procedure reads the list of parameters from the [consolidated] resistor in the layout (layRes) and compares it to the list of parameters from the [consolidated] resistor in the schematic (schRes). */

prog( ( )
    if( layRes->r != nil && schRes->r != nil then
        if( abs( layRes->r - schRes->r ) > 0.02 * schRes->r then
sprintf( error,  
    "Value mismatch: %g ohms (layout) vs %g ohms  
    (schematic)\",  
    float( layRes->r ), float( schRes->r )  
    return( error )

/* If compareRes finds the property r in both lists, it subtracts the  
    value of r in layRes from the value of r in schRes. If the  
    difference between the values is greater than 2%, compareRes  
    returns an error; if not, the resistor values are considered  
    matched. */

return( nil )

/* Parallel and Series Permutation and Parameter Consolidation */

permuteDevice( parallel "npn" )
permuteDevice( parallel "pnp" )
permuteDevice( parallel "resistor" parallelRes )
permuteDevice( series "resistor" seriesRes )

/* Parameter Comparison */

compareDeviceProperty( "resistor" compareRes )


\section*{Creating a Run-Specific File}

You use a run-specific file when you want to specify commands that apply only for a particular verification run. You put these run-specific commands in a separate file that you call when you perform a DRC or an extraction.

You can use the SKILL functions to create a run-specific file:

\begin{table}[h]
\centering
\begin{tabular}{|l|l|}
\hline
\textbf{To...} & \textbf{Use...} \\
\hline
Add text labels to the verification run without adding them to the original graphics data. & layerText \\
Define an area of the circuit to be processed by DRC. & verifyArea \\
\hline
\end{tabular}
\caption{SKILL functions for creating run-specific files}
\end{table}
You can put the run-specific file in any directory that is accessible through normal UNIX file conventions. You then specify the name of this file when you use the menu to set the DRC or extraction options or when you use the ?rsf option in the ivDRC and ivExtract commands.

The following sections describe the SKILL functions listed in the table above.
**changeLabel**

`changeLabel( ( oldLabel newLabel ) ...)`

**Description**

Temporarily changes the name of a label without affecting the database. You can use `changeLabel` as:

- An entry in the run specific file.
- A property on the cell being verified.
- An entry in the verification rules file.

The name change applies only to original labels. If label “a” is changed to “b” and label “b” is changed to “c,” label “a” is never changed to “c.”

The name change applies to labels recognized through the label function and pin names in the `geomConnect` command in the technology file. For labels, the function changes the name of the net to which it is applied. For pin names, the function generates the pin in the extracted and excell cellviews with the new name.

**Note:** The label changes every time you run DRC or extraction with those rules if you use `changeLabel` in the verification rules file. When you use `changeLabel` in the verification rules file, you must place it before any `geomConnect` commands.

**Fields**

`oldLabel newLabel`

Specify two text names, each enclosed in quotation marks, defining the old (existing) label name and its new (replacement) name.

Use the new name in place of the old name (except for the `geomGetTexted` and `geomGetUntexted` commands). The name change is global, affecting all cells in the circuit.

You can specify as many name pairs in one `changeLabel` command as you want.

You can use wildcards only for `oldLabel` following the Diva verification conventions for wildcards. (See Using Wildcards.) The command `changeLabel( "gnd*" "gnd" )` means that both `gnd1` and `gnd2` are changed to `gnd`. You cannot change the name of a label to a name that contains wildcards.
Property

You can also use changeLabel as a property of the cell. Only one changeLabel property is allowed per cell. For flat extraction, place the property on the top-level cell. Diva verification ignores any changeLabel property on lower-level cells. For hierarchical extraction, all cells can have this property, and the label affects only the cell where it was placed.

The property uses this format:

   changeLabel = "oldLabel newLabel [ oldLabel newLabel ] ..."

This example illustrates the changeLabel property.

   changeLabel = VDD vdd! GND gnd!

Example

This example illustrates the changeLabel command.

   changeLabel( ( "VDD" "vdd!" ) ( "GND" "gnd!" ) )

In this case, the original names of VDD and GND are changed to conform to the local conventions of vdd! and gnd!.
**drcLogFile**

```
drcLogFile( file_name [precision [page_width [page_length [line_limit]]]])
```

**Description**

Produces a DRC log file that lists all marker shapes created during the current or previous Diva verification run. The log file is created only in DRC and extraction modes. To generate this file, you must add the `drcLogFile` command to your run-specific file.

**Note:** While running Diva verification DRC, if you get an error message stating that the `drcLogFile` function is only allowed in the run-specific rules (and this tool generates this error message even if you run the function directly in the CIW), it is because you have put the `drcLogFile` function in the rule file (`divaDRC.rul`). The run-specific file is different from the rule file. So create a file with the following in it:

```
drcLogFile(<filename>.log)
```

where `<filename>.log` is the name of the file and can have any name.

Then run Diva verification DRC and in the DRC form turn on the Run-Specific Command File field and enter the name of the above file. Depending on the location of the file, you may have to enter the path along with the file name. Now when you run DRC, the `<filename>.log` is created in the directory you started the tool from. This is true for both the 4.3.4 and 4.4.x releases.

**Fields**

- **file_name**
  - Name of the log file. This file is deleted before any text is written to it, so that existing data in the file is overwritten.

- **precision**
  - Number of digits to the right of the decimal point to be used when the numbers are printed. For example, a value of 2 might cause coordinates to be printed such as 5.25, while a value of 3 might produce 5.252. The default value is 2.

  **Note:** In order to conserve disk space, a number such as 5.000 is always printed as 5, no matter what the precision.

- **page_width**
  - Page width. The default is 80 characters. Lines that exceed 80 characters are broken into multiple lines.
page_length  Page length. The default is 0, which disables this feature. If you use this argument, the system inserts a page header at the top of each page.

line_limit  Specifies the limit at which the program stops printing to the log file. The default is no limit.

Note: If this command is used without arguments, the drcLogFile log is written to the CIW.

Example

drcLogFile( "DRC.log" )
**drcZeroHalo**

drcZeroHalo( status )

**Description**

Sets (or resets) the halo to be used in area processing to zero. This halo is the distance added around the checking area to ensure all interactions and relationships that could affect the data in the area are included.

The halo distance normally used depends on the dimensions in the rules file. For some rule sets, the halo can be very large and can have a serious impact on run time.

**Caution**

*This command can miss true violations, so use it carefully.*

This command is disabled when running in hierarchical or incremental modes. This command overrides the *ivVerify* command -z option for this run.

Placing the *drcZeroHalo* command in the *.cdsinit* file sets the mode for the duration of the interactive session. Used in a run-specific file, this command sets the mode for a single Diva verification invocation.

**Fields**

- **status** Options used for status.
- **t** Sets the status to true and turns on the zero halo option, which sets the halo to a value of zero.
- **nil** Sets the status to false and turns off the zero halo option, which leaves the halo value at its automatic calculated value.

**Examples**

```
drcZeroHalo( t )
drcZeroHalo( nil )
```
**globalLabel**

`globalLabel( layer label [label] )`

**Description**

Treats labels as if they are on a top-level cell even though they actually come from lower level cells. For example, you can use power and ground labels inside subcells for net labeling.

The default purpose for the layer is *drawing*. To specify other purposes, you can use this command.

```
   globalLabel( (layer purpose) label )
```

The label is always considered to be the *drawing* purpose, regardless of the original purpose of the text.

**Note:** You can also use this command in the verification rules file. When you use it in a verification rules file, you must place it before any `geomConnect` command. Other commands that process text, such as `geomGetTexted`, are also affected by this command.

**Prerequisite**

The `globalLabel` command must precede the function that uses the labels. If you place it after `geomConnect`, it has no effect on the net names.

**Note:** Other commands that process text, such as `geomGetTexted`, are also affected by this command.

**Fields**

- **layer**
  
  Original graphics layer containing the label you want treated as if it is on a top-level cell. You can define the `textLayer` in two formats:

  - "layer"
    
    The label is accessed from the *drawing* purpose and remains on the *drawing* purpose.
If you specify a purpose, such as pin or text, the labels are accessed only from that purpose and remain on that purpose. If you specify the purpose all, the labels are accessed from all purposes and remain on their original purposes.

**Examples**

In this example, power and ground are properly labeled even though there are no vdd! or gnd! labels at the top-level cell.

```plaintext
globalLabel( "met1_lab" "vdd!" "gnd!" )
geomConnect(
    via( via metal1 metal2 )
    via( cus diff poly metal1 )
    label( "met1_lab" metal1 )
)
```

If two `globalLabel` commands apply to the same layer, you can combine them into a single command. For example,

```plaintext
globalLabel( "l1" "a" )
globalLabel( "l1" "b" )
```

is the same as

```plaintext
globalLabel( "l1" "a" "b" )
```
groundNet

groundNet ( "net _name" )

Description

Defines a specific net name to be used by programs that make connections to the circuit ground net. If you specify this command, the net name it defines is used instead of the default global ground net, which has the name “0!”.

This command creates the defined net in the extracted circuit regardless of whether the net name exists in the original layout.

**Note:** The net is defined as a global net, and all nets with the same name are assumed to be joined (as if you defined a `joinableNet` command). This means any open circuits in the net are not seen in LVS processing.

You can also use this command in the verification rules file. When you use it in a verification rules file, you must place it before any `geomConnect` command.

If you specify this command more than once, the first command is accepted. Diva verification ignores the remaining occurrences of the command. If you use this command in the RSF, it takes precedence over this same command in the rules file.

When you use this command, a property called `ivGroundNetName` is added to the extracted and abstract cellviews. The content of the property is a string whose value is the defined net name.

Fields

"net_name" The name of the ground net. Enclose the net name in double quotation marks.

Example

You can define a ground net as follows:

```plaintext
groundNet( "gnd" )
```

The following example shows parasitic measurement:

```plaintext
x = measureParasitic( area metal one_net )
saveParasitic( x "PLUS" "MINUS" "c" "capacitor" )
```
If you do not specify the *groundNet* command, a parasitic capacitor is created for each area of metal, with one terminal of the capacitor connected to the net to which the metal belongs, and the other terminal connected to “0!”. If you specify the *groundNet* command, the second terminal of the capacitor is connected to the net specified in the command (in this case, *gnd*).
**joinableNet**

\[
\text{joinableNet( } [\text{net } \ldots] \text{ )}
\]

**Description**

Specifies that selected nets with the same name are to be considered as a single net, not an error. You can use `joinableNet` as:

- An entry in the run specific file.
- A property on the cell being verified.
- An entry in the verification rules file.

Any two or more nets with the same label that are not specified as joined with a `joinableNet` command are flagged by a warning.

**Note:** The nets you selected will be joined every time you run DRC or extraction with those rules if you use `joinableNet` in the verification rules file. When you use `joinableNet` in the verification rules file, you must place it before any `geomConnect` commands.

**Fields**

- **net**: Any number of net names enclosed in quotation marks. For each name, all separate nets with that name are considered part of a single net. If you do not specify a list, DRC joins all nets.

You can use wildcard characters following Diva verification conventions for using wildcards. (See the section, Using Wildcards.) Only nets with the same name are joined. For example, if you use wildcards to specify `gnd*`, all nets labeled `gnd1` are joined together, and all nets labeled `gnd2` are joined together. Nets labeled `gnd1` are not joined to nets labeled `gnd2`.

**Examples**

This example illustrates the `joinableNet` command.

\[
\text{joinableNet( "vdd!" "gnd!" "clock1" ) } \\
\text{joinableNet( )}
\]

This example uses the `joinableNet` command with the `changeLabel` command to join nets with different names.

\[
\text{changeLabel(( "vdd1" "vdd" ) ( "vdd2" "vdd" ))}
\]
The separate nets `vdd1` and `vdd2` are both renamed to `vdd` with the `changeLabel` command, and then all nets with the name `vdd` are specified to be the same net with the `joinableNet` command.

**Property**

You can also add `joinableNet` as a property of the cell. Only one `joinableNet` property is allowed per cell. For flat extraction, place the property on the top-level cell. Any `joinableNet` property on lower level cells is ignored. For hierarchical extraction, all cells can have this property, and the property affects only the cells on which it was placed.

The property uses this format:

```
joinableNet = "net_name1 [ net_name2 ] ..."
```

This example illustrates the `joinableNet` property.

```
joinableNet = vdd! gnd! clock1
```
layerText

layerText( textLayer xy text [cell] ...)

Description

Temporarily adds text labels to a text layer. The labels appear on the drawing purpose at one user unit in length and in standard orientation.

Fields

textLayer Layer on which you want to place the label. The labels appear on the drawing purpose at one user unit in height and in standard orientation.

You can define the textLayer in two formats:

"layer" Places text on the drawing purpose.

("layer" "purpose") Places text on the purpose you specify. You cannot specify the purpose all for this layer.

xy x and y coordinates where you want to place the label, expressed in user units, such as microns. You can use floating point numbers or integers, separated by a colon.

text Label you want to place. Use a character string enclosed in quotation marks.

cell Optional cell name defining the name of the cell in which the text is placed. It can have the form

[cellName [viewName [libName ]]]

You must place the names in quotation marks in the following format:

"cell"
"cell view"
"cell view lib"
If you do not use the *cell* option, the label is placed in the top-level cell. The default view and library are those of the top-level cell.

If you specify a cellview combination that does not exist, the program ignores the *layerText* command and issues a warning message. A run generates a maximum of 100 warning messages.

**Examples**

This example places the label *clock_1* on the text layer at coordinates 34:25.

```plaintext
layerText( "text" 34:25 "clock_1")
```

This example uses a SKILL function to loop through the generation of 32 names for a bus. The resultant names become *bit_0* to *bit_31* in a vertical column at *x* = 100, with *y* varying from 200 to 386 in 6 unit steps.

```plaintext
prog( ( i y label )
  for( i 0 31
    y = 200 + 6 * i
    sprintf( label "bit_%d" i )
    layerText( "metalText" 100:y label )
  )
)
```
**lvsOption**

*lvsOption* (keyword value)

**Description**

*lvsOption* lets you control specific options that affect basic LVS algorithms.

**Prerequisites**

This function must only be used within *lvsRules()*. 

**Arguments**

- **REDUCE_PSEUDO_PARALLEL**
  - Controls the reduction of pseudoparallel devices.
  - **TRUE**: Run pseudoparallel reduction in this LVS run.
  - **FALSE**: Do not run pseudoparallel reduction in this LVS run.

The default value for REDUCE_PSEUDO_PARALLEL is TRUE. Only one value for REDUCE_PSEUDO_PARALLEL may be specified for the entire run. For example you cannot turn pseudoparallel reduction on, execute several LVS commands, and then turn pseudoparallel reduction off. The value for REDUCE_PSEUDO_PARALLEL in the final *lvsOption* command will be applied to the entire *lvsRules()* run.

**Example**

```plaintext
lvsRules(
    ....
    lvsOption(REDUCE_PSEUDO_PARALLEL FALSE)
    permuteDevice(MOS nmos)
    permuteDevice(parallel nmos someFunc)
    ....
)
```
macroCellFile

macroCellFile( "filename" )

Description

Provides a file of cell names that controls whether a cell becomes a macro cell or not.

You can use this file in addition to the ivCellType property on a cell. However, the values you define for each cell using the macroCellFile command override the ivCellType property on the cell.

Fields

"filename"  The name or path, in quotation marks, of a text file containing one line per cell with the following format:

    # libName cellName viewName value

    # Indicates a comment. If you do not specify # on a line, the line is active. If you specify # on a line, that line is ignored. Blank lines are also ignored.

    libName  Name of the library from which the cell is accessed.

    cellName  Name of the cell.

    viewName  Cellview name.

You can define any of the names using wildcard characters following the this tool's conventions for using wildcards. See the “Using Wildcards” section for more information about wildcards.

Value

This argument can have the following values:

macro  Forces the cell to become a macro cell.

graphic  Forces the cell not to become a macro cell.

none  Ignores the ivCellType property and lets the cell become a macro or not based on its prCellType property.
Examples

macroCellFile("macrocells")
macroCellFile("/tmp/mydata/cells")

The following examples illustrate possible file contents:

```
   myLib myMacro     layout macro
   myLib nonMacro*  routed graphic
   #  myLib anyCell  * none
```
verifyArea

verifyArea( ( box [ hole ] ...) ...)  

Description

Defines those areas of the circuit you want checked. You can define the area as one or more rectangles. You can optionally specify that each rectangle has one or more holes in it.

The area processed by the DRC program is the area of each rectangle minus the area of the holes.

Fields

box Two pairs of coordinates enclosed in parentheses, defining the bottom left and top right of a rectangular area to be checked. Separate the two pairs by at least one space. Each coordinate pair consists of an x and y value in user units, separated by a colon. For example,

( 10:100 5000:2500 )

hole Pairs of coordinates defining holes in the rectangle. You place the hole coordinate pairs after the box pair but within the same set of parentheses. The format for each hole coordinate pair is the same as for the original box. For example,

( 10:100 5000:2500 50:500 1000:1250 )

Examples

This example illustrates a simple rectangular verification area.

verifyArea( ( 0:0 1000:1400 ) )

This example illustrates a simple rectangular verification area with a single cutout.

verifyArea( ( 0:0 1000:1400 50:100 356:420 ) )

This example illustrates two simple rectangular verification areas.

verifyArea( ( 0:0 1000:1400 ) ( 2000: 2300 4000: 5000 ) )

This example illustrates a combination of areas and holes.

Sample Run-Specific File

This is a sample run-specific file.

/* The first command checks a rectangular area with a lower left coordinate of (10:100) and an upper right coordinate of (500:250). The next two coordinate pairs establish a (200:120) by (220:150) hole in the rectangle that is not checked. */
verifyArea( (10:100 500:250 200:120 220:150) )

/* This command places the string out1 on the text layer at the xy location (30:50). It also specifies the cell sreg in which the text label is to be placed. */
layerText( "text" 30:50 "out1" "sreg" )

/* This command changes all VDD and VCC labels to vdd! for the current run only. */
changeLabel( ("VDD "vdd!" ) ( "VCC" "vdd!" ) )

/* This command joins all disjoint nets of the same name. */
joinableNet( )

/* This command brings all labels to the top-level of the current design. */
globalLabel( "text" "vdd!" "gnd!" )

/* The last command lists all marker shapes created during the current or previous Diva run. */
drcLogFile( "DRC.log" )
Layer Processing Concepts

This section describes the concepts and terminology used when discussing layers and shapes in Diva verification.

Layers

Every layer is subdivided into purposes. The only time you reference a purpose in this product is when you use the `geomGetPurpose` command to select shapes.

The three groups of layers used in this tool system are as follows:

- Original graphics layers
- Derived layers
- Connected layers

Original Graphics Layers

Layer names you specify in the graphics editor are called original graphics layers. You enter the layer name as a text string enclosed in double quotation marks. The following are some typical original graphics layer names:

"polysilicon" "metal" "diffusion"

Original graphics layers are subdivided into layer purposes such as drawing, pin, or text. Some commands handle specific layer purposes. You can find details of these capabilities in the command descriptions. If a command does not reference layer purposes, the command processes all layer purposes except boundary by default. The boundary purpose is excluded because it contains a single polygon per cell defining the boundary of that layer in that cell. If this polygon is processed in conjunction with the shapes on other purposes of that layer, those shapes would be eliminated by the boundary polygon.

Note: Do not use any of the reserved layers used by the Virtuoso® layout editor (such as instance on the drawing purpose, for example) to create graphic information.
Derived Layers

Many verification commands generate results that can be processed the same way as an original graphics layer. These results are called derived layers.

There are significant differences between original graphics layers and derived layers. A derived layer is really a symbolic name associated with the data that forms the layer. You can manipulate a derived layer the same way as a symbol.

For example, consider the simple function

\[ \text{gate} = \text{geomAnd}( \text{"polysilicon"} \text{ "diffusion"} ) \]

This creates a derived layer called gate from original layers of polysilicon and diffusion.

Another example is

\[ \text{x} = \text{gate} \]

The symbol x and the symbol gate both refer to the same data generated by the geomAnd command. The layer name and the layer data are separate, and you can assign names to the data as required.

The derived layer names must conform to SKILL naming syntax conventions. The most important convention is that a name starting with a numeric (0 though 9) must be preceded by a \"\.". For detailed information about SKILL naming conventions, refer to the SKILL Language User Guide.

Another difference between original and derived layers is that derived layers do not exist in the graphics database unless you specifically save them there.

Derived layers can be empty. Empty layers are processed the same way as any other derived layer. In some circumstances, this tool automatically creates empty layers. For example, if you define an if-then-else branch in the rules, create a layer in the if branch but not in the else branch, and do not create that layer anywhere else, this tool creates an empty version of that layer so that after exiting the if branch, it consistently has that layer. You can also generate empty layers for other uses by using the geomEmpty command.

Connected Layers

Layers processed with the geomConnect command are called connected layers. This tool assigns net numbers to the connected layers and uses these net numbers in various places in the system.
Normally, derived layers generated from connected layers do not retain net information. However, you can use Relational Selection commands to generate layers that retain net information. You can also use `geomAnd` and `geomAndNot`. Refer to these sections later in this chapter.

**Shapes**

A shape consists of edges linked together to form a closed area. All shapes have an inside and an outside. In Diva verification, a shape is the name given to any graphic entity, such as a polygon or rectangle. *Shape* and *polygon* are often used to mean the same thing.

The following figure illustrates three shapes and their types. You can refer to each shape as a *shape* or a *polygon*.

![Shapes](image)

- Square
- Rectangle
- Polygon

**Raw and Merged Data Format**

Layer data format can be described as either *raw* or *merged*. When this product first reads a layer in the graphics editor database, the layer is in *raw* format. This means that individual shapes on the same layer can overlap and butt each other, and that the system retains internal construction edges such as the exit edges from donut holes.

Most layer processing functions generate *merged* data. This means that this tool merges overlapping and butting shapes on the same layer into single shapes and also removes construction edges. The resulting shapes have edges that form an interface between the inside and outside of the shape. If you use merged data during verification, processing is simplified and spurious errors are eliminated.

For some applications, you might want to process the data in raw format. To process data in raw format, you can use the DRC *raw* modifier.
The following figure illustrates the difference between raw and merged data format.

**Polygon and Edge Data Format**

With derived layers, the format of the layer data can be described as either *polygon* or *edge*. As discussed previously, *polygon* and *shape* are often used to mean the same thing. All original graphics layers are in polygon format.

Some layers can be in edge format. This means the data is a group of separate edges that originally belonged to a polygon. For example, you can direct DRC to flag errors as edges. DRC extracts only those segments of the original polygon edges that violate the design rule, and stores those segments on the output layer.

The edges on the edge layers retain all the information about the polygon from which they came, such as which side of the edge was the inside of the polygon and which side was the outside. You can use these edges in places where you would use polygons, such as DRC.

Although the edges for a polygon can appear on an edge layer, this tool treats the edges as separate edges and does not form a polygon. You cannot use edges to form polygons, although you can size edges using the *edges* option in the *geomSize* command.

There are some limitations to using edge format. For example, you cannot use edge layers with the *geomEnclose* command because an edge cannot enclose another shape.

The following figure illustrates the concepts of polygon and edge formats.
Conics and Paths

Conics and paths are any shapes you create that do not have a sequence of outside edges. The graphics editor recognizes conics as polygons and stores them as polygons in the database. However, the graphics editor does not store paths as polygons. Instead, this tool converts paths to polygons when they are brought up from the database.

Because of this, when this tool converts conics and paths to polygons, there might be inaccuracies during DRC checks. Regardless of how many edges you use to represent a conic, the spacing is always different from the edge spacing. The following figure illustrates the problem.

With paths, the angle of a path center line combined with its defined width might define polygon vertex points that lie off of the grid being used to represent the data.

Edge Boolean Functions

The results of logical operations on shapes are well defined. However the results of logical operations such as and and or on edges are not so clear as there are multiple possible interpretations. Following is the definition of how logical edge operations will be interpreted in Diva verification in version 4.4. This is a change from the previous definition (in versions 4.3.4 and earlier).

Wherever possible the results of edge boolean operations are based on the result of the equivalent shape boolean operations. For example, an and function between shapes will not have a result if the shapes do not overlap. Similarly, the and function between edges will not have a result if the edges came from shapes which did not overlap. An edge xor is considered to be the same as an edge or minus an edge and.
Edges resulting from boolean operation will carry the serial, node and direction of the original edges from which they were derived. Where both layers in a command contribute to a single resultant edge, this information will come from the first layer specified in the command.

The following graphic illustrates two-layer and single-layer boolean operations.

**Two Layer Operations**

- **Original shapes**
  - B A B
  - A and B
  - A or B
  - A xor B
  - A not B

- **A not B**
  - B not A

**Single Layer Operations**

- **Original shapes**
  - A A A
  - and A

- **or A**
  - xor A
Invalid Graphics Data

There are some shapes that this tool might interpret different from what you want. These shapes are referred to as *invalid* graphics data. They include the following:

- Self-intersecting shapes
- Zero-width rectangles
- Zero-width polygons
- Lines, zero-width paths, and arcs
- Text

Self-Intersecting Shapes

Many shapes can be drawn that violate the concept of “inside” and “outside” of a shape. These shapes are called *self-intersecting shapes*. This software product accepts self-intersecting shapes and tries to generate valid shapes during the merging process. If you use the DRC `raw` modifier, this tool won’t try to merge the shapes.

The following figure illustrates five shapes, four of which are self-intersecting. Note that some edges are drawn close together rather than coincident, so that the point at which the edges cross is not hidden. Also, the shapes are not shaded, since shading depends upon the concept of “inside” and “outside.”

The self-intersecting shapes might be the shapes you want. You can use the `dubiousData` command to flag self-intersecting shapes so you can decide whether they are invalid or not.
Self-Intersecting Paths

This tool will detect, using the same rule as above, a path which has an outline which qualifies as a self-intersecting polygon. You should note that the path outline checked is the same as the outline displayed by the layout editor.

Zero-Width Rectangles

This tool processes zero-width rectangles as raw data but deletes them during the merging process. Use the dubiousData command to flag these shapes.

Zero-Width Polygons

This tool processes zero-width polygons as raw data. When the merging process is done, Diva verification deletes any sections of the polygon that have zero width, leaving valid polygon pieces.

Lines, Zero-Width Paths, and Arcs

This tool flags lines, zero-width paths, and arcs as warnings if you use the dubiousData command. They do not appear in raw or merged data.

Text

This tool flags text as warnings if you use the dubiousData command. Text does not appear in raw or merged data.

Exclusive Selection

You can specify the “exclusive” option for all relational selection functions.

- If you don’t specify the “exclusive” option, a shape on the input layer is selected if it interacts with any shape on the related layer in the manner described by the selection command. It does not matter if there are other shapes on the related layer interacting in other ways with the input layer.

- If you specify the “exclusive” option, a shape on the input layer is selected only if every interaction with shapes on the related layer are in the manner described by the selection command.
The relationship being checked for the selection includes both the relationship defined by the command keyword and the \textit{sameNet} and \textit{diffNet} options. For example, if you specify
\begin{verbatim}
geomCoincident( A B sameNet exclusive)
\end{verbatim}
with the following conditions
\begin{itemize}
  \item A shape on layer A has 2 shapes of layer B interacting with it.
  \item Both shapes on layer B are coincident with the shape on layer A.
  \item The first shape on layer B is on the same net as the shape on layer A.
  \item The second shape on layer B is on a different net than the shape on layer A.
\end{itemize}
The shape on layer A is not selected because the second shape on layer B violates the exclusive requirement as it is on a different net.

You might find the exclusive concept confusing when dealing with relationships that appear to be inherently exclusive, such as \textit{geomButtOnly}. There is a significant difference between \textit{geomButtOnly} with and without the exclusive option.

\begin{itemize}
  \item The command \textit{geomButtOnly( A B )} selects any shape on layer A that interacts with one or more shapes on layer B with a butting-only relationship. The butting-only relationship refers to the way in which any individual shape on layer B relates to the shape on layer A. It does not matter if other shapes on layer B relate to the shape on layer A with some other relationship.
  \item The command \textit{geomButtOnly( A B exclusive )} selects a shape on layer A only if all shapes on layer B that interact with it have a butting-only relationship.
\end{itemize}

\textbf{Data Integrity Checks}

Data integrity checks let you evaluate the layout view for grid or shape errors.
dubiousData

dubiousData( inLayer [message] )

Description

The dubiousData function highlights on the marker layer improperly formed shapes in the database. This function highlights any shape that:

■ is self-intersecting,
■ is a path with a self-intersecting outline,
■ has labels (on the selected layer),
■ has zero width rectangles,
■ has zero width paths, or
■ has a section of zero width formed by two edges running coincident for a finite length.

The dubiousData function does NOT flag shapes that have a zero width formed by a point contact.

Fields

inLayer Input layer name. You can specify either a layer name in quotes, or a layer purpose pair. A layer purpose pair is defined in the form

( "layer" "purpose" )

The default layer purpose is “all.”

message Optional text string that is used as the message associated with errors. You see this text string when you use the explain command. If you do not define this message, the program uses the complete command as the message.

Examples

The following example illustrates testing polysilicon shapes with the dubiousData function.

dubiousData( "poly" "badly formed poly polygons" )

The following examples illustrate testing metal shapes with the dubiousData function.
dubiousData( geomCat( "metal1" "metal2" "metal3" )
"badly formed metal polygons" )

dubiousData( ("metal1" "drawing" ) "badly formed metal polygons" )
offGrid

offGrid( inLayer grid [size] [raw] [message] )

Description

You can use the offGrid check to find vertices of original graphics layers that are not on a specified grid. This tool places crossover offgrid vertices on the marker layer in the layout view.

All vertices on all shapes are checked unless those shapes are invalid. Conics and paths are expanded into polygons prior to the off-grid checking.

Specifying a list of layers using the geomCat command is not the same as specifying a number of offGrid commands, unless the raw option is used. The raw option prevents the layers from being merged and internal vertices from being removed.

Fields

inLayer Input layer name. This layer can be either a derived layer or an original graphics layer.

grid Floating point number in user units that defines the size of the grid. For example, a value of 0.25 microns highlights all vertices not on a quarter-micron grid.

size Floating point number that overrides the default size of the cross that the program uses to flag the position of any vertex not on grid. The default size of the cross is 1 user unit.

raw Optional keyword raw so the program does not merge an original graphics input layer before the check.

The raw option has no effect on derived layers because they are always merged by the program.

When the program merges data, some off-grid points move back onto the grid. Off-grid points in one shape that overlap another shape are removed by the program during merging.

message You can specify a text string in quotes as the message associated with errors. You see this text string when you use the
explain command. If message is not defined, the complete command is used as the message.

Examples

The following example illustrates the offGrid command.

offGrid( "metal" 0.25 "off grid metal" )
offGrid( "poly" 0.10 raw )

Logical Functions

Logical functions are used to generate layers using derived or original graphics layers.
**geomAnd**

```
[outLayer=] geomAnd( inLayer1 [inLayer2] )
```

**Description**

This function generates new shapes from the overlap of original shapes on the input layers. `geomAnd` works with one or two input layers and with polygons or edges.

- For two-layer polygon input, it generates the areas common to both input layers.
- For one-layer polygon input, it generates the areas where two or more shapes on that layer overlap.
- For two-layer edge input, it generates new edge segments where original edges from both layers are coincident and in the same direction.
- For one-layer edge input, it generates the new edge segment where multiple input edges are coincident and in the same direction.

If the first layer referenced in the command is connected, its net numbers are propagated to the output layer. A layer is connected if it has allocated net numbers through a `geomConnect` command, or any function that propagates node numbers.

```
polsrc = geomAnd( poly srcd )
```

If *poly* is connected, the *polsrc* layer is allocated its node numbers.
The following figure illustrates a polygon and an edge `geomAnd`.

![Original polygons](image1)

**Fields**

- **outLayer**: Layer name for the `geomAnd` result.
- **inLayerN**: Input layer name(s). You can specify either a derived layer or a graphics layer. One or two layers can be specified.

**Examples**

The following example illustrates the `geomAnd` of two original layers.

```
gate = geomAnd( "poly" "diff" )
```

The following example illustrates the `self and` of an original layer.

```
b_error = geomAnd( "boundary" )
```

The following example illustrates the `and` of two derived layers.

```
well_c = geomAnd( well pplus )
```
**geomAndNot**

```plaintext
[outLayer=] geomAndNot( inLayer1 inLayer2 )
```

**Description**

The `geomAndNot` function generates new shapes from areas of the first input layer that do not overlap any area of the second input layer. The `geomAndNot` function works on polygons or edge layers.

- When you specify polygon layer input, the program generates areas of the first layer not over the second layer.
- When one or more of the input layers is an edge layer, the program generates edge segments for edges of the first input layer except where original edges from both layers are coincident and in the same direction.

If the first layer referenced in the command is connected, its net numbers are propagated to the output layer. A layer is connected if it has allocated net numbers through a `geomConnect` command, or any function that propagates node numbers.

```plaintext
nminus = geomAndNot( diff, well )
```

If `diff` is connected, the `nminus` layer is allocated its node numbers.

The following figure illustrates polygon and edge `geomAndNot` functions.
Fields

outLayer
Layer name for the geomAndNot result.

inLayer1  inLayer2
Input layer names. You can specify either a derived layer name or a graphics layer name. Two layers must be specified.

Examples

The following example illustrates the geomAndNot of two original layers.

diffn = geomAndNot("diff" "poly")

The following example illustrates the geomAndNot of two derived layers.

sub_c = geomAndNot(nplus well)
Diva Reference
Layer Processing Concepts

geomCat

[outLayer=] geomCat( inlayer1 [inLayerN] )
[outLayer=] geomCat( inlayer1 [inLayer...] )
[outLayer=] geomCat( inlayerN... )

Description
The geomCat function combines shapes on the input layers without merging those derived polygon shapes.

Fields
outLayer The resultant unmerged derived polygon layer. This layer consists of the unmerged shapes from all input layers.
inLayerN Input layer name. You can specify any number and combination of derived layer names or graphics layer names.

Examples
The following example combines original layers.
metal = geomCat( "metal1" "metal2" "metal3" )
The following example converts an original layer into a derived layer.
poly = geomCat( "poly" )
The following example combines derived layers. Note that the results are not merged.
gates = geomCat( "ngate" "pgate" )
**geomNot**

[outLayer=] geomNot( inLayer )

**Description**

The *geomNot* function generates new shapes by inverting the original shapes. It is the equivalent of a *geomAndNot* between a layer and a single rectangle encompassing the area being processed. The *geomNot* function works on one polygon layer only.

The following figure illustrates a polygon *geomNot*.

![Original polygon vs. geomNot result](image)

**Fields**

- **outLayer**
  - Layer name for the *geomNot* result.

- **inLayer**
  - Input layer name. You can specify either a derived layer name or a graphics layer name.

**Example**

The following example illustrates the *geomNot* of an original layer.

```plaintext
ntub = geomNot( "pwell" )
```
Diva Reference
Layer Processing Concepts

geomOr

[outLayer] geomOr( inLayer1 [inLayerN] )
[outLayer] geomOr( inLayer [inLayer...] )
[outLayer] geomOr( inLayerN... )

Description

Use the geomOr function to merge all shapes on polygon or edge input layers.

When you merge polygons, you can specify one or more input layers. For polygon layers, a merge generates polygons whose area encompasses all the areas from all the layers and removes all the edges internal to the polygons.

When one of the input layers is an edge layer, the resulting merged layer is an edge layer. For edge layers, a merge generates edges in which multiple segments having coincidence or colinearity are combined into single segments.

The following figure illustrates how polygon and edges merge.

Fields

outLayer

Output layer name.
inLayerN

Input layer names. You can specify any number and combination of derived layer names or graphics layer names.

Examples

The following example merges multiple original layers.

poly = geomOr( "poly1" "poly2" )

The following example merges a single input layer.

metal = geomOr( "metal" )

The following example performs a geomOr on derived layers.

pnpref = geomOr( emitosize collector )
geomXor

\[
\text{geomXor}([\text{outLayer=}]) \text{ geomXor( inLayer1 [inLayer2] )}
\]

Description

The `geomXor` function generates new shapes from those portions of either input layer that do not overlap the other layer.

- When you specify one polygon input layer, the program generates areas of the layer that are not overlapping other areas of the same layer.
- When you specify two polygon input layers, the program generates areas of both layers that are not overlapping areas of the other layer.
- When you specify two input layers and one or more is an edge layer, the program generates edge segments for each of the input edges, except where original edges from both layers are coincident and in the same direction.
- When you specify a single edge layer input, the program generates edge segments for each of the input edges, except where multiple input edges are coincident and in the same direction.

The following figure illustrates a polygon or edge `geomXor`.

![Original polygons and geomXor result](image1)

![Original edges shown as polygons and geomXor result](image2)
Fields

outLayer
Layer name for the _geomXor_ result.

inLayerN
Input layer name. You can specify either a derived layer name or a graphics layer name. One or two layers can be specified.

Examples

The following example illustrates the _exclusive or_ of two original layers.

```
cap1 = geomXor( "metal1" "metal2" )
```

The following example illustrates the _self-exclusive or_ of an original layer.

```
open_bound = geomXor( "boundary" )
```

The following example illustrates the _exclusive or_ of two derived layers.

```
cap2 = geomXor( diffn1 diffn2 )
```

Relational Selection Functions

Relational selection functions select polygons from an input layer based on the relationship between that layer and another layer. Relational selection functions maintain net information.
**geomAvoiding**

```plaintext
outLayer = geomAvoiding( inLayer1 inLayer2 )
```

**Description**

The `geomAvoiding` function selects shapes on `inLayer1` that are completely outside and have no butting with shapes on `inLayer2`.

All select functions maintain net numbers of shapes in the output layer.

**Prerequisites**

The following figure illustrates a `geomAvoiding` selection and its differences from `geomOutside`.

![Diagram showing `geomAvoiding` selection](image)

**Fields**

- **outLayer**
  - Name for the `geomAvoiding` output.

- **inLayer1 inLayer2**
  - Input layer names. You can specify either derived layer names or graphics layer names. You must specify two layers.
**geomButting**

```plaintext
[outLayer=] geomButting(  
    inLayer1 inLayer2 [connection] [limits] [exclusive] )
```

**Description**

The `geomButting` function selects shapes on the first layer that abut shapes on the second layer.

Butting is edge coincidence when the areas of the shapes *at the abutment* do not overlap. A first layer shape is butting if any of its edges abut edges of a second layer shape. A butting shape is not affected by other geometric relationships. A first layer shape can be butting, overlapping, and coincident with a single second layer shape and still be classified by the Diva verification tool as butting.

All select functions maintain net numbers of shapes in the output layer.

**Prerequisites**

If you use the `connection` modifier when using `geomButting`, the input layers you specify must be connected layers. The following figure illustrates a `geomButting` selection.

![Diagram showing geomButting selection](image)

Shapes marked “B” are butting.

```
layer1 layer2
```

**Fields**

- `outLayer` Name for the `geomButting` output.
**inLayer1 inLayer2**

Input layer names. You can specify either a derived layer name or a graphics layer name. Two layers must be specified.

**connection**

Optional command modifier. The options are as follows:

- **sameNet**
  Consider and count butting only when the abutted shapes are on the same electrical net.

- **diffNet**
  Consider and count butting only when the abutted shapes are on different electrical nets.

**limits**

Optional qualifier you can use to limit the number of separate shapes on `inLayer2`. If the program finds that the number of shapes on `inLayer2` with the required relationship to `inLayer1` is within the limits specified, then it selects the shape on `inLayer1`.

You specify the limits qualifier as an absolute limit or range using one of the following keywords:

- **keep**
  Select the shape on `inLayer1` if the number of separate shapes on `inLayer2` having the required relationship falls within the numeric range you specify.

- **ignore**
  Reject the shape on `inLayer1` if the number of separate shapes on `inLayer2` having the required relationship falls within the numeric range you specify.

Operators you can use with the keywords are as follows:

- `<`
- `<=`
- `>`
- `>=`
- `==`

You can specify limits by combining a keyword with one or more operators and numbers using normal mathematical conventions. Examples of these combinations are as follows:

- `keep == 6`
- `ignore < 2`
A shape on \textit{inLayer1} with no shape on \textit{inLayer2} having the required relationship is not selected, even if the limits you specify encompass the value 0.

\textbf{exclusive} Optional modifier you can use to select shapes on \textit{inLayer1} that conform to the required relationship but that do not have any other relationship with other shapes on \textit{inLayer2}.

When you specify the \textit{sameNet} or \textit{diffNet} options, the shape on \textit{inLayer1} is not selected if shapes on \textit{inLayer2} have the required relationship but do not have the required net connections.

\textbf{Examples}

The following examples illustrate how you can select shapes using \textit{geomButting} with various options.

\begin{verbatim}
badgate = geomButting( gate diffn 2 < ignore < 4 )
pulldown = geomButting( poly diffn sameNet )
\end{verbatim}

The following examples illustrate how connectivity information can be passed through derived layers.

\begin{verbatim}
geomConnect( via( a b c )
e = geomStraddle( b f )
g = geomButting( e c sameNet )
\end{verbatim}
geomButtOnly

[outLayer=] geomButtOnly(
    inLayer1 inLayer2 [connection] [limits] [exclusive] )

Description

The geomButtOnly function selects shapes on inLayer1 that butt shapes on inLayer2. To be considered, the shapes on inLayer2 must be completely outside the shape on inLayer1. The relationship between a shape on inLayer1 and a shape on inLayer2 must be butting only. Butting is edge-to-edge coincidence, where the shapes do not overlap at the coincident edge.

All select functions maintain net numbers of shapes in the output layer.

Prerequisites

If you use the connection modifier when using geomButtOnly, the input layers you specify must be connected layers.

The following figure illustrates a geomButtOnly selection.

![geomButtOnly selection diagram]

Fields

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>outLayer</td>
<td>Name for the geomButtOnly output.</td>
</tr>
<tr>
<td>inLayer1 inLayer2</td>
<td>Input layer names. You can specify either derived layer names or graphics layer names. You must specify two layers.</td>
</tr>
</tbody>
</table>
connection  Command modifier that can take the following forms:

sameNet  Consider and count only butting when the interacting shapes are on the same electrical net.

diffNet  Consider and count only butting when the interacting shapes are on different electrical nets.

limits  Optional qualifier you can use to limit the number of separate shapes on inLayer2. If the program finds that the number of shapes on inLayer2 with the required relationship to inLayer1 is within the limits specified, it selects the shape on inLayer1.

You specify the limits qualifier as an absolute limit or range using one of the following keywords:

keep  Select the shape on inLayer1 if the number of separate shapes on inLayer2 having the required relationship is within the numeric range you specify.

ignore  Reject the shape on inLayer1 if the number of separate shapes on inLayer2 having the required relationship is within the numeric range you specify.

Operators you can use with the keywords are as follows:

<  
<=  
>  
>=  
==  

You can specify limits by combining a keyword with one or more operators and numbers using normal mathematical conventions. Examples of these combinations are as follows:

keep == 6
ignore < 2
2 < keep < 5
3 <= ignore <= 7
A shape on \textit{inLayer1} with no shape on \textit{inLayer2} having the required relationship is not selected even if the limits you specify encompass the value 0.

\textbf{exclusive}  

Optional modifier you can use to select shapes on \textit{inLayer1} that conform to the required relationship but that do not have any other relationship with other shapes on \textit{inLayer2}.

When you specify the \textit{sameNet} or \textit{diffNet} options, the shape on \textit{inLayer1} is not selected if shapes on \textit{inLayer2} have the required relationship but do not have the required net connections.
**geomButtOrCoin**

```plaintext
[outLayer=] geomButtOrCoin( inLayer1 inLayer2 [connection] [limits]
[exclusive] )
```

**Description**

The `geomButtOrCoin` function selects shapes on the first layer that abut or are coincident with shapes on the second layer.

Butting and coincidence are defined as any edge-to-edge coincidence, regardless of whether the shapes overlap at the point of coincidence.

All select functions maintain net numbers of shapes in the output layer.

This is a combination of the functions `geomButting` and `geomCoincident`.

**Prerequisites**

If you use the `connection` modifier when using `geomButtOrCoin`, the input layers you specify must be connected layers. The following figure illustrates `geomButtOrCoin` selection.

![Diagram showing geomButtOrCoin selection](image)

Shapes marked “B” and/or “C” are butting and/or coincident.

**Fields**

- **outLayer**
  Name for the `geomButtOrCoin` output.

- **inLayer1** and **inLayer2**
  Input layer names. You can specify either a derived layer name or a graphics layer name. Two layers must be specified.
connection  Optional command modifier. The options are as follows:

sameNet  Consider and count butting or coincidence only when the interacting shapes are on the same electrical net.

diffNet  Consider and count butting and coincidence only when the interacting shapes are on different electrical nets.

limits  Optional qualifier you can use to limit the number of separate shapes on inLayer2. If the program finds that the number of shapes on inLayer2 with the required relationship to inLayer1 is within the limits specified, then it selects the shape on inLayer1.

You specify the limits qualifier as an absolute limit or range using one of the following keywords:

- keep  Select the shape on inLayer1 if the number of separate shapes on inLayer2 having the required relationship falls within the numeric range you specify.

- ignore  Reject the shape on inLayer1 if the number of separate shapes on inLayer2 having the required relationship falls within the numeric range you specify.

Operators you can use with the keywords are as follows:

- <
- <=
- >
- >=
- ==

You can specify limits by combining a keyword with one or more operators and numbers using normal mathematical conventions. Examples of these combinations are as follows:

keep == 6
ignore < 2
2 < keep < 5
3 <= ignore <= 7
A shape on inLayer1 with no shape on inLayer2 having the required relationship is not selected, even if the limits you specify encompass the value 0.

exclusive

Optional modifier you can use to select shapes on inLayer1 that conform to the required relationship but that do not have any other relationship with other shapes on inLayer2.

When you specify the sameNet or diffNet options, the shape on inLayer1 is not selected if shapes on inLayer2 have the required relationship but do not have the required net connections.

Examples

The following examples illustrate how you can use various options to select shapes using geomButtOrCoin.

```
error1 = geomButtOrCoin( "poly" "diff" )
error2 = geomButtOrCoin( m1cut metal2 sameNet )
strap = geomButtOrCoin( m1cut metal1 sameNet keep > 2 )
```
geomButtOrOver

[outLayer=] geomButtOrOver ( inLayer1 inLayer2 [connection] [limits] [exclusive] )

Description

The *geomButtOrOver* function selects shapes on the first layer that abut or overlap shapes on the second layer. Butting is edge-to-edge coincidence where the shapes do not overlap at the point of coincidence. Overlap is defined as a common area between the shapes.

All select functions maintain net numbers of shapes in the output layer.

The *geomButtOrOver* function is a combination of the *geomButting* and *geomOverlap* functions.

Prerequisites

If you use the *connection* modifier when using *geomButtOrOver*, the input layers you specify must be connected layers.

The following figure illustrates a *geomButtOrOver* selection.

Shapes on layer1 marked B and O are butting or overlapping shapes on layer2.

Fields

outLayer

Name for the *geomButtOrOver* output.
inLayer1 inLayer2

Input layer names. You can specify either a derived layer name or a graphics layer name. Two layers must be specified.

connection

Command modifier. The options are as follows:

sameNet

Consider and count only butting and overlap when the interacting shapes are on the same electrical net.

diffNet

Consider and count only butting and overlap when the interacting shapes are on different electrical nets.

limits

Optional qualifier you can use to limit the number of separate shapes on inLayer2. If the program finds that the number of shapes on inLayer2 with the required relationship to inLayer1 is within the limits specified, then it selects the shape on inLayer1.

You specify the limits qualifier as an absolute limit or range using one of the following keywords:

keep

Select the shape on inLayer1 if the number of separate shapes on inLayer2 having the required relationship falls within the numeric range you specify.

ignore

Reject the shape on inLayer1 if the number of separate shapes on inLayer2 having the required relationship falls within the numeric range you specify.

Operators you can use with the keywords are as follows:

<

<=

>

>=

==

You can specify limits by combining a keyword with one or more operators and numbers using normal mathematical conventions. Examples of these combinations are as follows:

keep == 6
ignore < 2
2 < keep < 5  
3 <= ignore <= 7

A shape on inLayer1 with no shape on inLayer2 having the required relationship is not selected, even if the limits you specify encompass the value 0.

**exclusive**  
Optional modifier you can use to select shapes on inLayer1 that conform to the required relationship but that do not have any other relationship with other shapes on inLayer2.

When you specify the *sameNet* or *diffNet* options, the shape on inLayer1 is not selected if shapes on inLayer2 have the required relationship but do not have the required net connections.

**Examples**

The following examples illustrate how you can select shapes using *geomButtOrOver* with various options.

```plaintext
error1 = geomButtOrOver( "cut" gate )
error2 = geomButtOrOver( abc def diffNet )
widget = geomButtOrOver( abc def sameNet 2 < ignore < 4 )
```
**geomCoincident**

\[ \text{outLayer=} \] geomCoincident( inLayer1 inLayer2 [connection] [limits] [exclusive] )

**Description**

The *geomCoincident* function selects shapes on the first layer that have edges coincident with shapes on the second layer.

A first layer shape is coincident if any of its edges are coincident with edges of a second layer shape where areas of the shapes at those edges overlap. Shapes that are coincident are not affected by other geometric relationships. A first layer shape can be coincident, overlapping, and butting with a single second layer shape and still be evaluated by the program as coincident.

All select functions maintain net numbers of shapes in the output layer.

**Prerequisites**

If you use the *connection* modifier when using *geomCoincident*, the input layers you specify must be connected layers. The following figure illustrates a *geomCoincident* selection.

Shapes marked “C” are coincident.

**Fields**

`outLayer` Name for the *geomCoincident* output.
### inLayer1 inLayer2

Input layer names. You can specify either a derived layer name or a graphics layer name. Two layers must be specified.

### connection

Command modifier. The options are as follows:

- **sameNet**: Consider and only count coincidence when the coincident shapes are on the same electrical net.
- **diffNet**: Consider and only count coincidence when the coincident shapes are on different electrical nets.

### limits

Optional qualifier you can use to limit the number of separate shapes on `inLayer2`. If the program finds that the number of shapes on `inLayer2` with the required relationship to `inLayer1` is within the limits specified, then it selects the shape on `inLayer1`.

You specify the limits qualifier as an absolute limit or range using one of the following keywords:

- **keep**: Select the shape on `inLayer1` if the number of separate shapes on `inLayer2` having the required relationship falls within the numeric range you specify.
- **ignore**: Reject the shape on `inLayer1` if the number of separate shapes on `inLayer2` having the required relationship falls within the numeric range you specify.

Operators you can use with the keywords are as follows:

- `<`
- `<=`
- `>`
- `>=`
- `==`

You can specify limits by combining a keyword with one or more operators and numbers using normal mathematical conventions. Examples of these combinations are as follows:

```
keep == 6
ignore < 2
```
Diva Reference
Layer Processing Concepts

2 < keep < 5
3 <= ignore <= 7

A shape on inLayer1 with no shape on inLayer2 having the required relationship is not selected, even if the limits you specify encompass the value 0.

exclusive

Optional modifier you can use to select shapes on inLayer1 that conform to the required relationship but that do not have any other relationship with other shapes on inLayer2.

When you specify the sameNet or diffNet options, the shape on inLayer1 is not selected if shapes on inLayer2 have the required relationship but do not have the required net connections.

Examples

The following examples illustrate how you can select shapes using geomCoincident with various options.

   closecut = geomCoincident( cut metal sameNet )
   bridge = geomCoincident( metal cut sameNet keep > 2 )
## geomCoinOnly

\[
\text{geomCoinOnly}(\text{outLayer}=[], \text{inLayer1}, \text{inLayer2}, \text{connection}, \text{limits}, \text{exclusive})
\]

### Description

The `geomCoinOnly` function selects shapes on `inLayer1` that are coincident with shapes on `inLayer2`. To be considered, the shapes on `inLayer2` must be totally inside the shape on `inLayer1`. The relationship between a shape on `inLayer1` and a shape on `inLayer2` must be coincident only. Coincidence is edge-to-edge coincidence where the shapes overlap at the coincident edge.

All select functions maintain net numbers of shapes in the output layer.

### Prerequisites

If you use the connection modifier when using `geomCoinOnly`, then the input layers you specify must be connected layers.

The following figure illustrates a `geomCoinOnly` selection.

![Diagram of geomCoinOnly selection]

**Fields**

- `outLayer`: Name for the `geomCoinOnly` output.
- `inLayer1` and `inLayer2`: Input layer names. You can specify either derived layer names or graphics layer names. You must specify two layers.
connection

Command modifier that can take the following forms:

sameNet
Consider and count only coincidence when the interacting shapes are on the same electrical net.

diffNet
Consider and count only coincidence when the interacting shapes are on different electrical nets.

limits
Optional qualifier you can use to limit the number of separate shapes on inLayer2. If the program finds the number of shapes on inLayer2 having the required relationship to inLayer1 is within the limits specified, it selects the shape on inLayer1.

You specify the limits qualifier as an absolute limit or range using one of the following keywords:

keep
Select the shape on inLayer1 if the number of separate shapes on inLayer2 having the required relationship is within the numeric range you specify.

ignore
Reject the shape on inLayer1 if the number of separate shapes on inLayer2 having the required relationship is within the numeric range you specify.

Operators you can use with the keywords are as follows:

<
<=
>
>=
==

You can specify limits by combining a keyword with one or more operators and numbers using normal mathematical conventions. Examples of these combinations are as follows:

keep == 6
ignore < 2
2 < keep < 5
3 <= ignore <= 7
A shape on inLayer1 with no shape on inLayer2 having the required relationship is not selected even if the limits you specify encompass the value 0.

**exclusive**

Optional modifier you can use to select shapes on inLayer1 that conform to the required relationship but that do not have any other relationship with other shapes on inLayer2.

When you specify the *sameNet* or *diffNet* options, the shape on inLayer1 is not selected if shapes on inLayer2 have the required relationship but do not have the required net connections.
geomEnclose

[outLayer=] geomEnclose(
    inLayer1 inLayer2 [connection] [limits] [exclusive] )

Description

The *geomEnclose* function selects shapes on the first layer that enclose shapes on the second layer. A first layer shape is enclosing if the entire area of a second layer shape is covered by the first layer. Other relationships for the same shape do not affect the selection. Coincidence of edges does not affect the selection.

You can use optional limits qualifiers to specify the number of shapes on the second layer that must be enclosed within a single shape on the first layer for that shape to be selected by *geomEnclose*. With or without a limits qualifier, at least one shape must be enclosed for the shape to be selected.

All select functions maintain net numbers of shapes in the output layer.

Prerequisites

If you use the *connection* modifier when using *geomEnclose*, the input layers you specify must be connected layers. The following figure illustrates a *geomEnclose* selection.

Shapes marked “E” on layer1 are enclosing shapes on layer2.

<table>
<thead>
<tr>
<th>layer1</th>
<th>layer2</th>
</tr>
</thead>
</table>

Fields

outLayer  
Name for the *geomEnclose* output.
inLayer1 inLayer2
Input layer names. You can specify either a derived layer name or a graphics layer name. Two layers must be specified.

connection
Optional function modifier. The options are as follows:

sameNet
Consider and count only when the enclosing and enclosed shapes are on the same electrical net.

diffNet
Consider and count only when the enclosing and enclosed shapes are on different electrical nets.

limits
Number of shapes enclosed using a limit or range containing one keyword. The options are as follows:

keep
Select the shape from layer1 if the number of shapes on layer2 matches the limit or is within your specified range.

ignore
Reject the shape from layer1 if the number of shapes on layer2 matches the limit or is within your specified range.

You can use the following operators with the keywords:

<
<=
>
>=
==

You can use the following limit specifications:

keep == 6
ignore < 2
2 < keep < 5
3 <= ignore <= 7

A shape on layer1 that does not enclose shapes on layer2 is not selected even if the range specifies no lower limit. Zero is not considered inside the range.
exclusive  Optional modifier you can use to select shapes on \textit{inLayer1} that conform to the required relationship but that do not have any other relationship with other shapes on \textit{inLayer2}.

Shapes on \textit{inLayer2} that only butt the shape on \textit{inLayer1} do not prevent the selection of the shape on \textit{inLayer1}.

When you specify the \textit{sameNet} or \textit{diffNet} options, the shape on \textit{inLayer1} is not selected if shapes on \textit{inLayer2} have the required relationship but do not have the required net connections.

**Examples**

The following examples select shapes using \textit{geomEnclose} with various options.

\begin{verbatim}
collector = geomEnclose( "ndiff" c_cut )
tied_tub = geomEnclose( tub contact sameNet )
pass_tub = geomEnclose( tub contact sameNet keep > 1 )
multi_npn = geomEnclose( ntub npn_base 1 < keep < 3 )
error = geomEnclose( ntub npn_base 1 < ignore < 3 )
\end{verbatim}

The following example illustrates how you can pass connectivity information through derived layers.

\begin{verbatim}
geomConnect( via( a b c )
e = geomStraddle( b f )
g = geomEnclose( e c sameNet )
\end{verbatim}
**geomInside**

```plaintext
[outLayer=] geomInside( inLayer1 inLayer2 [connection] )
```

**Description**

The `geomInside` function selects shapes on the first layer that are totally inside shapes of the second layer. A first layer shape is inside if its *entire area is covered* by the area of a second layer shape. Coincident shapes are considered *inside* in this check. If shapes on two layers are coincident on all sides, then each layer can be defined as being inside the other.

All select functions maintain net numbers of shapes in the output layer.

**Prerequisites**

If you use the *connection* modifier with `geomInside`, you must specify connected layers as input.

The following figure illustrates a `geomInside` selection.

Shapes on layer1 marked “I” are inside shapes on layer2.

```plaintext
layer1  layer2
```

**Fields**

- **outLayer**
  
  Name for the `geomInside` output.

- **inLayer1 inLayer2**

  Input layer names. You can specify either derived layer names or graphics layer names.
connection  Optional command modifier. The options are as follows:
sameNet    Select only if shapes are inside the same electrical net.
diffNet    Select only if shapes are inside different electrical nets.

Examples

The following example selects a derived layer inside an original layer.

    ngate = geomInside( gate "pwell" )

The following example illustrates the sameNet option.

    well_poly = geomInside( poly well sameNet )

The following example illustrates how connectivity information can be passed through derived layers.

    geomConnect( via( a b c d ) )
    e = geomStraddle( b f )
    g = geomInside( e c sameNet )
**geomLineEnd**

\[
\text{outLayer} = \text{geomLineEnd}( \text{inLayer} \text{ maxLength} [\text{minLength}] [\text{legs}] [\text{inBox}] [\text{outBox}])
\]

**Description**

`geomLineEnd` selects edges at the end of a line. Line-ends are defined as horizontal, vertical or slanted edges at the end of polygonal lines. The adjacent lines must be perpendicular to the line end. A horizontal line-end is connected to two vertical legs. A vertical line-end is connected to two horizontal legs.

**Arguments**

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>outLayer</code></td>
<td>Layer to which line edges are written</td>
</tr>
<tr>
<td><code>inLayer</code></td>
<td>Layer whose line-ends are written to <code>outLayer</code>. <code>inLayer</code> may be a polygon layer or an edge layer. If <code>inLayer</code> is an edge layer, the adjacent leg edges are still required.</td>
</tr>
<tr>
<td><code>maxLength</code></td>
<td>Value that specifies the longest possible edge that qualifies as a line-end. Line segments longer than this value are never line-ends.</td>
</tr>
<tr>
<td><code>minLength</code></td>
<td>Optional value that specifies the shortest possible edge that qualifies as a line-end. Line segments shorter than this value are never line-ends.</td>
</tr>
<tr>
<td><code>legs</code></td>
<td>`legs( { length</td>
</tr>
</tbody>
</table>

A leg is an edge on the polygon that is adjacent to a line-end edge. Leg edges are used to determine if an edge is in an end-
configuration or step-configuration. Typical step and line end configurations are shown below.

The legs length specifies the minimum length of the adjacent edges needed to make the line-end valid. Both adjacent edges must be at least that length before the line-end is selected. You can specify the legs length as an absolute length or as a relative length.

Keyword used to allow the qualifying legs length to vary according to the size of the line-end. For example, legs(1.2 relative) means that the minimum legs length to qualify a 1 um edge as a line-end is 1.2 um. The minimum legs length for a .5 um edge is 0.6 um. This allows some degree of aspect ratio to be used in selecting end lines.

As shown below, both legs must be present and long enough for an edge to qualify as a line-end. Legs must be perpendicular to the end edge. If a legs option is not used, no leg length checking is done, but both legs must be present for an edge to qualify as a line-end.

This checks a minimum penetration depth the edge has on the polygon. It is an alternative method for selecting end lines. For
each end line candidate, a box of the specified length which penetrates into the polygon must lie inside the polygon.

A box is formed which penetrates into the polygon. The width of the box may be clipped by obscuring edges as shown above. When the box is not completely clipped and its width meets the width requirement, the whole edge is selected as a line-end edge.

You can specify the inBox length and width numbers as an absolute value or as a relative value to the length of the line-end. The relative keyword is used in a similar way as in the legs specification. The multiplier creates inBox length and width values based on the line-end length.

For example, geomLineEnd( inLayer 2 inBox( 1.2 0.9 relative) ) requires that a line-end that is 1 um long must have an unclipped inBox 1.2 in length and 0.9 in width to qualify as a line-end. A line-end that is 2 um long must have an inBox that is 2.4 in length and 1.8 in width to qualify as a line-end.

This checks whether the polygon edges are inside or coincident to the outbox edges. For each candidate edge, a box which is a little wider than the edge is constructed and a test made to see if the polygon is contained in the box. The only valid exit direction is opposite the edge.

In the figure example below, the line end on the left has an OutBox drawn to the right. Its length and extension are specified
in the outBox argument list. Since the polygon exits the box on the right, it is taken as a line-end. If the polygon had exited on any other side, it would have been disqualified and the edge would not be copied to the outLayer.

Note: The performance of this command decreases if a large inBox length is specified due to the extra work done in clipping the penetration box. Performance also decreases when outBox values are given since an extended search is done to see where the polygon exits.

If multiple requirements are specified, all must pass before the edge is copied to the outLayer. Thus if both legs and inBox are used, both must be satisfied.

Examples

Example 1

If all the edges in the figure below are less than 1000 um, geomLineEnd(layer 1000) copies 1, 2, 5, and 8 to outLayer. The decision is based solely on how the adjacent edges are oriented. A more reasonable maxLength would be smaller, say 2. In this case the very long edges are filtered but, the edges 1, 2, 5, and 8 are selected.

The fastest way to select edges 1 and 5 and reject edges 2 and 8 is with the legs specifier. Note that the two adjacent edges for 1 and 5 are long while edges 2 and 8 each have one short leg. By requiring two long legs 2 and 8 get rejected. geomLineEnd( layer 2.0 legs(1.5)) rejects edges 2 and 8 in its most efficient mode.
Other configurations may exist where legs may give inaccurate results.

**Example 2**

In the figure below, edges 1 and 2 are rejected because the legs are too small yet they could be considered to be ends. Although legs is fast, it doesn’t look at enough of the polygon to always make good decisions. To address this, the outBox specifier was created.

```
geomLineEnd( inLayer 2 outBox( 1.2 relative))
```

This command selects edges 1 and 2. It also selects edges 1 and 5, and rejects 2 and 8. However, outBox by itself may still select too many edges.

**Example 3**

In the figure shown below, outBox selects edge 1 if it is shorter than maxLength. outBox is not sensitive to the nearby edge 2 of the polygon. For this reason, inBox was created. inBox projects a box into the polygon. In this case, edge 2 clips the inside box making the inBox too short to qualify edge 1 as a line-end. The command that ignores edge 1 is

```
geomLineEnd( 1 0.2 inBox( 1 .5 relative) (outBox 1 0.2 relative))
```

![Diagram](https://via.placeholder.com/150)
**geomOutside**

```plaintext
[outLayer=] geomOutside( inLayer1 inLayer2 )
```

**Description**

The `geomOutside` function selects shapes on the first layer that are totally outside shapes, outside of the second layer. A first layer shape is outside if none of its area is covered by area of a second layer shape. Butting edges are considered outside.

All select functions maintain net numbers of shapes in the output layer.

The following figure illustrates a `geomOutside` selection.

Shapes on layer1 marked “O” are outside shapes on layer2.

**Fields**

- `outLayer`  
  Name for the `geomOutside` output.

- `inLayer1 inLayer2`  
  Input layer names. You can specify either a derived layer name or a graphics layer name. Two layers must be specified.

**Example**

The following example selects a derived layer outside an original layer.

```plaintext
pgate = geomOutside( gate "pwell" )
```
geomOverlap

[outLayer=] geomOverlap (  
    inLayer1 inLayer2 [connection] [limits] [exclusive]  )

Description

The `geomOverlap` function selects shapes on the first layer that overlap shapes on the second layer.

Overlap is any area common to shapes on both layers. A first layer shape overlaps if it has any common area with a second layer shape. Other relationships for the same shape do not affect the selection. A first layer shape can be overlapping, butting, and coincident with a single second layer shape and still be evaluated by this verification tool as overlapping.

All select functions maintain net numbers of shapes in the output layer.

Prerequisites

If you use the `connection` modifier when using `geomOverlap`, the input layers you specify must be connected layers. The following figure illustrates a `geomOverlap` selection.

![Diagram](image)

Shapes on layer1 marked “O” are overlapping shapes on layer2.

Fields

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>outLayer</td>
<td>Name for the <code>geomOverlap</code> output.</td>
</tr>
</tbody>
</table>
inLayer1 inLayer2

Input layer names. You can specify either a derived layer name or a graphics layer name. Two layers must be specified.

connection

Function modifier. The options are as follows:

sameNet

Consider and count only overlap when the overlapping shapes are on the same electrical net.

diffNet

Consider and count only overlap when the overlapping shapes are on different electrical nets.

limits

Optional qualifier you can use to specify the number of shapes on the second layer. A shape on the first layer must be overlapping for it to be selected.

With or without this qualifier, it must overlap at least one shape. A shape on the first layer that does not overlap any shape on the second layer is not selected even if the range specifies no lower limit. Zero is not considered inside the range. With the qualifier, the number of overlapping shapes is specified using a limit or range containing one keyword. The options are as follows:

keep

Select the shape from layer1 if the number of shapes it overlaps from layer2 matches the limit or is within your specified range.

ignore

Reject the shape from layer1 if the number of shapes from layer2 it overlaps matches the limit or is within your specified range.

You can use the following operators with the keywords:

<
<=
>
>=
==

You can use the following limit specifications:
keep == 6
ignore < 2
2 < keep < 5
3 <= ignore <= 7

exclusive

Optional modifier you can use to select shapes on \textit{inLayer1} that conform to the required relationship but that do not have any other relationship with other shapes on \textit{inLayer2}.

Shapes on \textit{inLayer2} that only butt the shape on \textit{inLayer1} do not prevent the selection of the shape on \textit{inLayer1}.

This relational function is intrinsically exclusive. The only relationship the function does not cover is external butting, and the program ignores any butting shapes.

However, the \textit{exclusive} modifier is meaningful when considering the \textit{sameNet} and \textit{diffNet} options. To satisfy the exclusive request, all interacting shapes must conform to the \textit{sameNet} or \textit{diffNet} relationship.

For example, consider what happens if you use the \textit{exclusive} modifier in combination with the \textit{sameNet} option. If the shape on \textit{inLayer1} is on net number 25, and multiple interacting shapes on \textit{inLayer2} are all on net number 25, the exclusive request is not violated. However, if one of the interacting shapes on \textit{inLayer2} is on net number 34, the shape has a relationship that does not conform to the \textit{sameNet} requirement. The requested exclusive \textit{sameNet} relationship is violated.

Examples

The following examples select shapes using \textit{geomOverlap} with various options.

\begin{verbatim}
error1 = geomOverlap( "cut" gate )
error2 = geomOverlap( m1cut metal2 diffNet )
bridge = geomOverlap( lapcon metal sameNet ignore <= 2 )
\end{verbatim}
**geomStraddle**

```plaintext
[outLayer=] geomStraddle(
    inLayer1 inLayer2 [connection] [limits] [exclusive] )
```

**Description**

The `geomStraddle` function selects shapes on the first layer that straddle shapes on the second layer. A first layer shape is straddling if part of its area is covered by a second layer and part is not. Butting and coincident edges do not affect this function.

All select functions maintain net numbers of shapes in the output layer.

**Prerequisites**

If you use the `connection (sameNet or diffNet)` modifier when using `geomStraddle`, then the input layers you specify must be connected layers.

The following figure illustrates a `geomStraddle` selection.

Shapes on layer1 marked “S” are straddling shapes on layer2.

**Fields**

- **outLayer**
  
  Name for the `geomStraddle` output.

- **inLayer1 inLayer2**

  Input layer names. You can specify either a derived layer name or a graphics layer name. Two layers must be specified.
### Diva Reference

**Layer Processing Concepts**

<table>
<thead>
<tr>
<th>term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>connection</strong></td>
<td>Optional command modifier. The options are as follows:</td>
</tr>
<tr>
<td><strong>sameNet</strong></td>
<td>Select only if shapes are straddling the same electrical net.</td>
</tr>
<tr>
<td><strong>diffNet</strong></td>
<td>Select only if shapes are straddling different electrical nets.</td>
</tr>
<tr>
<td><strong>limits</strong></td>
<td>Optional qualifier you can use to specify the number of shapes on the second layer that a shape on the first layer must be straddling in order for it to be selected. With or without this qualifier, the shape must straddle at least one shape. A shape on the first layer that does not straddle any shape on the second layer is not selected, even if the range specifies no lower limit. Zero is not considered inside the range. With the qualifier, the number of straddling shapes is specified using a limit or range containing one of the following keywords:</td>
</tr>
<tr>
<td><strong>keep</strong></td>
<td>Select the shape from <em>layer1</em> if the number of shapes on <em>layer2</em> matches the limit or is within your specified range.</td>
</tr>
<tr>
<td><strong>ignore</strong></td>
<td>Reject the shape from <em>layer1</em> if the number of shapes on <em>layer2</em> matches the limit or is within your specified range.</td>
</tr>
<tr>
<td><strong>exclusive</strong></td>
<td>Optional modifier you can use to select shapes on <em>inLayer1</em> that conform to the required relationship but that do not have any other relationship with other shapes on <em>inLayer2</em>.</td>
</tr>
</tbody>
</table>

Operators you can use with the keywords are as follows:

- `<`
- `<=`
- `>`
- `>=`
- `==`

You can specify limit or range as shown by the following examples:

- `keep == 6`
- `ignore < 2`
- `2 < keep < 5`
- `3 <= ignore <= 7`
Shapes on inLayer2 that only butt the shape on inLayer1 do not prevent the selection of the shape on inLayer1.

When you specify the sameNet or diffNet options, the shape on inLayer1 is not selected if shapes on inLayer2 have the required relationship but do not have the required net connections.

Examples

The following examples select derived layers straddling original layers.

schottky = geomStraddle( collector "pbase" )
pulldown = geomStraddle( gate "implant" )

The following example illustrates the diffNet option.

error = geomStraddle( oscont metal diffNet )

The following example illustrates how connectivity information is passed through derived layers.

geomConnect( via( a b c ) )
e = geomStraddle( b f )
g = geomStraddle( e c sameNet )

Sizing Functions

You can change polygon or edge layers in these ways:

- For polygon input layers, you can use sizing functions to generate new polygons by oversizing or undersizing the layer.
- For edge input layers, you can use sizing functions to expand an edge into a rectangle, or make an edge longer or shorter.
**geomSize**

[outLayer=] geomSize( inLayer size [edges] [raw] )

**Description**

The `geomSize` function performs an over-size or under-size operation on the specified layer. A size operation consists of changing the absolute dimensions of shapes without altering their dimensions relative to each other. Shapes that grow into one another are merged together. Acute angles are truncated.

**Caution**

*Do not confuse the `geomSize` function with scaling. Scaling changes the sizes of shapes, their relative positions, and the circuit size.*

The following figure illustrates the polygon size operations.

- Butting shapes cannot pull apart in a negative sizing operation since the layer has been previously merged, and construction lines (such as donut hole exit lines) are
automatically removed. Similarly, shapes that were separate originally and extend into each other during a positive sizing operation are merged into a single shape.

Polygons changed by *geomSize* maintain their original shape except when there are shape features that are less than or equal to twice the size value. Holes and notches disappear in a positive grow, and narrow sections disappear in a negative grow.

Acute-angle corners undergo special processing by the program. To avoid extension of an acute corner out of proportion in a positive sizing, the corner is truncated by the program.

Acute angle truncation is not performed during negative sizing.

The amount of truncation used by the program is the square root of 2 times the size value to match the maximum extension of the shape at a 90-degree corner.

You can use the *edges* option to size the edges of a shape. The following figure illustrates edge sizing.

Each edge is expanded out into a rectangle whose width is the size value. Positive sizing puts the rectangles on the outside of the original shape, and negative sizing puts the rectangles on the inside of the original shape. The resultant shapes are not merged together if they overlap.
If the original layer is composed of edges, rather than shapes, this is the default sizing mode. Outside and inside is determined by the edge direction, which is determined by the shape from which the edge originated.

**Fields**

- **outLayer**: Name for the geomSize output. The program always generates a polygon layer for geomSize.
- **inLayer**: Input layer name. This layer can be either a derived layer name or a graphics layer name. It can be a polygon layer or an edge layer.
- **size**: Integer or floating point number defining the amount of the size operation. It can be negative or positive. You can define negative values by preceding the value with a minus (-) sign. Positive values are assumed from an unsigned number. If you use the “+” sign, you must enclose the number in parentheses.
- **edges**: Optional keyword that expands edges into rectangles.
- **raw**: Optional keyword that specifies the shapes on the input layer are not merged together either on input or output. Shapes in the input layer that overlap stay separate. Shapes that overlap as a result of the sizing operation stay separate in the output layer.

**Examples**

The following examples illustrate the geomSize function.

```plaintext
   cblock = geomSize( "cut" 2 )
   terms = geomSize( gate 1.25 edges )
   widemet = geomSize( geomSize( met -1.5 ) 1.5 )
```

The **widemet** example produces wide metal by first under-sizing the metal to remove any narrow sections, in this case less than or equal to 3 wide, and then over-sizing the result to return the remaining shapes to their original size.
geomSizeInTub

outlayer = geomSizeInTub( contact_layer tub_layer
size [number_of_sides] [edge])

Description

The `geomSizeInTub` function checks the electrical integrity of wells.

Tub-tie checks are based on the physics of how a MOSFET works. The following figure illustrates a MOSFET, 4-terminal device.

![MOSFET Diagram]

Electrical isolation between the source and the drain is maintained by keeping the junctions between them and the tub in a strong reversed bias state. Because the tub resistivity is not negligible, the voltage on the right can be lower than the voltage on the left, which is near the n+ tub-tie. Where the voltage is lower, the p-n junction becomes forward biased and turned on.

From left to right, there is a pnpn configuration. This is the configuration of a silicon controlled rectifier. Once turned on, it stays on and can only be turned off by disconnecting it from the voltage source.

The solution is to guarantee that all transistors are close to a `tub-tie`, the n+ region. The farther away the transistor from the tub-tie, the more likely the reverse bias voltage is low. Since this is a resistance effect, the measurement of `close` means how far current must flow through the
tub to get to the transistor. A direct measurement of the distance from the transistor to the
tub-tie is insufficient as is shown in the figure.

The current must flow around the corner to get to the diffusion area near the transistor gate.
This distance is much longer than the direct distance from the gate to the tub-tie.

Measuring distances around bends is not a traditional operation of verification programs but
is a necessary one. This verification tool approximates the exact measurement with simplified
shapes.

**Center of Mass Points**

The first step is to approximate each tub-tie contact as a single point, called the apex. This is
done by calculating the *center of mass* of each tub tie. The precise shape of the tub-tie is not
considered. Each tub-tie is treated as a point. This point is the center from where distances
are measured.

**Create Wedges**

Around each tub tie center of mass point a regular polygon is formed as a collection of
wedges. This polygon is an approximation to a circle.
Wedges are processed one at a time with the sequence of operations.

- The wedge is ANDed with the well. This removes any part of the wedge outside the well. It might also break the wedge into pieces if an obstruction is in the way.

- Select those wedge parts that touch the apex. This has the effect of rejecting the area that must be accessed by turning a corner. This leaves the area that is directly visible to the apex.

- After the wedge is ANDed, a search is made to find vertices from the tub layer that face inward and are directly visible to the apex. A new collection of wedges, with a smaller
radius, is centered on the new point and the process starts again. This allows the program to *walk* around corners.

The following figure illustrates the final result on the original problem definition example (with the gate too far from the contact).

Wedges that totally encompass a corner, or a hole in the tub, are split at the corner into multiple wedges and the corner is used as a new starting (reflection) point.
Since the ends of the wedges are straight lines and not arcs, the circumference of an area encompassed by this check only approximates a circle. The higher the number of wedges used, the better the approximation. When two sets of wedges form different starting points (for example, reflection points intersect), there is not necessarily an exact match of the circumference edges. This results in small steps in the resultant shape. This is clarified using the illustration.

Because of this behavior, it is recommended that the output of this function not be used in any subsequent function that would be susceptible to these steps. An example would be a positive `geomSize`.

**Side Count Limitations**

The Diva verification internal algorithms are based on an integer grid. If you specify a side count of one million, the wedge would be so narrow that it would degenerate into a zero-area polygon when projected onto the integer grid.

The maximum number of sides allowed is limited by the integer grid and by the sizing value. The program checks the width of a wedge and adjusts the side count so that the width is at least 4 units. Even if one million edges are specified, you see much fewer edges in the output shape.

This effect impacts reflection points whose resultant sizing value can be quite small. If the grow size for a reflection point gets so small that a wedge that meets the four unit width (or area) criterion cannot be made, then that reflection point is discarded.
## Fields

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>out_layer</td>
<td>Layer name for the <code>geomSizeInTub</code> result.</td>
</tr>
<tr>
<td>contact_layer</td>
<td>Name for the layer to be treated as contacts.</td>
</tr>
<tr>
<td>tub_layer</td>
<td>Name of the <code>ub</code> layer that determines the bounds constraining the sizing.</td>
</tr>
<tr>
<td>size</td>
<td>Size to grow, measured from the apex of the contact.</td>
</tr>
<tr>
<td>number_of_sides</td>
<td>Optional. The number of wedges to create for each contact point. The minimum value allowed is 4.</td>
</tr>
<tr>
<td></td>
<td>The larger the value, the better the resolution of the sizing, at the expense of run time.</td>
</tr>
<tr>
<td></td>
<td>The default value is 8.</td>
</tr>
<tr>
<td>edge</td>
<td>This optional keyword causes the center of mass growth algorithm to not be used. Instead, the program will grow from the edges of the shape.</td>
</tr>
</tbody>
</table>

As shown in the figure below, rectangle shapes will grow into rectangles with round corners. When obstructions to the growth
are found, the growth will wrap around the corners as described for the center of mass.

Example

The following example illustrates the `geomSizeInTub` operation.

```
tubsize = geomSizeInTub( tubtie tub 10 16)
```
**geomStretch**

\[ \text{outLayer=} \] \text{geomStretch( inLayer stretch )} 

**Description**

The *geomStretch* function takes each edge of a polygon or edge layer and extends or reduces the edge by the amount you specify.

**Fields**

- **outLayer**
  
  Name for the *geomStretch* output. The program always generates an edge layer.

- **inLayer**
  
  Input layer name. This layer can be either a derived layer name or a graphics layer name. The layer name can be a polygon layer or an edge layer.

  If the input is an edge layer, no consideration is made of the vertex points of the original polygons from which the edges were derived. If the input is a polygon layer, each edge is considered independently of the polygon from which it came. The output is always an edge layer.

- **stretch**
  
  Integer or floating point number you use to define the amount of the stretch operation. It can be negative or positive. A positive value of stretch extends each end of the edges by that stretch value. The edges do not change their absolute location. A negative value of stretch reduces the edge length at each end by the stretch value.
Examples

The following examples illustrate the `geomStretch` operation.

```plaintext
cut_range = geomStretch( "cut" 2 )
limits = geomStretch( close_poly 2.75 )
```

General Selection Functions

General selection functions let you select specific kinds of shapes, or shapes with certain characteristics, such as angled edges or texted area. After selecting the shapes, you can use them in verification checking.

Most general selection functions maintain net numbers of shapes in the output layer. The only exceptions are `geomGetMacro`, `geomGetPurpose`, and `geomGetByLayer`, which select from original graphics data that could not have been connected.
**geomGetAdjacentEdge**

\[ \text{outLayer=} \text{geomGetAdjacentEdge( inLayer refLayer )} \]

**Description**

The `geomGetAdjacentEdge` function selects input layer edges that are adjacent to the reference edges. More specifically, this function selects input layer edges that have a head-to-tail relationship with edges of the reference layer.

If all polygons are considered to be a sequence of connected edges, tracing around the polygon in a counterclockwise direction traverses each edge from its *tail point* to its *head point*. An edge of the input layer is selected if its head point touches a tail point of an edge on the reference layer, or its tail point touches a head point of an edge on the reference layer.

The following figure illustrates the `geomGetAdjacentEdge` function.

- **Input layer**
- **Edge reference layer**
- **Polygon reference layer**

This function maintains the net numbers of shapes in the output layer.

**Fields**

- **outLayer**
  - Output layer name. This is an edge layer.

- **inLayer**
  - Input layer name. This layer can be either a derived layer name or a graphics layer name. It can be either a polygon layer or an edge layer.

- **refLayer**
  - Edge or polygon layer that acts as a reference layer.
Example

The following example illustrates the `geomGetAdjacentEdge` selection. The metal at the end of a cut has a higher clearance than at the sides of the cut. The sides are found directly by a DRC enclosure check with the metal. This function then gets the cut ends, given those sides.

\[
\text{cut\_end} = \text{geomGetAdjacentEdge}( \text{cut cut\_side} )
\]
**geomGetAngledEdge**

[outLayer=}geomGetAngledEdge(inLayer angle [fig] [mode])

**Description**

The *geomGetAngledEdge* function selects edges or shapes from the input layer by their angle relative to the X and Y axes.

This function maintains the net numbers of shapes in the output layer.

**Fields**

- **outLayer**: Optional polygon or edge layer name as output.
- **inLayer**: Input layer name. This layer can be either a derived layer name or a graphics layer name. The layer you specify can be either a polygon layer or an edge layer unless you specify the *fig* option. If you specify the *fig* option, the input layer must be a polygon layer. If you specify a graphics layer name, this verification tool merges that layer prior to processing this command.
- **angle**: The angle an edge must have for it to be selected by the Diva verification tool. The interpretation of the angle is dependent on the *mode* switch provided. Specify the value using a limit or inclusive range containing the keyword angle.

You can use the following operators with the keyword:

\[
< \\
\leq \\
> \\
\geq \\
= \\
==
\]

The *mode* switch determines whether you can define a negative angle. Typical formats of the angle definition are

angle \leq 45
angle > 10
-45 < angle < +45
The angle of an edge is measured totally independently of the polygon from which the edge came. The *inside* or *outside* concepts applied to polygons and their edges are not considered when measuring angle.

**mode**

This optional keyword defines the way in which the specified angle is interpreted by this tool. There are four mode switches, including *mode1*, *mode2*, *mode3*, and *mode4*. The default is *mode1*.

The following sections describe the four modes.

**mode1**

The angle or range of angles is specified as a positive value relative to the X or Y axis. All edges having a positive or negative angle conforming to this specification relative to the X axis are selected. This effectively specifies the angle in the first quadrant and mirrors it through X and Y into the other quadrants.

Examples:

30 < angle < 90
angle == 25
mode2

The angle or range of angles is specified as a positive or negative value relative to the X axis. All edges having a positive or negative angle conforming to this specification relative to the X axis are selected. This effectively specifies the angle in quadrants 1 and 4 and mirrors it through Y to quadrants 2 and 3.

Examples:

\[-50 < \text{angle} < 55\]
\[-90 < \text{angle} < 0\]

mode3

This mode is identical to mode2 except the angle is measured relative to the Y axis rather than the X axis.
mode4

The angle or range of angles is specified as a positive value relative to the X axis. All edges having a positive angle conforming to this specification relative to the X axis or Y axis are selected. This effectively specifies the angle in quadrant 1 and rotates it through all other quadrants.

Examples:

0 < angle < 45
45 < angle < 90

fig

Specifies that the figure containing the selected edge is passed to the output rather than to the edge itself. This applies only to polygon input layers.

Examples

The following examples illustrate the geomGetAngledEdge selection.

fortyfives = geomGetAngledEdge( gate angle == 45 fig )
app45 = geomGetAngledEdge( gate 43 < angle < 47 fig )
negs = geomGetAngledEdge( "poly" -90 < angle < 0 mode2 )
orth1 = geomGetAngledEdge( metal 0 <= angle < 2 mode4 )
orth2 = geomGetAngledEdge( metal 88 < angle <= 90 mode4)

Angle 0 to 90
X axis
Y axis
geomGetByLayer

[outLayer=] geomGetByLayer( layer1 layer2 [size] )

Description

You use the geomGetByLayer function to improve performance when running design rule checks on a dense layer that is related to a sparse layer. The function assumes that layer1 is dense and that layer2 is sparse.

For example, consider a long distance spacing requirement between the metal and pad layers. In general, the metal layer is very dense with data covering the entire chip, and the pad layer is sparse with data only near the periphery. The check drc(metal pad sep<50) runs slowly for each piece of metal in the chip. This tool searches around the metal for any pad within the specified distance, and usually there is no pad anywhere near the metal other than on the chip boundary.

You can use the geomGetByLayer command to create a subset of the metal layer from the layout database based on proximity to the pad layer. The result is that only the metal near the pads is processed.

To make the selection, the sparse layer (layer 2) is grown by the specified distance with angled edges converted to manhattan edges. If this grown shape overlaps the bounding box of a shape on the dense layer (layer 1) that dense layer shape is selected.

This function is intended to act as a fast data filter. It is not intended to provide selection by exact dimension separation. It often gathers more shapes than is necessary, but never misses a shape that should be selected. If you want exact selection by dimensions, you can use other functions subsequent to this function.

Note: The output can contain multiple copies of any shape selected from the dense layer and is therefore not suitable for any “raw” operation such as “single layer and.”

Fields

outLayer

Name for the geomGetByLayer output.

layer1

The name of the dense layer. The name can be a text string such as “metal1,” or a layer purpose pair, such as (“metal” “drawing”). This layer must be an original layer. It cannot be a derived layer.

layer2

The name of the sparse layer. The name can be any verification layer containing polygons. Edge layers are not valid.
Optional. You can specify how close a \textit{layer1} shape must be to a \textit{layer2} shape to be copied to the output layer. For example, if you specify a \textit{size} value of 52 microns, the system searches for any \textit{layer1} within 52 microns of the \textit{layer2} shape. The value must be a positive number. The default is 0.0, meaning all shapes that overlap one of the shapes on the sparse layer are copied to the output layer.

**Example**

The following example illustrates the \textit{geomGetByLayer} command.

```ruby
out = geomGetByLayer( "L1" "L2" 1.0 )
```
Note that slanted edges are converted to manhattan edges before being grown.

L1  Dense layer
L2  Sparse layer

Interaction area
Selected shapes on output layer

Manhattan example

Non-manhattan example
**geomGetEdge**

```outLayer = geomGetEdge( inLayer0 [op1 inLayer1] [op2 inLayer2] ... )```

### Description

The `geomGetEdge` function selects edges or edge segments from the input layer by their relationships to shapes on other layers. Those parts of the edges of the first layer that conform to the total relationship defined to other layers are selected.

The `geomGetEdge` function is intended for extraction of edges for DRC, data manipulation, and plotting. You cannot use `geomGetEdge` to generate data for parameter and parasitic extraction.

There is a limit of eight layers the program processes in a single `geomGetEdge` command. If you exceed this limit, the program aborts during rule compilation and displays an error message.

All operations you specify in a single `geomGetEdge` command are executed at the same time. The program does not preprocess data to extract a layer subset with the defined relationship. Also, the program processes multiple `geomGetEdge` commands using the same layers at the same time. The result is that additional complex functions can be added for a small increase in run time.

If the first layer referenced in the command is connected, its node numbers are propagated to the output layer. A layer is connected if it has been allocated node numbers with a `geomConnect` command, or any function that propagates node numbers.

```pint = geomGetEdge( poly inside tub )```

If `poly` is connected, the layer `pint` is allocated its node numbers.

### Fields

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>outLayer</code></td>
<td>Edge layer name for the output.</td>
</tr>
<tr>
<td><code>inLayer0</code></td>
<td>Edge or polygon input layer name. This layer can be either a derived layer name or an original graphics layer name. This layer contains the edge segments that are selected and copied to the output layer.</td>
</tr>
</tbody>
</table>
Diva Reference
Layer Processing Concepts

**op1 inLayer1**
A combination of operator and layer for the initial relationship between edges of layer0 and layer1. The *inLayer* must be a polygon layer.

You can use the following operators:

- **butting**
  Edge coincidence between layer0 and layer1 at a point where the areas of the layers do not overlap.

- **coincident**
  Edge coincidence between layer0 and layer1 at a point where the areas of the layers do overlap.

- **outside**
  Edges of layer0 that are outside the area of layer1 and are not butting.

- **inside**
  Edges of layer0 that are inside the area of layer1 and are not coincident.

- **not_over**
  Edges of layer0 that are outside the area of layer1, including those conforming to the butting operator.

- **over**
  Edges of layer0 that are inside the area of layer1, including those conforming to the coincident operator.

![Diagram](image)

**Note:** In the operator descriptions, the word *coincident* is used generically to denote the sharing of a common edge segment. This should be distinguished from the keyword *coincident* used specifically to denote coincidence when there is overlapping area at the point of coincidence.

**op2 inLayer2**
You can optionally specify a combination of an operator and a polygon layer that can be repeated as required. The combination
Diva Reference
Layer Processing Concepts

further defines the relationship required for selecting the edge segments.

Examples

The following examples illustrate how you can use the *geomGetEdge* function to extract the sides of gates, possibly for gate length or some other parametric measurement.

```plaintext
  gateside = geomGetEdge( gate coincident diff )
  ngateside = geomGetEdge( gate coincident diff over nplus )
```

The following example illustrates extracting a specific edge of diffusion to enable more complex design rule analysis.

```plaintext
  diffside = geomGetEdge( diff outside poly over
                          pwell not_over welltap )
```

The following figure illustrates a complex layer relationship.
geomGetHoled

[outLayer=] geomGetHoled( inLayer )

Description

The geomGetHoled function selects all polygons from the input layer that contain holes.

A hole is an area created when the perimeter of a polygon touches itself, enclosing an area that is not the polygon. These holes are sometimes referred to as “donut holes.”

This function maintains the net numbers of shapes in the output layer.

Fields

outLayer Output layer name.
inLayer Input layer name. This layer can be either a derived layer name or a graphics layer name.

Examples

The following examples illustrate the geomGetHoled selection.

holed_metal = geomGetHoled( "aluminum" )
holed_diff = geomGetHoled( diffusion )
**geomGetLength**

```plaintext
outLayer = geomGetLength( inLayer length [contiguous] [fig])
```

**Description**

The `geomGetLength` function selects edges from the input layer by their individual length or by the summed length of a contiguous sequence of edges.

The following figure illustrates the `geomGetLength` function.

![Figure](image)

This function maintains the net numbers of shapes in the output layer.

**Fields**

- **outLayer**: Edge layer name, or, if you use the `fig` option, a polygon layer name.
- **inLayer**: Polygon or edge layer as an input layer. This layer can be either a derived layer or graphics layer.
- **length**: Length an edge or contiguous sequence of edges must have for it to be selected. The value is in user units, that is, microns.

Specify a value using a limit or inclusive range containing the keyword `length`.

You can use the following operators with the keywords:

```plaintext
<
<=
>
```
You can use the following limit specifications:

length <= 10
length > 10
5 < length < 20

 contiguous

Specifies that the check is not made on the length of individual edges, but on the sum of lengths of contiguous edges. Edges are contiguous if their ends meet at the same coordinate.

If you specify this option with a polygon input layer, the perimeter of each polygon is checked. If the polygon has holes, the outside perimeter and the hole perimeter are considered separately.

 fig

You can specify the optional fig keyword with polygon layer input and/or the contiguous option to generate figure (polygon) output rather than edge output. Without the fig keyword, the program generates individual edges as output regardless of the input format and contiguous option.

If the input is a polygon layer and you specify the contiguous option, the length includes the outside perimeter of the polygon plus the perimeter of any holes in the polygon.

Examples

The following examples illustrate the geomGetLength command.

long_poly_edge = geomGetLength( "poly" length > 50 )
long_gate_edge = geomGetLength( gate-side length > 10 contiguous )
mid_dfig = geomGetLength( diffn 8 < length < 30 contiguous fig )
**geomGetMacro**

[outLayer=] geomGetMacro( inLayer purpose [pinsOnly])

**Description**

The *geomGetMacro* function selects shapes from macro cells and ignores other cells. The function selects shapes in macro cells from the input layer by their layer purpose.

**Fields**

- **outLayer**: Optional output layer for the unmerged polygon layer.
- **inLayer**: Polygon layer as an input layer. This layer must be an original graphics layer since only these layers have a purpose.
- **purpose**: Single text string or list of text strings defining the graphics purpose of the layer selected by this verification tool. For example

  geomGetMacro ( "met1" "drawing" )
  geomGetMacro( "met1" ("pin" "net"))

- **pinsOnly**: Optional keyword that specifies that *geomGetMacro* selects only pin shapes from the specified purpose.

**Example**

Here are some examples illustrating the *geomGetMacro* command.

The following example selects all shapes from metal *pin* purpose:

```plaintext
met_pins = geomGetMacro( "metal" "pin" )
```

The following example selects all shapes on the *pin* and *net* purposes:

```plaintext
mlp = geomGetMacro( "met1" ("pin" "net"))
```

The following example selects only pins on *drawing* purpose:

```plaintext
mlp = geomGetMacro( "met1" "drawing" pinsOnly)
```

The following example selects pins on all purposes:

```plaintext
mlp = geomGetMacro( "met1" "all" pinsOnly)
```
The following example selects only pins on *pin* and *net* purposes:

```plaintext
mlp = geomGetMacro( "met1" ("pin" "net") pinsOnly)
```
geomGetNet

[outLayer=] geomGetNet( inLayer text [text ...] )

Description

The geomGetNet function selects shapes from the input layer by the name of the electrical net to which they belong. The program copies from the input layer to the output layer any shape that is part of a net having one of the net names you provide.

This function maintains the net numbers of shapes in the output layer.

Prerequisites

The input layer you specify must have been previously referenced in a geomConnect command to extract the circuit connectivity. The names of the nets you request with this command must have been defined by pins or text labels.

Fields

outLayer
Optional polygon layer output name.

inLayer
Polygon layer for an input layer. This must be a derived layer name since the layer must have been processed by a geomConnect command or be derived from layers processed in a geomConnect statement by functions that retain the net information.

text
Text strings in quotes. Each text string represents the name of a net and must correspond to a pin name or label in the circuit. You can use wildcard characters following Diva verification conventions for using wildcards. (See the “Using Wildcards” section in Chapter 4 of this manual.)

Examples

The following examples illustrate the geomGetNet command.

    geomConnect (
via( cont metal poly pplus nplus)
  label( "text" metal )
)

power_met = geomGetNet( metal "vdd1" "vdd2"
ground_tie = geomGetNet( nplus "gnd" )
**geomGetNon45**

```
[outLayer=] geomGetNon45( inLayer [fig] )
```

**Description**

The `geomGetNon45` function selects edges from the input layer if they are not parallel to the x and y axes of the circuit and are not at an angle of 45 degrees to them. With the `fig` option, the shapes are selected if any of the edges conform to the criteria.

This function maintains the net numbers of shapes in the output layer.

**Fields**

- **outLayer**: Optional polygon or edge layer name as output.
- **inLayer**: Input layer name. This layer can be either a derived layer name or a graphics layer name. The layer can be either a polygon layer or an edge layer unless the `fig` option is specified, in which case it must be a polygon layer.
- **fig**: Specifies that the figure containing the selected edge is passed to the output rather than the edge itself. This only applies to polygon input layers.

**Examples**

The following examples illustrate the `geomGetNon45` selection.

```
badedge = geomGetNon45( "poly" )
badgate = geomGetNon45( gate fig )
```
**geomGetNon90**

[outLayer=] geomGetNon90( inLayer [fig] )

**Description**

The `geomGetNon90` function selects edges from the input layer if they are not parallel to the x and y axes of the circuit. If you specify the `fig` option, the function selects complete shapes if any of their edges are not parallel to the x and y axes.

This function maintains the net numbers of shapes in the output layer.

**Fields**

- **outLayer**: Optional output layer name. This is a polygon or edge layer.
- **inLayer**: Input layer name. This can be either a derived layer name or a graphics layer name. The layer can be either a polygon layer or an edge layer unless you specify the `fig` option, in which case the layer must be a polygon layer.
- **fig**: Specifies that the figure containing the selected edge is passed to the output rather than to the edge itself. This applies only to polygon input layers.

**Examples**

The following examples illustrate `geomGetNon90` selection.

```plaintext
badedge = geomGetNon90( "poly" )
badgate = geomGetNon90( gate fig )
```
**geomGetPolygon**

[outLayer=] geomGetPolygon( inLayer )

**Description**

The *geomGetPolygon* function selects all shapes that are not rectangles. This is the reverse of *geomGetRectangle*.

This function maintains the net numbers of shapes in the output layer.

**Fields**

- **outLayer**
  - Output layer name.

- **inLayer**
  - Input layer name. This layer can be either a derived layer name or a graphics layer name.

**Examples**

The following examples illustrate the *geomGetPolygon* selection.

```plaintext
non_rect_cuts = geomGetPolygon( "cut" )
complex_gate = geomGetPolygon( gate )
```
geomGetPurpose

[outLayer=] geomGetPurpose( inLayer purpose
    [top_level bottom_level] )

Description

The `geomGetPurpose` function selects shapes from the input layer by their layer purpose. This command has options to control the hierarchical levels from which the data is selected.

**Note:** The DFII system lets you create pins on any layer purpose. It is not necessarily true that a shape on purpose *pin* correlates to a real pin. This command cannot propagate pin information even if you select shapes on purpose *pin*. However, you can use this command to deliberately break the propagation of pin information when you select a layer purpose which you know does not include the pin shapes.

Fields

- **outLayer**: Optional polygon layer output name.
- **inLayer**: Polygon layer as an input layer name. This layer must be an original graphics layer since only these layers have a purpose.
- **purpose**: Single text string or list of text strings, enclosed in quotes, that defines the graphics purpose of the layer selected by this verification tool.
- **top_level bottom_level**: You can optionally specify two integers defining the range of hierarchical levels from which shapes are selected. The current level of the circuit is level 0.

  If you want data from a single level other than 0, set both values to that level. If you want data from all levels, you must specify the range “0 n,” where “n” is a value as large or larger than the highest level of hierarchy.

  In flat mode, if a range is not specified, shapes are selected from the current level down, or all levels. In hierarchical mode, if a range is specified, each level of hierarchy being processed relative to that level is applied. If you don’t specify a range,
shapes are selected in exactly the same way as for logical functions (for example, **geomOr**).

**Examples**

The following examples illustrate the *geomGetPurpose* command.

```plaintext
met_pins = geomGetPurpose( "metal" "pin" )
all_poly = geomGetPurpose( "poly" ( "drawing" "pin" ) )
top_pins = geomGetPurpose( "metal" "pin" )
all_pins = geomGetPurpose( "metal" "pin" 0 20 )
low_pins = geomGetPurpose( "metal" "pin" 1 20 )
```
geomGetRectangle

[outLayer=] geomGetRectangle( inLayer )

Description

The `geomGetRectangle` function selects all rectangles on the input layer whose edges are parallel to the axes and copies them to the output layer. This function maintains the net numbers of shapes in the output layer.

Fields

- `outLayer` Output layer name. This is a polygon layer.
- `inLayer` Input layer. This layer can be either a derived layer name or a graphics layer name.

Examples

The following examples illustrate the `geomGetRectangle` selection.

```plaintext
rect_cuts = geomGetRectangle( "cut" )
simple_gate = geomGetRectangle( gate )
```
**geomGetTexted**

\[
\text{outLayer} = \text{geomGetTexted}( \text{inLayer}, \text{textLayer}, \text{text} \) 
\]

**Description**

The `geomGetTexted` function selects shapes from the input layer by text strings placed on them. Shapes are selected by the program if the origin of the text string is over the area of the shape.

\[
\text{outLayer} = \text{inLayer \ texted \ by \ string \ "res" \ on \ the \ text \ layer}
\]

The program selects polygons on `inLayer` for inclusion on `outLayer` if they contain within their boundaries the origin point of any text item on `textLayer` consisting of the `res` text string. The program might miss a polygon that has the origin of a text item on its boundary.

The following figure illustrates the `geomGetTexted` function.

![Diagram of geomGetTexted function](image)

Optimization is performed by processing in parallel all `geomGetTexted` and `geomGetUnTexted` commands having the same input and text layers.

This function maintains the net numbers of shapes in the output layer.

**Caution**

Be careful when you use the text selection commands with incremental processing. If the shapes cross the incremental region, the text labels might be inside the shape but outside the region and be ignored by the program.
Fields

outLayer  

Polygon output layer name.

inLayer  

Polygon input layer name. This layer can be either a derived layer name or a graphics layer name.

textLayer  

The original graphics layer containing the text strings to be used.

The layer definition can have two formats:

"layer"  
The text is accessed from this layer with purpose drawing.

("layer" "purpose")  
If you specify a purpose, such as pin or text, the text is accessed from that layer purpose. If you define the purpose as all, the text is accessed from all purposes of the layer.

text  

Text strings, each enclosed in quotes. If any one or more of the text strings you specify is on the textLayer, and the text string origin is on a shape of the input layer, that shape is selected by the program.

The characters you use in text strings can be any characters the graphics editor allows you to place in a label. You can use wildcard characters following the Diva verification conventions for wildcards. (See the section, “Using Wildcards” in Chapter 4 of this manual.)

Each string can have multiple instances; each instance of a string selects a different polygon for a single geomGetTexted command. Each instance of a string can be used by multiple geomGetTexted commands to select from different layers. A single shape on an input layer can be selected multiple times by different strings in multiple geomGetTexted commands.

Examples

The following examples illustrate the geomGetTexted command.

    resistor = geomGetTexted( "polysilicon" "text" "res" )
    all_res = geomGetTexted( "polysilicon" "text" "res1"
                  "res2" )
    diode = geomGetTexted( base "mytext" "d" )
metlab = geomGetTexted( ( "metal" "label" ) "clock*" )
geomGetUnTexted

[outLayer=] geomGetUnTexted(inLayer textLayer text
 [text ...] )

Description

The `geomGetUnTexted` function selects input layer shapes that do not have certain text strings. Shapes are not selected if the origin of the text string is over the area of the shape or on any of its edges.

```
layer_out = layer_in not texted by string "vdd" on the
          layer_text
```

Polygons on `inLayer` are copied to `outLayer` if they do not contain within or on their boundaries the origin point of any text item or text layer consisting of text string.

The following figure illustrates the `geomGetUnTexted` function.

![Illustration](image)

This function maintains the net numbers of shapes in the output layer.

**Caution**

*Be careful when you use the text selection commands with incremental processing. If the shapes cross the incremental region, the text labels might be inside the shape but outside the region and be ignored by Assura Diva verification.*

Fields

outLayer Optional polygon layer output name.
inLayer  
Polygon layer as an input layer name. This layer can be either a derived layer name or a graphics layer name.

textLayer  
Graphics layer containing the text strings.

text  
Text strings, each enclosed in quotes. If any one or more of the text strings you specify is on the textLayer, and the text string origin is on a shape of the input layer, that shape is not selected by this tool.

The characters you use in text strings can be any characters that the graphics editor lets you place in a label. You can use wildcard characters following Diva verification conventions for using wildcards. (See the section, “Using Wildcards” in Chapter 4 of this manual.)

Examples

The following examples illustrate the geomGetUnTexted command.

```
non_PG = geomGetUnTexted( "metal" "text" "vdd" "vss" )
collector = geomGetUnTexted( ndiff "mytext" "e" )
```
**geomGetVertex**

[outLayer=} geomGetVertex( inLayer limits )

**Description**

The `geomGetVertex` function selects all shapes on the input layer that have a vertex count matching the limit or within the range you specify in the `limits` argument.

This function maintains the net numbers of shapes in the output layer.

**Fields**

- **outLayer**: Output layer name. This is a polygon layer.
- **inLayer**: Input layer name. This layer can be either a derived layer name or a graphics layer name.
- **limits**: Limits specifies the number of vertices a shape must have before it is selected by this tool. Specify the value by using a *limit or range* containing one keyword. The options are as follows:
  - **keep**: Select the shape if the number of vertices equals the limit or is within your specified range.
  - **ignore**: Reject the shape if the number of vertices equals the limit or is within your specified range.

You can use the following operators with the keywords:

- `<`
- `<=`
- `>`
- `>=`
- `==`

You can use the following limit specifications:

- `keep == 6`
- `ignore < 4`
Examples

The following examples illustrate the `geomGetVertex` selection.

```plaintext
triangles = geomGetVertex( "poly" keep < 4 )
lgates = geomGetVertex( gate keep == 6 )
```

Generation Functions

Generation functions let you generate geometric shapes from areas that are not drawn geometries. You can also supplement text labels on temporary layers for subsequent checking.
**geomBkgnd**

[\texttt{outLayer=} geomBkgnd( [halo] [border] )]

**Description**

The \textit{geomBkgnd} function generates a layer composed of a single rectangle. The size of the rectangle is made up of these elements:

- The cellview bounding box, the area you specify, or the incremental area.
  - For more information about incremental area, see the Incremental section in the Design Rule Checking chapter.
- The halo calculated from the rules
- A small border two user units wide

The optional arguments lets you control the size of the rectangle.

**Field**

- **outLayer**
  - Name for the \textit{geomBkgnd} output. This is a merged polygon layer.
- **halo**
  - The default option. Adds the halo.
- **border**
  - Numeric value that specifies the number of user units to be added. The default for this option is 2.0.

**Example**

The following examples illustrate the \textit{geomBkgnd} command.

- \texttt{substrate = geomBkgnd( halo 3.0 )}
- \texttt{substrate = geomBkgnd( )}
**geomEmpty**

[outLayer=] geomEmpty()

**Description**

The *geomEmpty* command generates an empty derived polygon layer. You can use an empty layer in any command in which you can specify a normal derived polygon layer.

You can use this function to ensure that all branches in the flow of the rules file generate a specific layer so that subsequent rules do not fail because of a missing layer.

**Field**

outLayer

Name for the *geomEmpty* output layer.

**Example**

In the following example, the *lay1* layer is defined early in the rules and is used for various functions. The layer is redefined in *ivIf* branching. The *then* branch creates a new layer, and the *else* branch creates an empty layer.

If you do not define *lay1* early in the rules, the *geomEmpty* command is not needed. This verification tool detects that *lay1* is only generated in one branch of an *ivIf* switch, and this tool automatically generates an empty *lay1* layer in the other branch.

```plaintext
lay1 = geomAnd( lay2, lay3 )
.....
.....
ivIf( switch( "A" )
  then
    lay1 = geomAnd( lay4, lay5 )
    ..... 
  else
    lay1 = geomEmpty()
    ..... 
)
```
geomHoles

[outLayer=] geomHoles( inLayer )

Description

The `geomHoles` function generates shapes consisting of all the holes in shapes of the input layer. A hole is an area created when the perimeter of a polygon touches itself, enclosing an area that is not the polygon. These are sometimes referred to as “donut holes.”

The following figure illustrates `geomHoles` generation.

![Original polygon vs. geomHoles output](image)

Fields

- **outLayer**: Layer name for the `geomHoles` output.
- **inLayer**: Input layer name. This layer can be either a derived layer or a graphics layer.

Examples

The following examples illustrate the `geomHoles` command.

```plaintext
tub = geomHoles( "guard_ring" )
diode = geomHoles( pbase )
```
**geomNoHoles**

[outLayer]= geomNoHoles( inLayer )

**Description**

The *geomNoHoles* function generates new shapes consisting of all the original shapes of the input layer without the holes. A hole is an area created when the perimeter of a polygon touches itself, enclosing an area that is not the polygon. These are sometimes referred to as “donut holes.”

The following figure illustrates *geomNoholes* generation for a single original shape.

```
Original polygon       geomNoHoles output
```

**Fields**

- **outLayer**: Name for the *geomNoHoles* output.
- **inLayer**: Input layer name. This layer can be either a derived layer name or a graphics layer name.

**Examples**

The following examples illustrate the *geomNoHoles* command.

```python
barrier = geomNoHoles( "guard_ring" )
keepout = geomNoHoles( geomSize( diffn 5 ) )
```
Layer Storage and Removal

The Layer Storage and Removal commands let you save layers you have generated with other commands, or remove layers you no longer need in your database.
copyGraphics

copyGraphics( ( layer purpose ) [all] [cell_view] )

Description

The `copyGraphics` function lets you copy data from the cellview being analyzed to another view. Any valid combination of layer and purpose can be copied. The data is copied by this verification tool to the same layer, but is always saved as `drawing` purpose.

Prerequisite

The layer to be copied must be defined in the technology file.

Fields

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>layer purpose</td>
<td>Original graphics layer name followed by a layer purpose, both in quotes, defining the layer to be copied. The layer can be any layer that is present in the layers definition. Any valid purpose can be used.</td>
</tr>
<tr>
<td>all</td>
<td>Keyword defining the levels of hierarchy from which data is to be copied. With the <code>all</code> option, data is copied from all levels of hierarchy. If you do not use this option, data is copied only from the top level of hierarchy.</td>
</tr>
<tr>
<td>cell_view</td>
<td>Optional keyword that lets you copy data to the <code>excell</code> view during hierarchical extraction. If you do not specify the <code>cell_view</code> optional keyword, this defaults applies.</td>
</tr>
</tbody>
</table>

- During extraction, shapes are copied from the view being analyzed to the `extracted` view.
- During abgen, shapes are copied from the view being analyzed to the `abstract` view.

Examples

The following example illustrates the `copyGraphics` command used to copy the text layer of drawing purpose from the top level of hierarchy.

```
copyGraphics( ( "text" "drawing" ) )
```

The following example illustrates the use of all the options.
copyGraphics( ( "label" "drawing" ) all cell_view )
**geomErase**

`geomErase( inLayer [purpose] )`

**Description**

The `geomErase` function lets you delete a layer from the graphics database. It erases the defined layer from the cellview you are verifying. You can use this command at the beginning of a command file to erase layers stored during previous runs before they are saved again in the next run.

When this command is used with incremental processing (as in DRC), erasure of layer data only occurs in your specified incremental processing area. Any shapes interacting with the area expanded by the maximum design rule dimension are removed. All shapes outside this area remain.

It is not necessary to erase the marker layer. Diva verification automatically removes any markers from previous runs. If you are running incrementally, or by area, only markers within the area being processed are erased.

**Caution**

*You can erase your original data without meaning to. Any data saved by a previous run that you want to erase should be written to a layer that you do not draw on. If you draw on metal1, save your generated `metal1` shapes to `metal1gen`. Only use the `geomErase` command on `metal1gen`, never on `metal1`.*

**Note:** When writing rules for hierarchical processing, users must be aware that each cell is processed in two passes. Any data saved to the cellview in the first pass will be erased in the second pass unless the `?currentCell` switch is used to prevent this. See [Improving Hierarchical DRC Performance](#), for more information on hierarchical processing.

**Prerequisite**

The layer to be erased must be defined in the technology file.

**Fields**

- **inLayer**
  
  Original graphics layer name, defining the layer to be erased. The layer can be any that is present in the technology file.
purpose

Layer purpose, enclosed in quotes. Only shapes on the specified layer with this purpose are erased by this tool. You can use any valid purpose. If you do not specify a purpose, the purpose drawing is used.

Examples

The following examples illustrate the geomErase command.

    geomErase( "y0" )
    geomErase( "border" "boundary" )
saveDerived

saveDerived( inLayer [( outLayer outPurpose )] [view][tile][message] )

Description

The saveDerived function lets you save a layer into the graphics database. The saveDerived function can operate in hierarchical mode, although you might get unexpected results. For more information about limitations for saving derived layers, refer to the DRC Limitations section of the manual.

Prerequisite

The layer to be saved must have been derived.

Fields

inLayer

Derived layer name or original graphics layer name in quotes, defining the layer to be saved. Any unmerged layer is merged when it is saved in the database.

outLayer outPurpose

Optional layer consisting of a layer name and a layer purpose to define which layer in the output cellview the input layer is to be saved on. You must specify the name and purpose in quotes and enclose the pair in parentheses. The output layer must also be defined in the layers definition. For example

( "poly" "drawing" )

If you do not specify an outLayer and outPurpose arguments, this tool defaults to the marker layer and error purpose. The default is equal to this definition:

( "marker" "error" )

If you want all purposes of a layer, specify all as the purpose.

( "poly" "all" )
view

Optional keyword to specify the view name in which the output data is to be saved. The possible view names are as follows:

lay_view layout view
ext_view extracted view
cell_view excell view
abs_view abstract view

The keywords are useful only in their correct context. A \texttt{saveDerived} command with the \texttt{abs_view} keyword only functions during abstract generation. In any other context, such as DRC or layout extraction, the \texttt{abs_view} keyword is ignored.

If you do not use the \texttt{view} keyword, this tool defaults to the view being verified.

tile

Optional keyword defining the format in which the output layer is to be saved.

When you use the \texttt{tile} option, this product generates the output layer in trapezoids (tiles).

If you do not use the \texttt{tile} option, the output layer is formatted by this tool as polygons with a vertex count limit of 200. Larger polygons are automatically broken up into smaller polygons within the 200-vertex count limit.

message

Text string that defines a message this product associates with any shape saved on the marker layer. The contents of the text string, exactly as specified, are attached to each shape by this verification tool and can be viewed with the \texttt{explain} command of the DRC program.

The DRC program generates errors automatically on the marker layer with an associated text string. The \texttt{message} argument lets you supplement this error information with derived layers and still associate a text string with the shapes.

\textbf{Examples}

The following example illustrates the \texttt{saveDerived} command with all options defaulted. This example saves the data on the marker layer.
saveDerived( bad_via )

The previous example can be expanded by adding a message to be associated with the error.

saveDerived( bad_via "Incorrectly formed vias" )

`saveDerived` command examples using multiple options are as follows:

```plaintext
saveDerived(ref_layer ( "y0" "drawing" ) cell_view tile)
saveDerived( wideMetal ( "marker" "warning" ) tile
             "metal wider than 5 " )
```
Connectivity Extraction Concepts

In the Diva verification system, you must establish the connectivity of the circuit for connectivity-related rules in DRC and for device, parameter, and parasitic extraction. You can establish connectivity with the `geomConnect` command, which is described in the command reference part of this chapter. If you use the `geomConnect` command, you can specify how the different layers in the circuit are to be labeled and connected.

**How Connections Are Formed**

A connection is formed when a single shape on a contact layer overlaps shapes on at least two interconnect layers. The connection is based on the contact layer. If there is no contact for a connection, this verification tool does not recognize a connection. However, if there is a contact, but the connection is badly formed because there is no interconnect layer, this tool recognizes this and flags the error on the marker layer in the layout.

External butting between a shape on one layer with a shape on another layer is not considered overlap. There must be a common overlap area between the via and all the interconnect shapes before this product forms a connection. It is not sufficient for each of the interconnect shapes to overlap the via, but in different areas.

If a contact overlaps multiple interconnect layers, this verification tool connects all of them and they become the same electrical net. To connect a single via with only two layers, regardless of how many overlap it, you must split the contact layer using the appropriate layer derivation functions. You can then define multiple via sections with the `geomConnect` command.

If a single contact forms more than one connection because of multiple overlapping interconnect layers, each set forms an independent connection. This tool allocates the via itself to only one of the resulting nets.

**Methodology**

Diva verification processes all connections on interconnect layers and groups all of the interconnect shapes into nets. Each net is a unique electrical entity. Each shape is marked
with the number of the net to which it belongs. DRC uses these numbers for the `sameNet` and `diffNet` options.

You extract connectivity during both DRC and Extraction. You can use connectivity in DRC independent of other extraction processing. You also extract connectivity during Extract. When you run Extract, the program creates an extracted cellview that contains the connectivity and device information.

**Limitations**

Some of the limitations for extracting connectivity are as follows:

- Only one `geomConnect` statement is allowed per run. (This restriction ensures that connectivity definitions do not conflict with each other.) However, using the `ivlf` command, you can have more than one `geomConnect` statement in a rules file and use switches to control which `geomConnect` statement is executed when you run the job.

- In DRC, connectivity information is meaningless unless you run in full flat mode. The connectivity is always meaningful in an extraction run.

- The contact must be on a different layer from the interconnect. For each contact layer, you can only specify one via section with the `geomConnect` command.

- If a contact does not connect to two layers, this tool flags the error on the marker layer (marker error). The via forming the bad connection is allocated a dummy net number of zero and is therefore not connected to any real net in the circuit. When the vias from badly formed contacts are used in commands requiring nodal information like the `sameNet` and `diffNet` options of DRC Parasitic Measurement, they can create false errors or hide actual errors.

**Labeling**

Diva verification assigns arbitrary net numbers to each electrical net it generates. To label the nets, you can use one or more of these options.

- Graphics text string labels
- Supplemental text labels
- Pin names

Pins are taken from the layers used by the `geomConnect` statement. Since the input to `geomConnect` often includes derived layers, this tool determines which original layers these
derived layers are based on and extracts the pins from those original layers wherever possible.

\[
\text{metal} = \text{geomAndNot( "aluminum" "logo_metal" )}
\]

\[
\text{geomConnect( via( cut metal poly ) )}
\]

In this example, the program automatically uses the pins from aluminum in order to label the metal during connectivity analysis.

For associating the original layer pins to derived layers, these rules apply for logical functions.

<table>
<thead>
<tr>
<th>Logical Function</th>
<th>Pins from Layer A</th>
<th>Pins from Layer B</th>
</tr>
</thead>
<tbody>
<tr>
<td>geomOr( A B )</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>geomAnd( A B )</td>
<td>yes</td>
<td>yes</td>
</tr>
<tr>
<td>geomAndNot( A B )</td>
<td>yes</td>
<td>no</td>
</tr>
</tbody>
</table>

The association is also made through any function that selects polygons without modifying them (for example, \textit{geomOverlap}).

**Graphics Text String Labels**

You can name a net with a text string from the graphics database. If you place the origin of a text string over an interconnect shape in the layout, you can use the \textit{label} section of the \textit{geomConnect} command to apply that label during connectivity processing.

You can apply one label name any number of times to a single net. If a single net has two or more different names, the system selects the first one it encounters, and the others are flagged with warning messages. If two or more nets have the same name, the systems labels only one with that name and generates a warning message.

If two or more shapes lie under the same label origin, the label is applied to the shape of the first layer you define in the \textit{layer} section of the \textit{geomConnect} command.

Labels are only accessed from the top level of the circuit hierarchy. This avoids the problem of multiple placements of cells causing duplicate label names.
Supplemental Text Labels

The *layerText* command lets you temporarily add text labels to the graphics database on a run time basis. You can add these labels at the beginning of the run. During the run, they behave exactly as normal graphics labels. The system deletes them from the database at the end of the run.

**Note:** You must reference these labels in the label section of the *geomConnect* command, exactly as normal graphics labels.

Pin Names

Pins define connections external to a circuit or cell, and must have a name. This tool automatically gives a net the name of any pin (terminal) on it. During extraction, the system copies pins, along with their names, to the *extracted* version, so the pin name is used for both the pin and the net.

Modifying Labels

There are four commands you can use to modify and manipulate labels. You can use them in the Run Specific File (RSF) or in the DRC or Extraction rules files. Below is a summary of their functions. See Chapter 4, “Writing Diva Verification Rules” for more details about these commands.

**changeLabel**

Use the *changeLabel* command or property during run time to change the name of a net without changing the text that defines that name. You can use this command to force names into a particular convention, such as *VDD* to *vdd!*, or to force different nets to have the same name, so you can use the *joinableNet* command.

**joinableNet**

Use the *joinableNet* command during run time to direct the system to extract disjoint nets having the same names as a single net.

Some cells can have multiple power or ground nets that are connected together at some higher level in the circuit hierarchy. When verifying at the cell level, you must use the *joinableNet* command to tell the program that the disjoint pieces of these global nets should be the same net.
To join multiple nets that have different names, use the `changeLabel` command to make all the names the same before using the `joinableNet` command.

`globalLabel`  
Use the `globalLabel` command to treat labels as if they were from the top level cells even though they actually come from lower-level cells. The `globalLabel` command must precede the function that uses the labels. If you place it after `geomConnect`, it has no effect on the net names.

`groundNet`  
Use the `groundNet` command to provide a meaningful net name to be used by programs that generate connections to ground.

The specified net name does not need to exist in the circuit. This tool creates the net name if it does not exist.

**Connectivity Commands**

The following sections discuss about the connectivity commands.
geomConnect

description

The *geomConnect* function defines circuit connectivity by specifying how the various layers interact. All shapes on conducting layers, plus vias, are assigned to electrical nets for use by subsequent Diva verification programs.

prerequisites

All the input layers you use with this command must have been derived through some previous function. If you want to use nonderived layers as input, they must be processed through the *geomCat* or *geomOr* functions. Only the label layer is entered as an original graphics layer.

The *geomConnect* command is allowed only once in any set of technology file command stream.

Note: To have different *geomConnect* statements in your technology file (for example, to describe DRC and Extract using different styles and logical layers), use the *ivIf* command. See the section, Conditional Execution of DRC and Extraction Rules (*ivIf*) in Chapter 2 of this manual for more information.

fields

via( viaLayer conLayer1 conLayer2 ... )

An internal function introduced by the *via* keyword defining the via layer (*viaLayer*) and at least two conducting layers that it interconnects (*conLayer1 conLayer2 ...*). Shapes on the *viaLayer* are checked against shapes on the conducting layers to determine if they form electrical connections.

To be connected, *conLayers* must have an area in common with the *viaLayer*. External butting of *conLayers* does not generate a connection. If a shape on the *viaLayer* has more than one disjoint set of common areas with conducting layers, each area forms an independent connection. The *viaLayer* itself is assigned to only one of the resulting electrical nets.
At most, 15 layers can be connected with a single viaLayer. All layers sharing a common area with a via are interconnected. An overlap of a viaLayer with one or fewer conducting layers results in a warning message in the system log and a copy of the via placed in the layout on the marker error layer.

Any number of via functions can be used. If multiple via functions use the same viaLayer, they are processed as a single via function.

The viaLayer cannot be the same as one of the conducting layers.

Some designers artificially divide layers, such as diffusion, into multiple ndiff and pdiff layers for analysis. These layers are then used in a via statement to extract the connectivity information required by subsequent rules.

If designers make too many artificial divisions, however, the layers per via limit can be exceeded. A better solution is to divide the layer after geomConnect. The rules normally used to divide layers are geomAnd and geomAndNot. For these rules, the output layer will contain nodal information from the first input layer.

These rules

\[
\text{ndiff} = \text{geomAnd}(\text{diff well }) \\
\text{pdiff} = \text{geomAndNot}(\text{diff well }) \\
\text{geomConnect}(\text{via}(\text{contact metal poly ndiff pdiff }))
\]

\[
\text{can be rewritten as}
\]

\[
\text{geomConnect}(\text{via}(\text{contact metal poly diff })) \\
\text{ndiff} = \text{geomAnd}(\text{diff well }) \\
\text{pdiff} = \text{geomAndNot}(\text{diff well })
\]

This arrangement produces the same connectivity in both cases, unless a diff geometry is not entirely inside or outside a well geometry. This one exception is normally a DRC error and should not be of concern.
An added benefit is that the *geomConnect* rule runs faster because it is processing fewer layers with the *vialayer*. The *geomAnd* and *geomAndNot* rules run in the same amount of time when propagating nodal information after *geomConnect* as they would when propagating nodal information before *geomConnect*.

```
label( labelLayer conLayer... )
```

This internal function introduced by the keyword *label* defines a *labelLayer*, and the conducting layers to which the label applies (*conLayer* ...). All text labels on the label layer are checked against shapes on the conducting layers. If the origin of a label is inside a shape, it names the electrical net of which that shape is a part. Any electrical net not named through a label is given a default name. Default net names are a number without alphabetic prefix or suffix.

The following rules apply to labels:

- The label layer must be an original graphics layer enclosed in quotation marks.
- The *labelLayer* can be defined with two formats:
  
  - "layer"
    - The labels are accessed from this layer with purpose *drawing*.
  
  - ("layer" "purpose")
    - If you define a specific purpose, such as *pin* or *text*, the labels are accessed from that layer purpose. If you define the purpose as *all*, the labels are accessed from all purposes of the layer.

- A label positioned over multiple conducting layers applies to the first layer on the label list. Layers are selected by their precedence in the label list, where the first layer takes highest precedence and the last layer takes lowest precedence.
- Labels are taken only from the top-level of the circuit hierarchy being processed. Any labels you place in lower level cells are ignored by the program.
- If more than one layer is detected, a warning is issued and the first layer is used.
- Duplicate labels can be placed on a single electrical net.
- If multiple labels with different names are associated with the same electrical net, that net is given the name of the first label found by this tool, and a warning message is generated for the other labels.
If multiple electrical nets are associated with labels of the same name, the first net found is given that name, and a warning message is generated unless the \textit{joinNet} option is specified at run time.

\texttt{soft( softLayer... )}

This internal function introduced by the keyword \textit{soft}, identifies one or more layers as forming “soft” connections.

A soft layer is electrically connected, but should not be used to provide connections between shapes. An example being the substrate or well layers. These layers must be connected to a power supply because they are too resistive to be used to carry power to other shapes.

When a soft layer shape is found to be connected to more than one shape, which would not be connected if the softlayer were not present, the soft layer shape is considered to be in error and a marker is produced.

Soft layers must be connected to layers mentioned in a via statement. Soft layers may not be the via layers themselves.

\textbf{Example}

The following example illustrates the \textit{geomConnect} command and the preprocessing of original graphics layers to generate derived layers. In this example, the name \textit{via} is used as a keyword, an original layer, and a derived layer.

\begin{verbatim}
well = geomOr( "well" )
metal1 = geomOr( "metal1" )
metal2 = geomOr( "metal2" )
poly = geomOr( "poly" )
cut = geomOr( "cut" )
via = geomOr( "via" )
geomConnect(
   via( via metal1 metal2 )
   via( cut poly diffn metal1 well)
   label( "mltext" metal1 )
   soft( well)
)
\end{verbatim}
**geomStamp**

```plaintext
outLayer = geomStamp( inputLayer connectLayer [error] )
```

**Description**

The *geomStamp* function lets you transfer electrical connectivity information (net number) from a previously connected layer to any other layer.

Each shape on the *inputLayer* is copied to the *outLayer* with the net number of the shape on the connect layer it overlaps. If an input layer shape is not overlapping a connect layer shape, the output shape is given a new, unique net number.

For shapes that are stamped by multiple stamping shapes having different net numbers, the number of stamping shapes is counted for each different net and the stamped shape is given the net number having the most stamping shapes. The other stamping shapes are ignored.

For the unstamped and multiply stamped shapes, the output layer shape and the ignored stamping shapes can optionally be echoed on the marker error layer.

**Prerequisite**

The *connectLayer* must be a connected layer.

**Fields**

- **outLayer**: The optional name for the merged polygon output. This layer is a copy of the stamped layer (*inputLayer*) with the connectivity information added. If you specify the same name for the *outLayer* and *inputLayer*, the *outLayer* replaces the *InputLayer*.

  If you do not specify *outLayer*, no connectivity information is added. If you do not specify *outLayer* and do not use the *error* option, the rules compiler does not use this command.

- **inputLayer**: The input layer name. The name can be either a derived layer name or a graphics layer name enclosed in quotes. This verification product copies shapes from the input layer to the output layer. The input layer must be a layer that contains complete polygons that are merged or unmerged. Layers of incomplete polygons, also known as edge layers, are not valid.
**connectLayer**  
The layer from which the net information is to be taken. This must be a connected layer.

**error**  
Use this keyword to request that all error shapes be written to the marker error layer. Error shapes are as follows:

- All input shapes not overlapped by a single connect shape.
- All input shapes overlapped by multiple connect shapes.
- All stamping shapes which contribute to multiple stamping but were not selected as the majority net number.

**Example**

The following example illustrates the `geomStamp` command. The `tubtie` is used in a `geomConnect` statement without connecting it to the tub. This command not only gives the tub the appropriate net numbers, but also flags all tubs that are not connected or that have multiple connections.

```plaintext
geomConnect( via ( cut tubtie diff ) )
tubnet = geomStamp( tub tubtie error )
```
DRC Overview

DRC lets you verify that your layout conforms to the technology design rules. You can perform checks ranging from single layer, single dimension checks to multilayer, multidimensional checks. DRC accepts polygonal data at all angles and does not limit the type, amount, configuration of data, or the number of rules being checked.

The result of any DRC command either generates an error or creates a derived layer. You can use a derived layer as input for other Diva verification commands.

Modes of Operation

DRC runs in six operation modes. Three of these modes, called “check modes,” are as follows:

- Full
- Incremental
- Area

Three of the operation modes, called “check limits,” are as follows:

- Flat
- Hierarchical without optimization
- Hierarchical with optimization

You can specify check modes and limits using menu, SKILL, and UNIX commands. You can specify only one check mode and one check limit per run. You cannot use area check mode with a hierarchical check limit.

**Note:** Commands in hierarchical and area modes might not behave the same as in flat mode. This can result in significant differences when you use other functions and save data to the database. For an example of hierarchical mode reaction, see the “DRC Limitations” section.
Full

In full mode, DRC verifies the entire circuit, regardless of changes made since the last time the circuit was verified.

Incremental

In incremental mode, DRC verifies only those areas of the circuit that were modified since the last time the circuit was verified. The first verification run on a circuit is always performed in full mode even if incremental mode is specified. Everything in the circuit is now considered “new,” meaning the entire circuit has changed.

Changes to the circuit or design rules drive incremental processing. These changes are recorded in the circuit database via shapes on a special layer (changedLayer). Before any DRC is run, the creation of data generates a shape on the changed layer encompassing the complete circuit. A full run of DRC clears out any previously created changed layer shapes. If you run DRC by area after a full run with a different rule set, a changed layer shape is introduced over the reveredified area to ensure that incremental processing continues correctly.

During checks, DRC extends changed areas to ensure that all design rules are considered. DRC joins separate but nearby changed areas into single changed areas, which might result in redundant checks of unchanged areas. If you change the DRC commands between runs, the complete circuit is verified regardless of the changed areas.

If a cell is changed, a change layer is generated in that cell. If the cell is verified as the master cell, that changed layer is utilized and eliminated. However, if you use that cell later in a previously verified design, this tool detects that the cell has changed, and creates a changed layer shape to completely encompass the cell. Incremental processing on the previously verified design will, therefore, behave correctly.

The only exception to the automatic detection of changed cells is for pcells. A new set of layout data is created every time a pcell is loaded. Therefore, the data on the pcell is always different. This tool cannot detect when a pcell has changed contents and needs to be re-checked. To avoid having every pcell rechecked when incremental checking is invoked after loading a design, this tool does not attempt to generate a change layer for pcells as they are loaded. Because pcell changes rarely occur, and can be detected through full flat or hierarchical checking, this limitation has minor impact.

Area

In area mode, DRC checks only the area of the circuit that you define. You can define the area with menu commands or with the run-specific verifyArea command. If you use menu commands, only a single rectangular area can be defined. If you use the verifyArea
command, the area can consist of multiple rectangles, and you can specify areas within the rectangles that you do not want DRC to check.

**Flat**

In flat mode, DRC does not consider the hierarchy of the layout and verifies the circuit as if it was a single cell. DRC expands all instances of all cells into their parent cells up to the master (top-level) cell.

**Hierarchical Without Optimization**

In hierarchical mode, DRC considers the hierarchy of the layout during verification. Hierarchical mode can reduce run-times for designs that have several cell repetitions and few intercell interactions throughout the hierarchy. A design with little repetition or many inter-cell interactions might run slower in hierarchical mode.

This methodology is efficient as long as the layout methodology of the circuit is simple. Any circuit that has a high degree of cell interaction or shapes that overlap cells won’t be processed efficiently. The design rule dimension values are considered to prevent relationships from being overlooked.

DRC methodology is as follows:

- **DRC computes a halo value based on the DRC rules.** The halo value is the distance that DRC needs to look outside the area being checked to ensure that all violations are discovered. Conceptually, the halo is the largest design rule. If you consider the rules

  \[
  a = \text{geomSize} \ (M, \ 5) \\
  \text{drc} \ (a, \ \text{sep} < 5)
  \]

  you might find that the true halo distance is 15, not 5. It is possible for compound checks to have additive distances. In another example, consider the rules

  \[
  a = \text{drc} \ (M, \ \text{sep} < 3) \\
  \text{drc} \ (a, \ b, \ \text{sep} < 4)
  \]

  that have a halo of 7.

- **DRC flattens “small” cells into their parent cells.** A cell is considered “small” if both its width and length are less than twice the halo distance.

- **DRC computes two types of interaction areas: cell-to-cell and geometry-to-cell.** A cell-to-cell interaction is when two cells overlap within the halo distance. A geometry-to-cell interaction is when a piece of geometry overlaps a cell within the halo distance.
Typical Interaction Areas

- DRC processes the geometry of each cell. Geometries from other cells are not considered. Some errors discovered in this phase are removed during the next phase, the processing of interaction areas.

- DRC processes all interaction areas. All data involved in the interaction area is considered. Errors discovered during previous steps are removed if they are not real errors.

  Errors that are unique to a cell are reported within that cell. Errors that involve interactions are written to the cell where the interaction occurs.

Hierarchical with Optimization

Hierarchical mode utilizes a pattern recognition feature. Before processing all the interaction areas, DRC identifies unique cell-to-cell interactions by considering cell names, their relative placement, and their relative orientation. Unique interactions are then processed.

When processing the unique interactions, hierarchical optimized mode does not remove errors detected in previous steps of hierarchical DRC. A fundamental assumption of hierarchical optimized mode is that interaction areas do not differ. This assumption can give misleading results as shown in the “DRC Limitations” on page 262.

Pattern recognition is usually applied when you have regular arrays. DRC first checks the periphery cells, along with their adjacent cells, then looks at the relationships between blocks of adjacent cells. DRC can then define all possible relationships between cells that are repeated throughout the hierarchy.
Area Halo Processing

During area processing, the Diva verification tool calculates a “halo” distance around the area to capture all interactions that might impact the correctness of the processing. This halo is based on the maximum distance that can cause interactions or relationships as defined by the dimensions in the design rules.

The area processing can be turned on in three places.

- The UNIX `ivVerify` command `-area` option
- The SKILL `ivDRC` function `?area` option
- The run-specific file `verifyArea` command

In some cases, the halo value can be extremely high and therefore slow down the processing significantly. If you determine that the halo processing is unnecessary, you can switch it off.

The area processing halo value can be forced to zero by turning the zero halo option on in these ways:

- The UNIX `ivVerify` command `-z` option
- The SKILL `drcZeroHalo` function used either in the run-specific file or in the `.cdsinit` file

Placing the `drcZeroHalo` command in the `.cdsinit` file sets the mode for the duration of the interactive session. Used in a run-specific file, this command sets the mode for a single Diva verification run.

**Note:** This option is disabled when running in hierarchical or incremental modes.

If you set the inclusion limit to 4 on startup, cell C is ignored because its property value is greater than the start-up inclusion value.

Specifying a value in the `ivIncludeValue` property extends the property’s capability beyond an on-off switch. You can set different property values for cells at different levels of hierarchy or for cells requiring differing degrees of re-verification. You can then control which level of cells get checked by varying the inclusion value at start-up.
DRC Limitations

These sections describe the DRC limitations for hierarchical mode, as well as limitations for incremental and area modes.

Hierarchical Mode

Running DRC in hierarchical mode does not always generate the same results as flat mode when using select, separation/notch, apposition and length, or for any nodal checks. The results are different because checking is done in a local area and select, separations/notch, appositions, lengths, and nodal checks require a global view. In addition, hierarchical optimized mode can differ because this mode does not observe “fixes” from higher levels of hierarchy. Sizing to a database layer is not permitted in hierarchical mode because the results differ from flat mode.

Unlike the hierarchical without optimization mode, the optimized mode does not consider all the data in an interaction area. Instead, optimized mode processes only the data from both cells in the cell-to-cell interaction. Data from lower level and higher level cells are not included. Because only unique cell-to-cell interactions are processed, the data might not be included.

For example, two cell-to-cell interactions occur with four placements of A. Cell A contains only a single area of metal. Two of the cell A placements are completely covered by metal. The other two placements do not have any metal geometry near them.
The top two placements of A form one cell-to-cell interaction, and the bottom two form another. Since the interactions are the same in terms of relative placement and orientation, they share the same unique relationship. The A-to-A interactions cause a metal separation violation. If the top interaction contains an error and is the interaction that DRC processes (the choice is arbitrary), a separation violation is reported for that interaction only. This is misleading because the interaction, in context with the metal overlapping the top two cells, implies that no error occurs. Actually, an error does occur in the interaction with the bottom two cells, but the error is reported in the top two cells that are interacting. DRC uses this method to make sure that all real errors are located.

**Select**

Relational selection functions, such as `geomInside`, `geomStraddle`, and `geomOverlap` might differ when a limit option is used. For example, a `geomOverlap` command might try to select metal that has more than two contacts on it.

The metal within the interaction area contains only one contact.
**Separation/Notch**

Separation and notch checks might not be accurate, because only one interaction area is processed at a time. It might not be possible to distinguish whether two polygons belong to the same shape.

In the dashed-line interaction area there are two unjoined polygons that indicate a separation violation might occur, rather than a notch violation. Because the two polygons belong to the same shape, a notch violation should have been detected.

**DRC Area**

In the separation and notch diagram, the drc area is calculated for only the two parts of the shape intersecting the interaction area. The remaining parts of the shape are not measured. For all shapes composed of multiple parts, any parts outside the interaction area are not included in the calculation of drc area.

**Apposition and Length**

Several polygons from different cells can be abutted to form a long edge. You can select a polygon based on the length of an edge with `geomGetLength`. Unlike flat mode, unless the interaction area covers the entire edge, the polygon is not selected.

In this example, two A cells abut to form the long, narrow figure. The interaction area (shown with a dashed line) covers only part of the polygon. The length check sees only part of the long, narrow figure.
Note that large apposition checks have the same problem because the entire apposition area might not be included in the interaction area.

### Nodal checks

Many of the relational selection operations contain “connection” options, where you specify `sameNet` or `diffNet` values. If you consider a `geomOverlap` command using the connection “sameNet” option, DRC might not see the entire net in the interaction area and therefore generates different results than it would in flat mode.

### Saving Derived Layers

When you run DRC hierarchically, it processes each cell independently and then processes the interaction areas cell-to-cell and cell-to-polygon. Each of these processes implies a pass through the rules.

All functions in the rules including `saveDerived` are therefore executed multiple times by this tool. As a result, the `saveDerived` function might produce unintended results. During the
processing of the interaction areas, the function only processes data in those areas and the results go in the parent cell.

This is not a problem if you intend the saved data to flag an error condition (for example, \texttt{badcon = contact andNot metal}). However, you do not necessarily obtain meaningful results if you intend to generate new layers or replace existing layers.

If you use the \texttt{currentCell} switch name to limit the scope of the command (refer to the \textit{Simulation and Environment Control} chapter), data is saved only during the internal processing of each cell. Data is not saved when this tool processes interaction areas.

**How DRC Performs Checks**

DRC performs checks on pairs of edges. Every edge has an inside and an outside relative to the polygon that it was derived from. Each function relates these edges in a different way. For example, \textit{width} is an inside edge to an inside edge of the same polygon, and \textit{ovlp} is an inside edge to the inside edge of different polygons.

Whether DRC checks between edges depends on their positions to each other relative to their insides and outsides. It also depends on the function being performed. The following figure illustrates the relationships for a \textit{sep} (separation) rule that checks outside-to-outside edge spacing of separate polygons.
The following figure shows how DRC checks between two edges. DRC builds a region around each edge based on the check dimension. Any part of the other edge that intersects that region is in error. DRC creates the default error region for a check as a quadrilateral built from the edges in error.

The measurements between edges are triggered by the end points of the edges. Be careful when you write rules that involve distance checks. For example, consider an enclosure check that flags violations from inside edge to outside edge.

The example on the left flags violations, while the example on the right does not. No distance violation occurs from any of the metal or contact endpoints.
You can use the result of a logical operation as input to the enclosure check.

The resultant figure from the `geomAnd` command has edge endpoints that trigger DRC violations.

Normally, DRC does not check adjacent edges (two edges that meet at a single point in a polygon) because their endpoints touch. This is not true for width and notch checks where DRC checks adjacent edges if the angle between them is less than 90 degrees.

This difference can be eliminated if you specify a lower dimensional limit of greater than 0. For example

```plaintext
drc( metal 0 < notch < 3 )
```
The lower limit specifies that a spacing of 0 between edges is acceptable. The point that the edges meet does not trigger a check. However, the other end points of the edges might possibly trigger a check.

When you specify a range of dimensions in a check (for example, $1 < sep < 3$), using end points to detect DRC violations can have significant effects. If you are checking the separation of angled edges, and both end points of an edge fall outside the range, the edge is not detected as a violation even if the edge itself spans the specified range. The following graphic illustrates an angled edge passing through a DRC check range.

If you do not use the *opposite* option and the dimension range encompasses one or more of the angled edge endpoints, the complete angled edge is flagged as an error.

If you use the *opposite* option and the dimension range encompasses one endpoint of the angled edge, only the section of the edge inside the range is flagged as an error.

If you use the opposite option and the dimension range encompasses both endpoints of the angled edge, this results in two DRC errors (one from each endpoint of the edge).

**Angled Edge Tolerance**

Angled edges are treated differently when doing normal spacing and width checks. If you draw a path in a layout with a defined width, that path does not have exactly the width you
defined because the editor can only put end points of the path edges on a finite integer grid. For example, if you create a path with a width of 1 micron, the edges are 1.4142 microns apart along both the X and Y axes. This means the path might have a width slightly less or slightly greater than 1 micron depending on the size of the layout grid you are using and the angles of the corners of the path.

If DRC checked for an exact dimension on angled edges, these paths might be flagged as errors. For example, if on a 1000 dbu per micron grid you create a horizontal path with a width of 1 micron that turns 45 degrees, the resulting width is .99985 microns. An exact DRC width check for 1 micron flags this an error.

To avoid this situation, DRC automatically introduces a very small tolerance into angled edge checks. The tolerance is always biased in the direction of not producing an error. However, if you use a DRC check which includes an “=” operator (for example, \( 1 \leq sep \leq 2 \)), the tolerance is removed and checks are made to exact dimensions.

**Looking Through the Wall**

DRC checks are normally considered as acting between polygons. However, the DRC program actually considers edge relationships. For example, a spacing check is considered the separation between the outside of one edge and the outside of another edge, where “outside” refers to the polygon from which the edge was taken.

An enclosure check measures the spacing between the outside edge of one polygon and the inside edge of another polygon. For enclosure checks, you might normally think of one polygon nested inside another. However, the check can apply to any polygon configuration. Consider the following diagram, which illustrates the many possibilities for an enclosure check.

Example B is the configuration you might normally consider for enclosure. Example C can also be accepted as normal usage. However, examples A, D, and E show the enclosure measurement going through another edge. This is commonly referred to as “looking through the wall.”
Whether examples A, D and E are correct depends upon what you want. For example, the following external spacing configuration is not normally what you want and this verification tool does not normally generate errors for it (depending on the exact configuration).

This is a situation where the edge you are “looking through” is adjacent to the edge being checked. Configurations C and D are not of this form and the program, by default, always checks them. You have further capabilities for checking these situations using the DRC shielded option, which specifies whether the program can “look through walls” and in what manner it should do so.

You can also control “looking through walls” by using the various edge selection and processing functions to derive only those edges or shapes you want checked. In the case of examples A through E, the operation of selecting edges of the enclosed layer inside the enclosing layer eliminates any edges that cause the “looking through the wall” condition.

**Check Conjunction**

In DRC, derived layers are used to facilitate complex checks. Each design rule check is a separate check generating output. You can direct that output to an error layer or to a temporary file. The output can be edges or polygons, and behaves like any other derived layer.

For input, each check uses an original graphics layer, a derived layer, or the output of another DRC check. In this way DRC achieves “check conjunction” by passing data from check to check through intermediate layers.

For example, the check shown validates metal spacing in the presence of polysilicon that is within one micron of either side of one of the metal edges (single reflection). The first two steps find the metal edges that are close to polysilicon edges. The third step checks the metal separation using both of the previously extracted metal edges. You can apply this concept of check conjunction to double and triple reflections with a minimal number of steps.

```
ref1 = drc( poly metal enc < 1 parallel edgeb )
ref2 = drc( metal poly enc < 1 parallel edgea )
```
Alignment and Registration

There are three main methodologies for checking the separation of shapes in a layout. These methodologies can be simulated individually to some degree using existing checks. The intent of alignment and registration modifiers is to provide the flexibility of using combinations of these methodologies in a single check, without the overhead in processing time associated with the existing methods.

The three methodologies are associated with the process requirements for separation, alignment, and registration, although the checking performed might not necessarily relate directly to the process characteristics these names reference. It is up to you to determine which methodology or combination of methodologies best fits your requirements.

The following sections describe each of these methodologies in detail. All methodologies give the same result for parallel orthogonal edges. It is only at the corners that differences occur.

Separation

This methodology of detecting spacing violations is the most commonly used in DRC programs. Separation has the simple requirement that one shape must maintain a minimum separation from another shape irrespective of the relationship between the shapes. If the required separation is 3 microns, then parallel edges must be at least 3 microns apart, corners must be at least 3 microns from edges, and corners must also be 3 microns from corners. This is independent of the angles of the edges.

The following figure illustrates for this methodology the area around a shape which is inviolate for other shapes. The illustration shows the normal radius at the corner, where all points on the arc are equidistant from the corner.
normalGrow

This methodology of detecting spacing violations is directly related to the normal process of growing shapes. To detect violations, the area around each shape is processed the same as if the original shape was grown by the spacing amount. The angles and configuration of the edges at the shape corners are maintained in the violation area except when an acute angle corner is grown. This is truncated to prevent excessive extension.

squareGrow

This methodology of detecting spacing violations is based on mask movement in processing and has the effect of moving the mask a maximum amount in the X and Y axes. To detect violations, the area around each shape is processed as if the original shape had been misaligned in all directions at the same time.

This is best illustrated in the following diagram which shows the misalignment area and also the way in which it is derived by moving an orthogonal square (whose size is the movement value in X and Y) around the periphery of the polygon.
The characteristics of this methodology are the truncated non-orthogonal corners and the extended expansion on angled edges (root 2 expansion for 45 degree edges).

**Checking Philosophy**

Given the three possible spacing methodologies, you can now apply them in such a way as to simulate your own processing methodology. Since it is unknown what this is, the best method is provide a check that allows for all three methods at once, and provides the capability to selectively determine whether the methods are applied, and with what value.

Normal DRC spacing checks apply a single separation value. Even with the other methodologies, this value is the same for parallel orthogonal edges. The normal separation check and its single value must be the basis for the new check.

The new check is applied only when corner-to-corner, corner-to-edge, or edge-to-edge spacing is required. If the spacing is below the absolute minimum, an error is generated and no complex analysis is done.

The check can be applied to polygon input layers or edge input layers. When used for edges, corners formed by the edges are treated exactly the same as polygon corners. Open ends of edges are treated as 90 degree outside corners.

**Command Syntax**

This syntax is the same as the normal spacing check with the addition of two modifier keywords.

```
drc( layer1 layer2 sep < val1 normalGrow = val2 squareGrow = val3 )
```

The `normalGrow` and `squareGrow` modifier keywords represent the separation methodologies described in the previous sections. The next example explains how their values are applied.

**Note:** The processing of this check is not performed by growing the shapes, although the effect is the same.

**Example**

In this example it is required that the minimum normal separation between shapes is to be 5 microns. However, at the corners you want to allow for a normal grow of 0.8 microns and a square grow of 1.0 microns.

```
drc ( metal poly sep < 5 normalGrow = 0.8 squareGrow 1.0 )
```
How these values are applied is demonstrated in this figure.

The final effect of the check is as if the original shapes had first been grown using the normal grow methodology by an amount of 0.8, then grown using the square grow methodology by an amount of 1.0, and then checked for normal separation by an amount of 1.4, which is 5 minus twice the sum of the other amounts.

**Note:** The normal and square grows are applied to both shapes before the separation is checked.

With “all-angle” edges, the order in which the normal grow and the square grow are applied makes a difference in the end result. The order in which these operations is applied is taken from the order of the keywords in the command. If, in the above example, the square grow was needed before the normal grow, the check would appear as follows:

```plaintext
drc ( metal poly sep < 5 squareGrow = 1.0 normalGrow = 0.8 )
```

If the sum of the modifier values is equal to half the separation value, then no separation check would apply. The sum of the modifier values must not exceed half of the separation value.

If either of the modifiers is not required, their values can be set to zero, or the modifier keyword can be omitted from the check. If both modifiers are set to zero, the result would be a normal spacing check.
These modifier keywords can be applied to the width, notch, overlap, and enclosure commands as well as the separation command. They are not applicable to the area command.
Cell-Based Options

hdrc = mode

Description

For the ivDRC and ivVerify commands, the run modes can be overridden on a cell-by-cell basis by adding a property to the cell. The property overrides the option on the command line for the cell containing the property and all cells below it in the hierarchy.

Fields

hdrc

The property name.

mode

The value of the property enclosed in double quotes. The value can be any one of these single characters as shown, or a string starting with that character. The options are as follows:

F

Flat. Equivalent to the ?hier nil option.

N

Nonoptimized hierarchical. Equivalent to ?hier t plus ?optimize nil.

E

Optimized hierarchical with expanded mosaics. Equivalent to ?hier plus ?optimize, except mosaics are expanded.

U


The references to expansion of mosaics identifies how they are processed in the pattern recognition optimization. In expanded mode, the mosaic planes are treated as contiguous data without interplane boundaries or repetition. In unexpanded mode, the interfaces between the planes are optimized the same as interfaces between cells.

Example

In this example, the cell is checked hierarchically with optimization, while mosaics remain unexpanded.

    hdrc = "U"
Checking Commands

Checking commands are used to check the design rule integrity in the layout.
checkAllLayers

checkAllLayers( [ exceptLayer ] ... )

Description

The checkAllLayers command starts a set of design rule checks based on properties in the technology file. These properties are considered only if they have a purpose of all.

For each of these technology file properties, a check is executed by the program on the layers as defined by the technology file properties, unless you exclude layers from checking with the except_layer option.

- minWidth
- minSpacing
- minNotch
- minEnclosure
- minOverlap

If you do not specify input layers in the checkAllLayers command, the checks executed are all those defined by the DRC check properties.

If you specify any input layers in the checkAllLayers command, the checks executed are those defined by the DRC check properties, excluding the checks using the specified input layers.

A check is not fully generated from a property if that property does not match the criteria of the check. For example, if the property is minSpacing and it has an Order state of t, it cannot be checked because this tool’s spacing check is not ordered.

Fields

exceptLayer

An optional argument consisting of one or more original graphics layer names. Each layer specified in the argument is omitted from the checks executed by the program whether or not the layers have technology file properties.

Examples

The following example is the default command.

checkAllLayers( )
The following example checks all layers except for *tub* and *nwell*.

```plaintext
checkAllLayers( "tub" "nwell" )
```
checkLayer

cHECKLayer( layer1[ layer2 ] )

Description

The checkLayer command initiates a set of one-layer or two-layer design rule checks based on properties in the technology file. These properties are only considered if they have a purpose of “all.”

When you specify the input layer(s), checks are performed if the layer(s) in the technology file property matches the input layer(s).

If the two-layer property minEnclosure exists in the technology file, a check is performed only under these conditions:

- If the layers in the property are the same as the checkLayer input layers and the layers are in the same order.
- If the layers are in reverse order and the property contains the flag “t.” This flag specifies that two checks are required, one with the layers in the specified order, and one with the layers in reverse order.

Fields

layer1

An original graphics layer name. If this is the only layer you specify, the checks executed by this product are those defined by these technology file properties:

- minWidth
- minSpacing
- minNotch

layer2

An optional original graphics layer name. If you specify this layer, the checks executed by the program are those defined by these technology file properties:

- minEnclosure
- minSpacing
- minOverlap
Examples

The following example performs single layer checks of the *poly* layer.

```python
checkLayer( "poly" )
```

The following example performs dual layer checks of the *poly* and *diffusion* layers.

```python
checkLayer( "poly" "diffusion" )
```
drc


Description

The drc command is used for Design Rule Checking (DRC). The complete range of design rule checking is accomplished by the drc command with different options, qualifiers, and keywords.

Fields

outLayer
If this layer is defined, the results of the DRC check are written to that layer. The format is defined by output modifiers. If you do not specify an output modifier, the drc check results are written to the outLayer as error regions. For an area check, the original polygon in error is written to the outLayer. If you do not define an outLayer, the results of the check are written directly to the layout cell view on the marker layer with error purpose.

inLayer1
The first or only layer to be processed. It is a derived layer name or an original graphics layer name enclosed in quotes. When this layer is used with a second layer, the order of the layers can be important. The enc function and the various output modifiers give different results if the first and second layers are reversed.

inLayer2
The optional second layer to be processed. It is a derived layer name or an original graphics layer name. This layer is only required for those functions performing checks between two layers.

function
The check to be performed, consisting of a function keyword and a dimensional limit specification. The function has one of these three syntax forms:

- lower_limit operator function_keyword operator upper_limit
- function_keyword operator upper_limit
- function_keyword operator lower_limit
The lower and upper dimensional limits can be integers, floating point numbers, expressions, or references to layer properties defined in the technology file. The upper limit is required for all functions except area; the lower limit is optional. Both limits are defined in user units (for example, microns).

You can use the following function keywords:

- width
- notch
- area
- sep
- enc
- ovlp

You can use the following operators with the keywords:

- `<`
- `<=`
- `>`
- `>=`
- `==`

The following examples illustrate drc function syntax:

- sep < 5
- 1 < enc < 3.5
- area > 16

Modifiers are used to restrict the scope of a check or modify the output information. There are four different types of modifiers: function, output, alignment and registration, and other.

The different modifiers are listed here by type.

<table>
<thead>
<tr>
<th>Function modifiers</th>
<th>Output modifiers</th>
<th>Alignment and registration modifiers</th>
<th>Other modifiers</th>
</tr>
</thead>
<tbody>
<tr>
<td>with_perp</td>
<td>fig</td>
<td>normalGrow</td>
<td>raw</td>
</tr>
<tr>
<td>only_perp</td>
<td>figa</td>
<td>squareGrow</td>
<td>message</td>
</tr>
</tbody>
</table>
You can use any number of nonexclusive modifiers in a drc command.

<table>
<thead>
<tr>
<th>Function modifiers</th>
<th>Output modifiers</th>
<th>Alignment and registration modifiers</th>
<th>Other modifiers</th>
</tr>
</thead>
<tbody>
<tr>
<td>parallel</td>
<td>figb</td>
<td></td>
<td></td>
</tr>
<tr>
<td>notParallel</td>
<td>edge</td>
<td></td>
<td></td>
</tr>
<tr>
<td>sameNet</td>
<td>edgea</td>
<td></td>
<td></td>
</tr>
<tr>
<td>diffNet</td>
<td>edgeb</td>
<td></td>
<td></td>
</tr>
<tr>
<td>app</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>opposite</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>length</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lengtha</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lengthb</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

You can use any number of modifiers in a check providing they are not mutually exclusive, such as sameNet and diffNet.

**Examples**

The following examples illustrate some of the possible combinations of the drc command.

```plaintext
drc( "metal" width < 3 )
drc( diffn sep < 2 )
drc( "poly" pcut enc < 1.3 )
```

You can use any number of nonexclusive modifiers in a drc command.

```plaintext
drc( gate cut 1 <= sep < 2.1 opposite parallel edgea )
drc( area cut > 16 )
```

You can find values from properties listed in the technology file.

```plaintext
cvId = ivGetCellView()
techId = techGetTechFile(cvId)
drc( "poly" width < techGetLayerProp( techId list( "poly" "drawing" ) "minWidth")
```
drcAntenna

[outLayer] = drcAntenna(
    refLayer
    gate( mode( inLayer1 [inLayer2....] ))
    antenna( mode ( inLayer3 [inLayer4....] ))
    [factor( inLayer5 value1 [inLayer6 value2 ...] )]
    limits
    ["message"]
)

Description

The `drcAntenna` command detects unacceptable antenna ratios. An antenna ratio is the ratio between the area of specific conducting layers and the area of MOS gates on a net. An antenna check is measured for each net in the circuit.

You usually do antenna checks on a partially fabricated circuit. For example, a check might be required for the ratio of `metal1` to gate areas prior fabricating the `metal2` layer. To achieve this partial connectivity in this tool, you must use the antenna check with a `geomConnect` command, which connects only those layers available at the stage of fabrication you specify. To check the complete connectivity of a circuit, you must specify multiple runs with different `geomConnect` commands.

To use the antenna check, all layers involved must be connected or have net numbers derived from connected layers. Normally in Diva verification, the MOS gate layers do not have net numbers because these layers represent devices that are connected to multiple nets. For antenna checking, you must use the `geomStamp` command to assign net numbers to the gate layers. For normal MOS circuits, the appropriate stamping layer is the polysilicon that was used to define the gate layer and to specify the gate (G) connection in the `extractDevice` command.

Fields

* **outLayer**: Optional output layer for nets that fail the required ratio calculation. This verification tool uses this layer in conjunction with the `refLayer`. If you do not specify an output layer, this tool uses the `error marker` layer for output.

* **refLayer**: If a net fails the ratio check, this tool writes all shapes on the `refLayer` for that net to the output layer.
**Diva Reference**

**DRC Overview**

---

**gate**

Keyword introducing the layers to be measured for the gate area of the check. This tool derives a single value of *gate area* for each net in the circuit. It has the form:

\[
gate( \text{mode( inLayer1 [inLayer2....] )} )
\]

You can specify one of three different modes. If you do not specify the *mode* keyword, the default is *sum*.

**sum**

For every net, this product measures the area of each shape on the specified layers (for example, *inLayer1* and *inLayer2*) on that net and sums the areas into a total value.

**min**

For every net, this tool measures the area of each shape on the specified layers (for example, *inLayer1* and *inLayer2*) on that net and chooses the lowest area value.

**max**

For every net, this product measures the area of each shape on the specified layers (for example *inLayer1* and *inLayer2*) on that net and chooses the greatest value of area.

You can specify any number of layers.

**antenna**

Keyword introducing the layers to be measured for the antenna area of the check. This product derives a single value of *antenna area* for each net in the circuit. The *antenna* keyword has exactly the same format as the *gate* keyword.

The antenna ratio is calculated as

\[
\frac{\text{antenna area}}{\text{gate area}}
\]

**factor**

Optional keyword introducing the definition of factors to apply when a net has specific connections that modify the value of the measured ratio. It has the form:

\[
factor( \text{inLayer1 value1 [inLayer2 value2....] } )
\]
The option consists of one or more pairs of layers and values:

<table>
<thead>
<tr>
<th>inLayer</th>
<th>value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Any connected layer.</td>
<td>An integer or floating point value.</td>
</tr>
</tbody>
</table>

If a net connects to the specified layer, this tool uses the factor for that layer to modify the calculated antenna ratio for that net. For example, if the layer is diffusion and the value is 2, a net that connects to diffusion and has an antenna ratio of 200 ends up with an adjusted ratio of 100. This means any net connected to diffusion can have twice the actual antenna ratio before it fails the check.

If you specify multiple layers in the factor option and a net is connected to more than one of these layers, this verification tool takes the value of the first layer you specify that has a connection. The list is prioritized from left to right.

The antenna ratio is calculated as:

$$\frac{\text{antenna area}}{\text{gate area}} \div \text{factor}$$

limits

Specifies the limits this tool applies to the final antenna ratio after the application of any factor.

If the ratio for any net meets the specification in these limits, that net fails the antenna check and an output is generated. (See the outLayer and reflayer field descriptions for more information on output layers.)

You specify the limits as an absolute limit or range using one of the following keywords:

keep

Retains the net as an error if its antenna ratio is within the limits you specify.

ignore

Ignores the net if its antenna ratio is within the limits you specify.
Operators you can use with the keywords are as follows:

- `<`  Less than
- `<=` Less than or equal to
- `>`  Greater than
- `>=` Greater than or equal to
- `==` Equal

You can specify limits by combining a keyword with one or more operators and numbers using normal mathematical conventions. For example:

```
keep > 200
ignore <= 150
100 <= keep <= 300
```

Regardless of the limits you specify, a net is not selected as an error if:

- The gate area is zero.
- The antenna area is zero.
- The factor is specified as zero.
- The final ratio is zero.

"message"

Optional text string enclosed in quotation marks that this product uses to annotate error shapes that are written on the *error marker* layer. This tool assigns the string a suffix with the net number and the final ratio value and appends it to the error shape as a *drcWhy* property. You can view this property using the *Verify - Markers - Explain* menu entry.

**Example 1**

The following example is the simplest application of this command. It checks the ratio of *metal* area to *ngate* area and flags the gates of all nets that have a ratio greater than 200 on the *marker error* layer.

```
drcAntenna( ngate gate( ngate ) antenna(metal ) ignore < 200 )
```
Example 2

The following example uses all options in the drcAntenna command. The keyword \textit{sum} on the \textit{antenna} line is used only for clarity because the default mode is \textit{sum}. The \textit{message} is not used because an output layer is specified, and the \textit{message} applies only to the \textit{marker error} output. The \textit{geomStamp} commands apply net numbers to the gate layers and the \textit{geomCat} command creates an error reference layer containing all layers involved in the check so that the \textit{badAntenna} layer does not contain all gates and metal in error.

```plaintext
ngate = geomStamp( ngate poly )
pgate = geomStamp( pgate poly )
antError = geomCat( ngate pgate metal1 )
badAntenna = drcAntenna(
  antError
  gate( min( ngate pgate )
  antenna( sum( metal1 )
  factor( ndiff 2.0 pdiff 1.7
  ignore < 125
  "Antenna ratio less than 125"

)

Example 3

The following example uses the drcAntenna command to check for nodes that have both \textit{poly1} and \textit{poly2} attached anywhere on the same node, which might be illegal in some processes.

```plaintext
(drcExtractRules
  (poly1 = (geomOr "poly1"))
  (poly2 = (geomOr "poly2"))
  (metal1 = (geomOr "metal1"))
  (cont = (geomOr "cont"))
  (via = (geomOr "via"))
  (via2 = (geomOr "via2"))
  (metal2 = (geomOr "metal2"))
  (metal3 = (geomOr "metal3"))
(geomConnect
  (via cont poly1 poly2 metal1)
  (via via metal1 metal2)
  (via via2 metal2 metal3)
)
/* Find any poly2 on the same node as any poly1 */
/* Note that everything from here to the end of the file is a single command.*/
saveDerived(drcAntenna(
poly2   /* Use "poly2" as the "output" layer for errors  
(The "error flag" is "poly2"). You can use  
another layer here, or you can output the  
entire node by doing a "geomCat" of all  
pertinent layers, and use that layer instead  
of "poly2," for example,  
plp2Out = geomCat(poly1 poly2 metal1 metal2  
metal3)

drcAntenna(plp2Out) */
gate(sum(poly1))   /* Look for total amount of poly1 on each node */
antenna(sum(poly2))   /* "antenna" (error) is defined as  
the sum of the area of the  
"poly2". Any node that has  
"gate" (poly1) and "antenna"  
(poly2) on the node - is an  
error */

"poly2 shares net with poly1") ) /* Message that is  
attached to error flags */

}

Functions

Used with the drc command, DRC functions let you evaluate geometric relationships in a layout.
area

Description

The area function checks the area of individual shapes on a single layer.

Specifying upper and lower dimensional limits are optional. The dimensions you specify are in user units.

area < 16
4 < area < 16
area > 16
area == 16

You can use the raw option to keep the shapes on the input layer separate. Any resultant error shapes are merged on the output layer.

Be careful when you specify limits that define an exact area. During calculations in floating point, systems cannot represent numbers exactly. Values such as 30 might be represented as 29.99999. Such values are compared with the limit, in this example 30, and are found to be different.

The area function might not give meaningful results in hierarchical, incremental, or by-area modes. These modes select shapes within a defined boundary, and might not select entire shapes if the shapes are made up of abutting or overlapping parts. Parts of the shape outside the defined boundary might not be selected.

Examples

The following example flags all cut shapes whose area is less than 12.

drc( cut area < 12 )

The following example flags all shapes of bcw whose area falls into the specified range (greater than 3 but less than or equal to 16).

drc( bcw 3 < area <= 16 )

The following example flags all gate shapes having an area greater than 16.

drc( gate area > 16 )
**enc**

**Description**

The *enc* function measures the enclosure of shapes on the second layer by shapes on the first layer. The enclosure measurement is the distance between inside facing edges of shapes on the first layer to outside facing edges of shapes on the second layer.

The order of the input layers is important. The program verifies the second layer as enclosed by the first layer. Each shape on one layer is checked against every shape on the other layer.

You must specify an upper dimensional limit. The dimensions you specify are in user units. You can specify a lower dimensional limit, but it is not required.

\[
\begin{align*}
\text{enc} &< 4 \\
0 &< \text{enc} < 4.2 \\
3 &\leq \text{enc} \leq 4
\end{align*}
\]

The following figure illustrates the meaning of *enc* in various configurations. The arrows indicate the edge pairs that you can check.

![Diagram](image)

The program checks whether the edges are enclosed, not the shapes. The check does not guarantee that one layer is completely inside another.

**Examples**

The following example flags areas of *metal* that enclose *via* less than 2.

\[
\text{drc( "metal" "via" enc } < 2 )
\]

The following example flags all *poly* edges that face an enclosure of *bcw* of less than 1.5. The resulting *poly* edge is the same length as the facing *bcw* edge.
drc("poly" bcw enc < 1.5 opposite edgea)

The following example flags all gate edges that face inside edges of implant enclosing a gate. The range specified flags distances of greater than 1 to less than 3 only where the edges are parallel.

drc( implant gate 1 < enc < 3 opposite parallel edgeb)
notch

Description

The *notch* function checks the separation of edges forming notches on individual shapes on a single layer. A *notch* is the distance between outside facing edges of the same shape.

You must specify an upper dimensional limit. The dimensions you specify are in user units. You can specify a lower dimensional limit, but it is not required.

\[
\begin{align*}
\text{notch} & < 3 \\
1 & < \text{notch} \leq 3 \\
\text{drc( metal } & 0 < \text{notch} < 3 )
\end{align*}
\]

The following figure illustrates the meaning of *notch* in various configurations. The arrows indicate the edge pairs that are checked.

Adjacent edge checking is described in the description of the *drc* command.

Examples

The following example flags all notches of *metal* whose dimensions are less than 3.

\[
\text{drc( metal notch } < 3 )
\]

The following example flags all notches of *poly* that fall in the range specified (greater than 0 but less than or equal to 4.5).

\[
\text{drc( poly } 0 < \text{notch} \leq 4.5 )
\]

The following example outputs all shapes that have notches less than 1.3 user units.

\[
\text{drc( cut notch } < 1.3 \text{ fig )}
\]
ovlp

Description

The ovlp function measures the overlap of shapes on one layer by shapes on the other layer. Overlap is the distance between an inside facing edge of a shape on one layer to an inside facing edge of a shape on the other layer.

The order that you specify the input layers is not important. Each shape on one layer is checked against every shape on the other layer.

You must specify an upper dimensional limit. The dimensions you specify are in user units. You can specify a lower dimensional limit, but it is not required.

\[
\begin{align*}
\text{ovlp} &< 4 \\
0 &< \text{ovlp} < 4.2 \\
3 &\leq \text{ovlp} \leq 4
\end{align*}
\]

The following figure illustrates the meaning of ovlp in various configurations. The arrows indicate the inside facing edge pairs that are checked.

Although the function ovlp means overlap, it can check shapes that are fully enclosed. The program checks whether the edges overlap, not the shapes.

Examples

The following example flags overlaps of bcw and bcwcut that are less than 2.

\[
\text{drc( bcw "bcwcut" ovlp < 2 )}
\]

The following example flags edges of lapcut that overlap and face metal less than 1.5.

\[
\text{drc( "metal" lapcut ovlp < 1.5 opposite edgeb )}
\]

The following example flags shapes of implant that overlap and are parallel to gate in the range of greater than 1 to less than 3.
Diva Reference
DRC Overview

drc( implant gate 1 < ovlp < 3 parallel figa )
sep

Description

The sep function measures the separation of different shapes on the same layer or on different layers. Separation is the distance between the outside facing edges of shapes.

If you define only one input layer, each shape on that layer is checked against every other shape on the same layer. If you define two input layers, each shape on one layer is checked against every shape on the other layer. The order of the layers is not important to the program.

You must specify an upper dimensional limit. The dimensions you specify are in user units. You can specify a lower dimensional limit, but it is not required.

\[
\text{sep} < 4 \\
0 < \text{sep} < 4.2 \\
3 \leq \text{sep} \leq 4
\]

The following figure illustrates the meaning of sep in various configurations. The arrows indicate the edge pairs that are checked.

Although the sep keyword is an abbreviation for separation, in some cases sep checks shapes that interact. The program bases its checking on whether the edges have separation,
rather than whether the shapes have separation. If parts of your shapes overlap, use the `geomOverlap` command in addition to the `sep` function.

These overlapping layers are missed by a `sep` check and must be detected by a `geomOverlap` function.

**Examples**

The following example flags *metal* to *metal* spacing violations of less than 2.

```plaintext
drc( metal sep < 2 )
```

The following example flags *poly* to *diffn* spacing violations only when the edges of the violating region are parallel and the checked shapes are the same net.

```plaintext
drc( poly diffn sep < 1 parallel sameNet )
```

The following example flags a range of spacing violations of greater than 1 and less than 2.3 between *gate* and *bcw*.

```plaintext
drc( gate bcw 1 < sep < 2.3 )
```
**width**

**Description**

The *width* function checks the width of individual shapes on a single layer. Width is the distance between inside facing edges of individual shapes.

You must specify an upper dimensional width limit. You can also specify a lower dimensional limit. The specified dimensions are in user units.

\[
\begin{align*}
\text{width} & < 3 \\
1 & < \text{width} \leq 3
\end{align*}
\]

The functionality of having only a lower dimensional bound (\(3 < \text{width}\) or in other terms \(\text{width} > 3\)) can be obtained using layer processing functions. A negative size followed by a positive size of the same amount leaves only those shapes whose width is greater than twice the sized amount.

The following figure illustrates the meaning of *width* in various configurations. The arrows indicate the edge pairs that are checked.

![Diagram Illustrating Width in Various Configurations]

The longest dimension of a rectangle is checked by the *width* definition in exactly the same way as the shortest dimension. If both widths of a rectangle are shorter than the *width* check dimension, then two separate errors are written by the program to the output layer.

**Examples**

The first drc finds metal shapes with a width less than 3.

```
drc( metal width < 3 )
```

This drc finds poly shapes with a width greater than 0 and less than or equal to 4.5.

```
drc( poly 0 < width <= 4.5 )
```

This drc finds cut shapes with a width less than 2.
drc( cut width < 2 )

The following example identifies all metal shapes whose width is greater than 3 (twice the size of a negative size followed by a positive size).

geomSize( geomSize( metal -1.5) 1.5 )

Function Modifiers

Function modifiers apply restrictions on the edge relationships for the DRC functions.
app

Description

The *app* modifier is used to check *apposition*. Apposition is the projection of one edge onto the other. You can specify the limits required for the apposition of edges for a check to be made by the program.

The following figure illustrates positive and negative apposition.

![Diagram of apposition](image)

Key: + positive app. - negative app. 0 zero app.

You can use the following apposition values:

- `lower_limit operator app operator upper_limit`
- `app operator lower_limit`
- `app operator upper_limit`

You can use the following operators:

- `<`
- `<=`
- `>`
- `>=`
- `==`

Keep these points in mind when you use *app*:

- When checking two parallel edges, the application and meaning of apposition is clear. If two edges are not parallel to each other, the amount of apposition might depend upon which edge is being used as the reference edge. Also for extreme cases, you might see positive apposition when referencing one edge and negative apposition when referencing the other edge. In this case this product defaults to the positive apposition.
Note: The scenario for *apposition* on angled edges are the same as the scenarios for the DRC *opposite* modifier on angled edges. You can see diagrams of these scenarios in the section, “opposite” later in this chapter.

- The apposition value is independent of the length of the edges checked.
- Apposition is measured only within the range of the checking dimension. If angled edges diverge, the apposition is measured only up to the point where the separation equals the upper limit check dimension.

Examples

The following example checks *poly* to *diffn* spacing of less than 2 but only flags error regions with an apposition greater than 3.

```
   drc( poly diffn sep < 2 app > 3 )
```

The following example flags parallel *metal* edges that are less than 4 apart with an apposition less than 2.

```
   drc( metal sep < 4 app < 2 parallel )
```

The following example flags edges of *cut* enclosed by *metal* by less than 2 whose mutual shapes have an apposition greater than 1.

```
   drc( metal cut enc < 2 app > 1 edgeb )
```

The following example flags edges of *metal* that are parallel to *poly* and less than 2 apart from *poly*, using an apposition that is greater than 5 but less than 20.

```
   drc( metal poly sep < 2 5 < app < 20 parallel edgea )
```
**diffNet**

**Description**

The *diffNet* modifier specifies that a check be made only if the related edges come from polygons on different electrical nets. This modifier only applies to two-layer checks or single-layer separation. You can only use the *diffNet* modifier if you use connected layers.

The *diffNet* modifier is only useful during full flat DRC checking. Incremental and hierarchical checks only verify a portion of the circuit at a time. The connectivity in incremental and hierarchical checking is dependent on which parts of the layout are being analyzed.

If a connection between shapes occurs outside the checking area, those shapes appear to be unconnected although they are partially included in the checking area.

![Diagram illustrating the use of the diffNet modifier](image)

**Example**

A typical use for the *diffNet* modifier would be the same as described for *sameNet*: the separation of polysilicon and diffusion in a MOS technology.

If shapes on these layers are electrically connected, the separation ensures that unwanted transistors are not formed. The separation distance you use is greater than the normal separation you need to ensure adequate electrical isolation of unconnected shapes.

\[
\text{drc}( \text{poly} \ \text{diffn} \ \text{sep} < 2 \ \text{diffNet} )
\]
**length, lengtha, lengthb**

**Description**

The *length, lengtha, and lengthb* modifiers specify that the check be made only if the lengths of the individual edges involved in the check are within specified limits.

Without these modifiers, the edge length is not considered in checking.

**Modifiers**

There are three length modifiers.

- **length**
  The lengths of edges from *both layers* in a two-layer check, and the single layer in a single-layer check must be within the modifier limits.

- **lengtha**
  The length of the edge from the *first layer* in a two-layer check must be within the modifier limits.

- **lengthb**
  The length of the edge from the *second layer* in a two-layer check must be within the modifier limits.

You can use the following value of length syntax:

```
lower_limit operator length operator upper_limit
length operator lower_limit
length operator upper_limit
```

You can use the following operators:

- `<`
- `<=`
- `>`
- `>=`
- `==`

A limit is an integer or floating-point number in user units.
Examples

The following example checks spacing of parallel *metal to metal* lines less than 2 apart, then flags only those regions whose violating edges are greater than 20 in length.

```
drc( metal sep < 2 parallel length > 20 )
```

The following example checks *metal to poly* spacing of less than 2, then flags only violating regions whose *metal* edges are less than 15.

```
drc( metal poly sep < 2 lengtha < 15 )
```

The following example checks *metal* edges parallel to *poly* edges that are greater than 5 and less than 20 long and are less than 2 apart from *metal*.

```
drc( metal poly sep < 2 5 < lengthb < 20 parallel )
```
**notParallel**

**Description**

The `notParallel` modifier restricts checking to edges that are not parallel to each other.

Normally, dimensional checks are applied between all edge configurations other than perpendicular ones. This modifier further limits the checked configurations by only checking edges that are not parallel to each other.

The following figure illustrates `notParallel` modifier checking.

![Diagram illustrating the `notParallel` modifier checking](image)

**Example**

The following example checks `poly` to `diffn` spacing of less than 3 but only flags nonparallel edges of `poly`.

```drc
 drc( "poly" diffn sep < 3 notParallel edgea )
```
only_perp

Description

The *only_perp* modifier restricts checking to only edges that are perpendicular to each other.

Normally, dimensional checks are applied between edges in all configurations other than perpendicular to each other. This modifier limits the checked configurations by only checking edges that are perpendicular to each other.

The following figure illustrates *only_perp* modifier checking.

Use of this modifier excludes some other modifiers, such as *parallel*.

Examples

The following example checks *metal* enclosure of *cut* less than 2 but flags only perpendicular error regions.

```
drc( "metal" "cut" enc < 2 only_perp )
```

The following example flags *cutedge* edges that are perpendicular to *gateedge* and whose spacing is less than 1.5.

```
drc( cutedge gateedge sep < 1.5 only_perp edgea )
```
opposite

Description

The *opposite* modifier restricts checking to only the portion of those edges that are opposite of each other.

The following figure illustrates the *opposite* modifier.

The portion of the edges that get checked is based on a perpendicular projection of each edge. As the figure shows, the meaning of “opposite” is clear for parallel edges.

For the previous figures, this tool generates one of the projection error shapes, however it is not clearly defined which of the two possible errors it produces. In some cases it produces both errors.
The following example shows how *opposite* has totally different meanings, depending on which edge is being used as the reference edge.

![Diagram showing difference in opposite usage](image)

**Examples**

For the previous figures, this verification product generates the error region. It does not miss the error.

The following example flags *poly* to *diffn* spacing less than 2, then trims the default output region.

```
drc( poly diffn sep < 2 opposite )
```

The following example flags regions of parallel *metal* lines separated less than 4 whose apposition is greater than 2.

```
drc( metal sep < 4 app > 2 opposite parallel )
```
**parallel**

**Description**

The *parallel* modifier restricts checking to edges that are parallel to each other.

Normally, dimensional checks are applied between all configurations other than perpendicular ones. This modifier limits the checked configurations to parallel edges.

The following figure illustrates *parallel* checking.

![Diagram illustrating parallel checking](image)

**Example**

You can use the *parallel* modifier to detect reflection conditions between polysilicon and diffusion in a MOS technology.

The following example flags *poly* edges that are parallel to *diffn* and less than 2 apart.

```
drc( "poly" diffn sep < 2 parallel edgea )
```
**sameNet**

**Description**

The *sameNet* modifier specifies that a check is made only if the related edges come from polygons on the same electrical net. This modifier only applies to two-layer checks or single-layer separation. You can use the *sameNet* modifier only on connected layers.

The *sameNet* modifier is meaningful only during full flat DRC checking. Incremental and hierarchical checks verify only a portion of the circuit at a time. The connectivity in incremental and hierarchical checking is dependent on which parts of the layout are being analyzed. If a connection between shapes occurs outside the checking area, those shapes appear to be unconnected even though they are partially included in the checking area.

```
drc( metal poly sep <3 sameNet )
```

**Example**

A typical use for the *sameNet* modifier would be when you check the separation of polysilicon and diffusion in a MOS technology.

If shapes on these layers are electrically connected, the separation ensures that unwanted transistors are not formed. The separation distance you use for this kind of situation is different from the normal separation you need to ensure adequate electrical isolation of unconnected shapes.

```
drc( poly diffn sep < 1 sameNet )
```
**shielded, shieldedA, shieldedB**

**Description**

The *shielded* modifiers change the way DRC reports errors when other edges of the same layers intervene between the edges being checked (when the error between edges is being shielded).

The following figure shows examples of where the *shielded* modifier affects the result.

![Diagram showing examples of shielded modifier effect](image)

The effect of the *shielded* modifier varies depending on whether you have the *opposite* modifier specified also.

Without the *opposite* modifier, the *shielded* modifier eliminates an error only if the intervening edge or shape completely shields the edges being checked. In the examples above, no error is formed for figures 1 and 2, but the error for figure 3 remains unchanged.

With the *opposite* modifier, an error is created between the edges being checked for each part of the edge pairs that have an unobstructed view of each other.

In the examples above, figures 1 and 2 still create no error, but figure 3 generates errors as shown in the following diagram.

![Diagram showing opposite modifier effect](image)

With the *opposite* option, the apposition length of the unshielded errors is well defined. It is illustrated in the previous diagram by the segments A and B. These are two separate appositions and are checked independently against any apposition specification.
The shieldedA and shieldedB options allow the user to control which layer of a two layer check is used as the shield.

**Examples**

The following example flags *poly* to *diffn* spacing less than 2, with both the opposite and shielded modifiers.

```
drc( poly diffn sep < 2 opposite shielded )
```

The following example flags regions of parallel *metal* lines separated less than 4 with only the shielded modifier.

```
drc( metal sep < 4 shielded )
```
with_perp

Description

The `with_perp` modifier checks perpendicular edges in addition to all other meaningful edge relationships.

Normally, dimensional checks between edges are not applied when those edges are perpendicular to each other. This is because any check violations are found with the adjacent edges (those edges preceding or following the perpendicular edges in the polygon). In some cases it is desirable to include the perpendicular edge configuration in the checking.

The following figure illustrates `with_perp` checking.

You can use this modifier when you want to
- Check disjoint edges from other checks that do not include nonperpendicular edges
- Have perpendicular edges in the output

When this modifier is used, your check run time increases. For regular polygon checking, the `with_perp` modifier doubles the number of checks performed by the program.

Examples

The following example includes perpendicular checks and flags all `metal` enclosures of `cut` less than 2 user units.

```plaintext
drc( "metal" "cut" enc < 2 with_perp )
```

The following example includes perpendicular checks and flags edges of `cutedge` that are spaced less than 1.5 from `gateedge`.

```plaintext
drc( cutedge gateedge sep < 1.5 with_perp edgea )
```
Output Modifiers

Output modifiers let you change the format of the output for a DRC check.
**edge, edgea, edgeb**

**Description**

The `edge`, `edgea`, and `edgeb` modifiers specify that the segments of the edges in error become the output of the check rather than the error region between the edges, or the figures themselves. Without these modifiers, or the `fig` modifiers, the error region between the edges is generated as an output.

You can use a variation on the edge modifiers to extend or reduce the length of the edge segments generated. This variation has the following form:

```
edge[a,b] == value
```

The value can be any positive or negative fixed or floating point number in user units. If you specify a value, the edge segments generated by the “edge” modifier are extended at each end by the specified value, although only up to the endpoints of the original edges. If you specify a positive value, the edges are stretched. If you specify a negative value, the edges are shrunk. The program discards any resultant edge length of zero or less.

**Modifiers**

The three edge modifiers and their outputs are as follows:

- **edge**
  The error segments of the edges of both layers, in a two-layer check, and the single layer in a single-layer check.

- **edgea**
  The error segment of the edge of the first layer in a two-layer check.

- **edgeb**
  The error segment of the edge of the second layer in a two-layer check.
The following example illustrates these modifiers.

```
DRC( cut sep < 2 edge )
DRC( metal cut encl < 2 edgeb )
DRC( via gate sep < 2 parallel edge )
DRC( metal poly sep < 2 opposite parallel edgea )
DRC( metal poly sep < 2 opposite parallel edgea == 2 )
DRC( metal contact enc < 1 opposite edgea == 2 )
```

**Examples**

The following example flags cut edges less than 2.
```
drc( cut sep < 2 edge )
```

The following example flags cut edges enclosed by *metal* less than 2.
```
drc( metal cut encl < 2 edgeb )
```

The following example flags parallel *via* and *gate* edges with spacing less than 2.
```
drc( via gate sep < 2 parallel edge )
```

The following example flags those portions of the edges of *metal* shapes that are parallel and separated less than 2 from *poly* shapes.
```
drc( metal poly sep < 2 opposite parallel edgea )
```

The following example is similar to the previous example except that the resultant edges are stretched by 2 microns at each end.
```
drc( metal poly sep < 2 opposite parallel edgea == 2 )
```

The following example illustrates how the extension of an error segment can be used. The increased separation of metal must extend 2 microns either side of the contact.
```
x= drc( metal contact enc < 1 opposite edgea == 2)  
drc( x metal sep < 1.5 opposite)  
drc( metal sep < 1>
```
Note: This drawing is not to scale.
fig, figa, figb

Description

The fig, figa, and figb modifiers specify the output of the check is the figures that the edges originated, rather than the edges themselves or an error region between the edges. Without one of these modifiers or an edge modifier, the error region between the edges is generated as an output.

Modifiers

There are three fig modifiers.

fig

The figures of both layers in a two-layer check, and the single layer in a single-layer check.

figa

The figure of the first layer in a two-layer check.

figb

The figure of the second layer in a two-layer check.

The following figure illustrates these modifiers.

```
layer2

layer1

Default error region

This figure only for figb output

fig output

This figure only for figa output
```

Examples

The following example outputs cut shapes violating spacing of less than 2.

```
drc( cut sep < 2 fig )
```

The following example outputs cut shapes enclosed by metal less than 2.

```
drc( metal cut enc < 2 figb )
```
The following example outputs *via* and *gate* shapes whose edges are parallel and whose spacing is less than 2.

\[
drc( \text{via gate sep } < 2 \text{ parallel fig } )
\]

The following example outputs *metal* shapes parallel to *poly* shapes that violate spacing less than 2. The *metal* shapes are then trimmed to positive apposition.

\[
drc( \text{metal poly sep } < 2 \text{ opposite parallel figa } )
\]

### Alignment and Registration Modifiers

The following section discusses these alignment and registration modifiers:

- normalGrow
- squareGrow
normalGrow

Description

Specify the normalGrow option when you want to simulate the effect of mask misalignment during a DRC dimensional check.

The option only affects the manner in which corner to corner and corner to edge dimensional checks are made.

The effect of this option is the same as if the shape whose spacing you want to check is grown by the dimensional value you specified with this option before the normal spacing check is applied. The shape is not actually grown but the effect of the checking is the same as if it had been.

The form of the grow applied by this option is the same as the grow applied by the geomSize function.

You can use this option in conjunction with the normal DRC functions such as sep, and the registration option squareGrow. When you use normalGrow in conjunction with squareGrow, the order in which you specify the option defines the order in which the original shape is grown.

Refer to the “How DRC Performs Checks” section.

Example

```
drc( layer1 layer2 sep < 5 normalGrow = 0.8 squareGrow = 1.0 )
```

In this example, the effect can be described by

- Applying a normalGrow of 0.8 to both of the shapes involved in the spacing check.
- Applying a squareGrow of 1.0 to the two shapes generated by the previous step.
- Applying a normal separation check of 1.4 to the two shapes generated by the two previous steps. The value of 1.4 is calculated from the original value of 5 minus the sum of both grow values on both shapes, for example, 0.8 + 0.8 + 1.0 + 1.0.
squareGrow

Description

Specify the *squareGrow* option when you want to simulate the effect of mask registration displacement during a DRC dimensional check. The option only affects the manner in which corner to corner and corner to edge dimensional checks are made.

The effect of this option is the same as if the shape whose spacing you want to check is grown by the dimensional value you specified with this option before the normal spacing check is applied. The shape is not actually grown but the affect of the checking is the same as if it had been. The form of the grow is not the conventional grow used by the *geomSize* function.

You can visualize the grow effect of this option by drawing a square whose side dimension is equal to the dimension specified in this option. You trace the square around the outside of the shape you want to check while keeping the edges of the square parallel to the circuit axes. The area encompassed by the trace defines the dimensional bounds of the check.

You can use this option in conjunction with the normal DRC functions such as *sep*, and the alignment option *normalGrow*. When you use it in conjunction with *normalGrow*, the order in which you specify the options defines the order in which the original shape is grown prior to the spacing check.

Refer to the “How DRC Performs Checks” on page 266.

Example

```
    drc( layer1 layer2 sep < 5 normalGrow = 0.8 squareGrow = 1.0 )
```

In this example, the effect can be described by

- Applying a *normalGrow* of 0.8 to both of the shapes involved in the spacing check.
- Applying a *squareGrow* of 1.0 to the two shapes generated by the previous shape.
- Applying a normal separation check of 1.4 to the two shapes generated by the two previous steps. The value of 1.4 is calculated from the original value of 5 minus the sum of both grow values on both shapes., for example, 0.8 + 0.8 + 1.0 + 1.0.

Other Modifiers

*Other* modifiers let you change the format of the input for a DRC check and the message associated with the DRC error flag.
message

Description

The *message* modifier is a text string enclosed in quotes that annotates an error result from a check. The *message* modifier does not limit the conditions of the check.

Each error has a property associated with it that contains a text string. The *Markers – Explain* command in the Verify menu displays this text string when requested. Without the *message* modifier, the text string is the check itself.

Examples

This example flags the separation of *metal to metal* regions, then fixes a message on it.

```
drc( metal sep < 3 "metal separation of 3. Rule 3.1.5 " )
```

This example flags parallel *poly* to *metal* spacings less than 2 and fixes a message to it.

```
drc( metal poly sep < 2 opposite parallel "reflection condition" )
```
raw

Description

Specify the raw modifier when you want the layers checked to remain unmerged during checking. Overlapping figures are not combined into a single figure prior to the checking. The raw modifier only applies to original graphics input layers and not processed layers.

Examples

This example inputs metal data unmerged and checks it for spacing violations of less than 3.

   drc( "metal" sep < 3 raw )

This example inputs poly and cut data unmerged and checks for poly enclosure of cut less than 2.

   drc( "poly" "cut" enc < 2 opposite raw )
Models of Device Extraction

A *device* is an electrical device such as a transistor or capacitor that is intentionally designed into the circuit by the arrangement of mask layers. This is sometimes referred to as a “designed device,” as compared to a parasitic device, which is a side effect of the mask layer arrangement. Designed devices usually appear in the schematic of the circuit.

There are four modes for extracting devices:

- Flat
- Macro cell
- Full hierarchical
- Incremental hierarchical

**Flat Mode**

The normal mode for extraction is *flat*. In flat mode, the Diva verification tool flattens the hierarchy of the layout. This ensures that all structures and interactions forming devices, parameters, and parasitics are recognized. The result of a flat extraction is a single extracted view representing the complete original circuit.

**Macro Cell Mode**

Normally, for the extraction of hierarchical structured layouts, this tool expands the contents of each instance of a cell in the hierarchy into a single flat representation. The flattening process continues until all cell instances are gone, leaving only shapes on layers.

Macro cell mode is similar to flat mode, except that this product treats certain cells as if they are devices. Each macro cell is considered a “black box.” This tool does not flatten a macro cell’s contents into the rest of the layout. The result of a macro cell extraction is a single extracted view in which the macro cells appear only as device instances. When you look at the extracted cellview, instances of macro cells appear the same as normal cell instances. See Chapter 2, “Macro Cell Mode,” for more information on macro cell mode.
Full Hierarchical Mode

In full hierarchical mode, this tool extracts each cell at the lowest level of hierarchy exactly as in flat mode. This verification product then extracts each cell at higher levels of hierarchy and treats the instances of lower level cells as device instances. The contents of higher level cells (other than other cell instances) are extracted as in flat mode. Macro cells can be included in hierarchical extraction.

The result of a full hierarchical extraction is a single extracted view for each cell, in which the other cell instances appear only as device instances. You can use the results of hierarchical extraction the same way as flat extraction.

When you process a hierarchical version with the netlister, the program adds extracted to the representationPath so it can switch between views to trace connectivity through the hierarchy. In addition, this product creates an excell view for each extracted cell. This view provides an interface to the cells that contain it.

Advantages

There are several advantages to running extract in full hierarchical mode.

- Analysis time is reduced for repetitive circuitry because the program analyzes each cell only once, regardless of how many times it appears in the circuit.
- Disk space is reduced because the program maintains only one copy of each extracted cell.
- Virtual memory and graphic display time are reduced because the program partitions the extraction process into smaller sections.
- Errors for a cell are located within the master for that cell rather than on a flattened representation of the circuit.

One of the major advantages of hierarchical extraction is that it reduces the run time. However, reduced run time occurs only if there is cell repetition. Running in hierarchical mode without cell repetition might actually increase run time.

Controlling Hierarchy Selection

You can use the ivCellType property to control the way the program interprets a cell. The ivCellType property can have these values:

- graphic 
  Tells the program the cell is not a level of hierarchy, even if the cell has pins.
macro

Declarations the cell a macro cell. The cell must have pins to be considered a macro cell. If it does not have pins, it becomes a graphic even if you set this value.

If a hierarchical cell has a `ivCellType` property of `macro`, it is treated in the same way as macro cell processing. When the cell is reached in the hierarchy, it is treated as a normal device instance and is not pushed into. No `excell` is created.

**Hierarchical Extraction Tools**

Some of the tools available in the tool's environment that are helpful during hierarchical extraction include:

- **saveDerived**
  This function saves layers you can use with the `excell` version with the `cell_view` keyword. It copies derived layers into the `excell` version so you can do checks against the data in the cell containing the `excell`. See Chapter 5, “copyGraphics” for more information.

- **geomNoholes**
  This function constructs “keep-out” layers that you can save in the `excell` version using the `saveDerived` command. See Chapter 5, “geomNoHoles” for more information.

- **geomGetPurpose**
  This function accesses the cell boundary for copying to `excell`. See Chapter 5, “geomGetLength” for more information.

- **geomAnd**
  This function performs a single layer self-and that you can use to check the overlap of one `excell` boundary onto another. See Chapter 5, “geomAnd” for more information.

- **geomGetMacro**
  This function selects shapes on layers from macro cells. See Chapter 5, “geomGetVertex” for more information.

**Limitations of Hierarchical Mode**

Hierarchical mode limitations include the following:

- The program does not recognize connections between cells that aren’t connected with pins. To prevent the program from missing design errors, you should create keep-out
layers. You can use the keep-out layers to perform dimensional or logical checks between the connection layers and the parent cell data.

- The program performs hierarchical extraction one cell at a time. The excell versions of a cell contain only a skeleton of the data and have no internal connectivity information. Therefore, it is impossible to generate cell-to-cell parasitic measurements during hierarchical extraction. Parasitics and parameters totally contained inside a cell are still valid.

- The program might count parasitic measurements at pins multiple times, once for the original cell, and once for each instance of the excell of that cell. It does this because the area of the pins exists for both. You can avoid this by using the geomGetPurpose command, which allows shapes to be accessed from different levels of the hierarchy. Therefore, you can separate the pins at the current level from pins derived from lower levels of hierarchy.

- The program doesn’t recognize any device formed between cells. You can avoid this by creating “keep-out” layers.

- The hierarchical extraction is only valid with a true hierarchical design methodology. The integrity of the hierarchy and its extraction are violated if cells are modified in any way by the overlay of shapes from other cells, whether they are at the same level or at different levels of hierarchy.

- When processing hierarchical extraction results for LVS, you need separate switching lists for the extracted and schematic circuits. Refer to Chapter 13, “Comparing Layout to Schematic (iLVS)” for more information.

How the Program Processes Hierarchically

In hierarchical mode, the program starts by looking at the master (top level) of the cell containing the entire circuit. Traversing the hierarchy, the program looks for “leaf” cells. Leaf cells contain no instances of other cells.

For each leaf cell, the program performs regular (flat) extraction, creating an extracted version of the cell and an excell view of the cell. The excell view is a simplified version of the original cell containing copies of pins and any other shapes you tell the program to copy from the original cell. You can copy shapes to perform integrity checks on parent cells, or for display purposes.

After the program extracts the leaf cells, it proceeds to the next level up in the hierarchy. At this level, the program replaces instances of cells in the current cell with instances of the excell version of those cells. The program then continues to recognize regular device and connectivity, with the excell pins contributing to the connectivity of the current cell as if they
are normal pins to external connections. The result is an *extracted* version and an *excell* version of the current cell.

The program continues up the hierarchy until it processes all cells including the top level cell. The program creates an *excell* view, even at the top level, for extraction at a higher level.

**Note:** When the program places an *excell* as an instance in another cell, it gives it the same name as the instance in the original cell. If the *excell* instance is the result of flattening a cell and bringing up an instance from a lower level, the program automatically names the instance +1, +2, +3, and so on.

**Using Hierarchical Extraction**

To start hierarchical extraction for interactive processing from a layout window, select the *Extract* command from the Verify menu, then select *hier* for the extract method in the Extract form.

To start hierarchical extraction using SKILL, use the *ivExtract* command to specify a *t* value for the *?hier* argument.

**Incremental Hierarchical Mode**

When you make a layout change to a small number of cells, you can run in incremental hierarchical mode to minimize the amount of processing. In this mode, the program reextracts only those cells that have layout changes within them, as well as cells that contain other cells whose pin connections have changed. Before you can run in incremental hierarchical mode, you must have performed a full extraction on the cell, and an *excell* view must exist for each original cell.

You can use these properties to determine if the layout or extraction rules have changed since the last time the cell was extracted.

- The *instancesLastChanged* property on each layout cell indicates when the cell was last modified.
- The *extractTime* property on the *excell* view indicates when the cell was last hierarchically extracted.
- The *extractSignature* property on the *extracted* view indicates which set of rules was used for the extraction.
Device Recognition

Device recognition is the process by which the program recognizes devices and creates an instance for each device in the extracted version of the circuit.

Using Device Recognition

A single device is recognized by the existence of a single shape on a specific layer. You must use layer processing functions to create a single shape for each device. For example, if you use polysilicon and diffusion shapes in the graphics editor to form a MOS transistor and you use the `geomAnd` command on these layers in Diva verification, you create a shape that represents the MOS transistor. This shape is the device recognition shape.

Once the device recognition shapes are isolated, you direct the program to create the devices in the extracted version by using either the `extractDevice` command or the `extractMOS` command. You use these commands to look at each device recognition shape and determine the electrical connections to the device terminals. To do this, you must have defined the circuit connectivity using the `geomConnect` command.

After the program recognizes and validates the device terminals, it matches the terminal with devices being extracted and places an instance of the device in the extracted version of the circuit. The instance must reference a device model in the system library, and the devices must exist in a library. The program uses the device model to verify the terminal configuration of your extraction command. You define a device model using either the `extractMOS` or `extractDevice` command.

The process of recognizing devices in the circuit automatically creates a cellview containing the extracted representation of the circuit. This cellview is given the default name `extracted` view.

The process of hierarchical extraction also creates a view for each cell in the hierarchy. These views are given the default name `excell` view.

You can override the default cellview names of `extracted` and `excell` by defining your own names. However, you must ensure that the view names you specify do not conflict with any other existing view names for cells that you wish to keep. This tool only checks that the `extracted` and `excell` view names are different.
Device Recognition Polygons

When the program generates devices and interconnects from a layout, it generates an extracted view of the circuit. This version initially contains only devices and nets. The device instances are the symbols that represent the devices. You can make the nets visible by displaying the pin-to-pin direct connections (“fly-lines”). Neither of these displays is associated with the layout mask layers, other than the dimensional location of the device instances.

To see the layout shapes associated with the nets, you use the `saveInterconnect` command. This command lets you save the derived layers used to define the circuit interconnect. When the program saves interconnect shapes in the extracted view, the shapes are associated with the electrical net they helped create. If you probe one of these shapes, the graphics system can locate the net to which the shape belongs and highlight all the shapes associated with that net.

To make device recognition shapes visible, you use the `saveRecognition` command. This command lets you save the shapes that are used to recognize the devices in the `extractDevice` or `extractMOS` commands. Each recognition shape is associated with the device it created, so you can probe a device shape and have the shape and its device symbol highlighted. More importantly, it allows a cross-probe from the LVS program, so you can highlight the device shape directly.

Saving Device Recognition Polygons

The `saveRecognition` command works by storing a property on each device instance. This property is the device recognition shape. To save device recognition polygons, you must:

1. Create a parameterized cell for each device type. You do this in the graphics editor before running any extraction that uses the cell. Default `ivpcells` are in the `samples` library. Refer to the Chapter 8, “About Parameterized Cells” for more information.

2. Define the device type using the `extractDevice` or `extractMOS` commands. The device model (`nfet`, `pfet`, and so forth) remains the same, however, you must define the version as `ivpcell`. Normally, the version default is symbol. `ivpcell` is detailed in the SKILL procedure description.

3. Save the device recognition layer with the option to redefine the graphics layer on which the shapes are displayed. You must do this for each device recognition layer to be saved, using the `saveRecognition` command. The `saveRecognition` command looks at each device recognition shape on the specified layer, determines which device instance it represents, and attaches a property to that instance using the shape coordinates as its contents.
Device Instance Property

When you use the `saveRecognition` command, the program creates a property on the device instance. The property has two parts.

- The property name (`recognitionShape`)
- The property format

The property format is a string that looks like a SKILL list. The first part of the property format is the layer number on which you display the polygons. The second part of the property format lists one or more polygons that form the device shape. Most devices require only a single polygon, however, you might need to specify more than one polygon. For example, you might have to break very large device recognition shapes into multiple polygons in order to conform to the vertex limit of the graphics system.

The format for specifying the shape coordinates is

```
( ( x1 y1 ) ( x2 y2 ) ( x3 y3 ) ... )
```

You specify each coordinate in user units.

For example, the `recognitionShape` displays a 1 by 3 rectangle on layer 10.

```
( 10 ( ( 0 0 ) ( 1 3 ) ) )
```

About Parameterized Cells

A parameterized cell is a special type of cell that has an associated procedure in SKILL. The procedure reads the properties on the cell instance (a device in the extracted cellview) and builds a “variant” of the cell. Instances of the variant are then placed instead of instances of the original cell. For a complete description of parameterized cells, refer to *Virtuoso Layout Editor Help*.

The system library comes with a number of ready-made parameterized cells with view names of `ivpCell`. You can use these cells for saving device recognition shapes. Each cell has a resolution of 1000 database units per micron. You can use the `ivCreatePCells` SKILL procedure to create your own parameterized cells, or to change cells in the system library.

The `ivCreatePCells` procedure copies the pins and graphics from the symbol view of your device to the `ivpCell` view and magnifies them by a value you specify, so they are visible when the cell is placed in the extracted view. `ivCreatePCells` also generates the correct SKILL procedure for storing the recognition shape when you use the `saveRecognition` command.
When you use `extractMOS` or `extractDevice` without the `saveRecognition` command, the extracted view displays the graphics and pins in the `ivpCell` for each device instance. When you use the `saveRecognition` command, you can also see the recognition shape.

**The SKILL Procedure**

You can use the `ivCreatePCells` SKILL procedure to create your own parameterized cells. When Extract places a device instance using the `saveRecognition` command, each logical terminal in the master cell is copied to the variant. The terminals facilitate the circuit connectivity of the device.

The SKILL procedure

- Copies each physical pin from the master to the variant.
- Copies all shapes from the master to the variant.
- Evaluates all recognized properties associated with the instance. In Diva verification, the procedure uses the `recognitionShape` property to create the defined shape on the required layer. It also creates the shape on the instance layer so that the device can be selected. The instance layer shape also allows the device to be seen in unexpanded graphics mode.

**Note:** When you use a symbol view of a cell, you must draw the pins on the cell, instead of using instances of pins. If you do not draw the pins, `ivCreatePCells` creates cellviews without pins. When you do an Extract, the pin names do not match and the extraction fails.

**Processing Pins**

Normally, pins form part of the interconnect of a circuit. If you want to use this connectivity, you must specify the pin’s layer, or one derived from it, in a `geomConnect` command.

If you store a pin layer in the extracted view of the circuit using the `saveInterconnect` function, the system treats the pins as any other shape on that layer and stores them with the net purpose.

During layout extraction, pins are copied to the output cellview (extracted, excell, abstract). Pin instances are copied as pin instances and are not flattened.

Pins have two fundamental uses.

- Pins at the topmost level in the hierarchy are used to label nets and are copied to the output cellview.
Pins are used during macro-cell and hierarchical extraction. The existence of the pins indicate that a cell is a macro cell or a hierarchical cell and tells the program where connections between the parent and child cells are made.

During hierarchical extraction the pin names might be changed by the program to correctly reflect the true connectivity in a cell. If there are two pins with the same name that are not on the same physical net (intended to imply a connection external to the cell), the program renames the second pin and creates a new net for it. The program also adds a property called connectToTerminal to the changed pin. The property contains a reference to the pin that retains the original pin name.

The Diva verification extraction process and the full abgen analysis place a property called netNumber on each pin in the output cellview. The netNumber and connectToTerminal properties are used by the hierarchical and macro cell extraction programs as follows:

- If two pins are on the same net and neither has a netNumber property, the pins are assumed to be connected inside the cell.
- If two pins are on the same net and both have the same netNumber property, the pins are assumed to be connected inside the cell.
- If two pins are on the same net and they have different netNumber properties or only one of the pins has a netNumber property, the pins are assumed to be unconnected in the cell. However the fact that the pins are on the same net indicates they should be connected. This is commonly known as a “must-connect” and an error is generated if the connections in the parent cell to these pins are on different nets.
- If two pins are on different nets but one pin has the property connectToTerminal referencing the other pin, then the pins must be connected in the parent cell otherwise an error message is generated. This is another example of the “must-connect.”

If, for macro-cell extraction, the macro cell is not generated automatically by this product’s extraction or by full abgen, the netNumber property does not exist and you must add this property manually if you want to control the connectivity.

Pin Instances

If a cell contains nothing other than a pin, you need to assign that cell a property of ivCellType with a value of graphic. This ensures that during macro cell extraction the cell is not treated as a macro cell and that during hierarchical extraction it is not treated as another level of hierarchy. With the ivCellType of graphic, the pin instance is expanded into a pin in the parent cell.
Device Extraction Commands

The following section discusses the device extraction commands.
**extractDevice**

```
extractDevice( recLayer termLayer ...
    model [physical] )
```

**Description**

The *extractDevice* function extracts devices of any type from the layout and stores them in an extracted view of the circuit.

**Prerequisite**

The terminal layers (*termLayer*) must be connected layers. The device recognition layer (*recLayer*) must be a derived layer.

**Fields**

- **recLayer**
  - The device recognition layer name. This must be a merged *derived layer name*. Each shape on this layer causes the extraction of a separate device.

- **termLayer**
  - The device terminal definition. There must be a separate terminal definition for each different terminal type on the device. Each terminal definition consists of a derived layer name and any number of terminal name references enclosed in parentheses.

  The terminal layers must be connected layers.

- **model**
  - The device model name with a character string enclosed in quotes or the keyword “nil.”

  If you provide a model name, any device that matches the configuration specified with this command will have an instance of that device model created in the extracted cellview.

  If you provide the keyword “nil,” any device that matches the configuration specified with this command is accepted as valid. However, no instance is created in the extracted cellview and no errors are generated.
All device commands having the same device recognition layer are considered together when the program determines if a device matches the specified configuration. If a device matches any of the configurations specified with this command, the action defined by the command is executed (either a device is created or is ignored). If a device does not match any of the defined configurations, an error is generated.

The character string you use must define the cell name and can optionally define the view name and a library name. If you don’t specify the view name, the program defaults to the name symbol.

Your defined model must be available in a library accessible by the program so that the terminal configurations can be verified and instances of the device can be placed in the extracted view of the circuit. If you specify a library name, the library is searched. Otherwise, the library that contains the layout cellview is searched. If that search fails, all available libraries are searched in arbitrary order.

The following example illustrates model definitions.

"nfet lvs mylib" Use model nfet lvs from library mylib.

"nfet lvs" Search for model nfet lvs in the same library as the layout cellview, and if that fails, search all available libraries in arbitrary order.

"nfet" Search for the nfet symbol in the same library as in case nfet lvs.

physical Optional keyword which specifies that the terminals of this model are matched by physical terminal count, rather than logical terminal count.

The program gathers the terminals of a device by both figure number and node number. Each unique figure number represents a physical terminal. Each unique node number represents a logical terminal. Logical terminals must, by definition, be a subset of physical terminals.

For each device the program detects, it cycles through the provided device definitions and their different terminal types. It
then tries to match either the logical or the physical terminal count as defined by this keyword or lack of keyword.

A device is matched if all its terminals match either the physical or logical counts. A device is not matched if some of its terminals match the physical count and some match the logical net count.

If a device could match more than one model, it matches whichever one it comes across first. There is no precedence for logical or physical counts.

The use of the “physical” option does not necessarily allow for all possible terminal connection scenarios. The following examples illustrate structures which are ambiguous.

- If a device matches one model for physical terminal counts and another model for logical net counts, there is no way for the program to know which is the correct model.
- If a device could match a model because some terminal types match on physical count and some on logical count, this device is not recognized.

### extractDevice Terminal Limit Removed

In previous releases, there was a limit on the number of terminals an individual device could have. This limit of 200 terminals has been removed so that any number of terminals may exist per device.

### Examples

The following example illustrates the `extractDevice` command for a four-terminal MOS transistor with terminal names of “S,” “D,” “G,” and “B.” The device recognition layer is `gate`. The `poly` layer is used for a single terminal, but the `diff` is used to define both “S” and “D.” The device is recognized, even if the source and drain are shorted together, providing that these terminals are formed by independent shapes.

```plaintext
extractDevice( ngate poly("G") ndiff("S" "D") pwell( "B" ) "nfet ivpcell" physical )
```

Diva verification is careful in forming devices. In order to extract a device, Diva must find a shape on the recognition layer that touches or overlaps shapes on each of the specified terminal layers.
There must be exactly the right number of terminals of each type, in this case, exactly one each of "poly" and "pwell" and exactly 2 of "ndiff". If any of these conditions are not met, you will get the "cannot match terminal counts ..." message. The cause will depend on the particular extraction rules you are using and you will need to check each terminal layer to determine the root cause. Use of saveInterconnect for each layer can help in debugging.

**Note:** Diva verification checks the specified terminals against the exact names on the device which will be placed in the "extracted" view. (Capitalization counts: there must be "G," "S," "D," and "B" - and only those terminals - or the Diva verification product will point out the problem and ask you to fix it.)
extractMOS

extractMOS( recLayer gate s_d [bg] model
[ length width scale ] )

Description

The extractMOS function extracts MOS devices from the layout and stores them in an extracted view of the circuit.

It is similar to extractDevice except that it is optimized for MOS transistors and can measure gate width and length without additional measurement commands.

Prerequisites

The terminal layers (gate, s_d, and bg) must be derived and must be connected layers. The device recognition layer must have been merged. The options length, width, and scale must be used together and in order.

Fields

recLayer

Specifies the device recognition layer name. This must be a merged derived layer name. Each shape on this layer causes the extraction of a separate device.

gate

The transistor gate terminal. The definition contains a layer name for the terminal followed by an optional text string in parentheses specifying the terminal name. The default terminal name is “G.” Use this format for the definition.

layer [ ( "name" ) ]

s_d

The transistor source and drain terminals. This definition contains a layer name for the terminals followed by two optional text strings in parentheses specifying the terminal names. The default terminal names are “S” and “D.” Use this format for the definition.

layer [ ( "t_sname" "t_dname" ) ]

bg

The optional transistor back-gate terminal. This definition contains a layer name for the terminal followed by an optional
text string in parentheses specifying the terminal name. The default terminal name is "B." Use this format for the definition.

layer [ ( "name" ) ]

model

The device model name, which is a character string enclosed in quotes.

You must define the model name and the view name, and optionally, the library name in the character string. If you don’t specify the view name, the program defaults to the name symbol.

The defined model must be available in a library accessible by the program so that the terminal configurations can be verified and instances of the device can be placed in the extracted view of the circuit. If you specify a library name, the library is searched, otherwise, the library that contains the layout cellview is searched. If that search fails, all available libraries are searched in arbitrary order.

The following example illustrates model definitions.

"nfet lvs mylib" Use model nfet lvs from library mylib.

"nfet lvs" Search for model nfet lvs in the library that contains the layout cellview. If that search fails, all available libraries are searched in arbitrary order.

"nfet" Search for nfet symbol in same library, as example above.

length

You can optionally specify a name in quotes for the transistor length property. If you do not define this name, the program uses the default name of l. This option must be used with the width and scale options.

width

You can optionally specify a name in quotes for the transistor width property. If this name is not defined, the program uses the default name of w.

scale

You can optionally specify a floating-point number defining the scale of the width and length properties. The scale is relative to
the user units in use. With user units of microns, the default value of 1.0 implies that the width and length are measured in microns. Similarly, a value of 1.0e-6 converts the units to meters.

Caution

extractMOS expects the source/drain to extend beyond the gate. If this is not the case, the system does not always recognize MOS devices.

Problem transistor

Acceptable transistor

Acceptable transistor

Source/drain  Gate

The extractMOS command causes problems whenever it encounters the above configurations. These problems could be due to errors in the device recognition or errors in parameter values.

Use the extractMOS command only for rectangular devices with extending terminals. For all other devices, use the extractDevice command.

It is recommended that you use the extractMOS command only for simple device shapes and when you want very fast device recognition. Although it runs much slower, the extractDevice command has none of the recognition drawbacks of the extractMOS command.

Examples

The following example illustrates the extractMOS command for a four-terminal MOS transistor with terminal names of $S$, $D$, $G$ and $B$, and width and length properties of $w$ and $l$.

```
extractMOS( gate poly diff substrate "nfet" )
```

All the terminal names, properties, and model views are defaulted.

The next example illustrates the extractMOS command for the same device, but explicitly defines the previously defaulted values.
extractMOS( gate poly("G") diff("S" "D") substrate("B") "nfet symbol" "l" "w" 1.0 )

Data Storage
saveInterconnect

saveInterconnect( layer_list )

Description

The `saveInterconnect` function copies shapes into the extracted cellview. If a layer being copied has nodal information because it was being processed in a `geomConnect` command or because it was being processed by functions that maintain net information, it is saved in the extracted cellview with `net` purpose. Any other layer, whether an original graphics layer or a derived layer, is saved with `drawing` purpose.

Fields

layer_list A list of layer definitions. Each definition has one of two syntax forms.

- A single original layer name enclosed in quotes, or a derived layer name. Shapes from the layer are saved in the extracted view on the graphics layer having the same name.
- A single original layer name enclosed in quotes or a derived layer name, followed by a graphics layer name in quotes, all in parentheses. Shapes from the first layer are saved in the extracted view on the second layer.

Examples

The following examples illustrate the `saveInterconnect` command.

Save a single derived layer `poly` on the graphics layer “poly.”

```
    saveInterconnect( poly )
```

Save `diffn` to the graphics layer “diffusion.”

```
    saveInterconnect( ( diffn "diffusion" ) )
```

Save two layers in one command.

```
    saveInterconnect( ( metal "cuts" ) (poly "polyl") )
```
saveProperty

saveProperty( deviceLayer propname expression )

Description

This function saves a preset property on devices created using the extractMOS or extractDevice commands.

Fields

deviceLayer

A device recognition layer reference previously used in an extractMOS or extractDevice command.

This entry can also be a layer derived from a device recognition layer by a selection function such as geomInside or geomGetTexted. For each shape on this layer, the property is saved on the original device from which the shape was derived.

propname

The name of the property, in quotes, to be saved on the devices formed by the deviceLayer.

expression

The contents of the property to be saved on the devices formed by the deviceLayer. This can be either a character string in quotes or a fixed or floating-point number. The number might include an expression involving preset variables.

Examples

The following examples illustrate the saveProperty command.

    saveProperty( gate "type" "nfet" )
    saveProperty( pgate "fan-out-limit" 5 )
    p = "float-property" x = 2
    saveProperty( n_gate p sqrt(x) )
saveRecognition

saveRecognition( recLayer [display_layer] )

Description

The saveRecognition function stores the extracted device recognition shapes used in the extractDevice and extractMOS commands in the extracted cellview and attaches them to the appropriate device instances.

Fields

recLayer The device recognition layer used in the extractDevice or extractMOS commands.

display_layer You can specify an optional graphics layer name enclosed in quotes. If this layer is specified, the shapes from the recognition layer are displayed on it. Otherwise, the shapes are displayed on a graphics layer with the same name as the recognition layer.

Examples

The following example saves the recognition layer gate and displays it on the original graphics layer "gate."

    saveRecognition( gate )

This example saves the recognition shapes from the layer gate and displays them using the "gate_display" graphics layer.

    saveRecognition( gate "gate_display" )
Extracting Parameters (iLPE)

Introduction

Device and connectivity extraction creates an extracted version of the circuit layout that contains instances of devices and nets in the form of a true electrical network.

The verification system provides tools for measuring parameters of those devices, performing calculations with them, and storing the results on the extracted network as parameter properties.

A parameter is a property associated with a device which contains some information relative to that device. These properties can be accessed by application programs or SKILL routines as required. A good example is the netlister, which can annotate a transistor network listing with width and length parameters.

The verification tools measure a wide range of characteristics of devices and can associate any number of them with a single device as parameters. The measureParameter command definition gives full details of the capabilities.

The extractMOS command has built-in parameter extraction of gate width and length.

The commands available for manipulating parameters are as follows:

- measureParameter
  Measures the size of the device and the values of its relationships to other shapes.

- calculateParameter
  Allows calculations to be performed on and between measured values.

- saveParameter
  Saves measured or calculated values as parameter properties on devices.

Parameters are normally associated with a device via the device recognition polygon. They can also be associated with any shape derived from a device recognition polygon, providing
that the shape is created by a selection function, such as `geomOverlap`, rather than a logical function such as `geomAnd`.

**Measurement Process**

The measurement process consists of two functions: `measure` and `calculate`.

**Measure**

The measurement commands associate each measurement with the object being measured. For parameter measurement, this is the device number.

The results from any measurement are given a name. This is a reference to those measurements that can be used in other processing commands.

**Calculate**

A calculation can be performed on a single measurement result (referenced by name), or a calculation can be performed between measurements of results. All measurement names referenced in the calculation must be associated with the same device type.

You cannot mix device measurements with parasitic measurements. Similarly, you cannot mix device measurements of one device type with those of another device type. For example, it is meaningless to calculate the $W/L$ relationship of devices if the $W$ comes from an `nfet` and the $L$ comes from a `pfet`.

The best way to define how the calculation of measured values works is to show an example.

```l = calculateParameter( lc - ( w * b * 0.5 ) )```

In this example, the effective length of a device is calculated as the center line length minus half the width for each bend.

**Measurement Functions**

The measurement functions that are available are detailed in the command reference section of this manual. Some, such as `area` and `perimeter`, are obvious and need no further explanation. Others, however, do need clarification.
Length

The term *length*, when applied to a random polygon, is meaningless. For a rectangle, it could mean the length of the longest axis; but for any more complex shape, no simple definition is possible.

For the purposes of parameter and parasitic measurement, the definition of length of a shape on a layer is defined as the length of edges of that shape relative to shapes on other layers.

An example of this is the length of a transistor gate. In this context it is *the length of the gate shape which is totally inside the polysilicon shape, divided by two*. The division by two is required because the relationship defined measures both sides of the gate.

Many tools are provided to define the relationship. They are as follows:

- over
- not_over
- inside
- outside
- butting
- coincident

The relationship can include up to eight layers.

Bends

The definition of bends is not simple. Consider this figure.

The first two figures are squares with a corner cut out. The first one appears to have a bend in it. Based on the first one having a bend in it, the third figure appears to have two bends in it.
Consider this figure, which represents an MOS transistor gate with source and drain diffusions.

Both gates have bends in them, but one has a bend in the length of the gate and the other has a bend in the width of the gate. The bend is relative to the diffusion layer.

As in the definition of length, the definition of bends depends upon a shape’s relationship to shapes on other layers. The measurement command allows you to define this relationship to other layers as part of the bend determination.

By the program’s definition, a bend is created by an inside corner (concave corner) in a shape, so a count of bends is a count of the inside corners.

In addition to counting bends, you can also use the program for counting corners and angles. A corner is an outside corner (convex corner) in a shape, and an angle is any corner, outside or inside. Therefore, a count of angles is the same as a count of bends plus a count of corners.

**NULL Measurements**

The parameter and figure mode parasitic extraction measurements have these characteristics.

- If a value is measured to be 0, as is the case with bend counts, the value is treated as any other value. If there are no limits specified, the value is retained. If there are limits specified, the value is checked against them and retained if it conforms to those limits.

- If no measurement is made, as is the case of area of gate over metal when there is no metal under the gate, no measurement is created, and a value of zero is created if the limits specified request a value of zero to be retained. If the limits exclude the value of zero, no measurement value will be created.

The difference between a value of zero and no measurement is in creation of the resultant property. A value of zero can be used to create a property with a value of zero. If no measurement is made, a property cannot be created.
Measurement Optimization

For general use, the measurement capabilities of the system are designed to be flexible. As a result, the code cannot be optimized as would be the case if the measurements were made from predefined layer relationships, as in the case of MOS transistor width and length in the extractMOS command.

You can achieve some degree of run-time optimization, however, by using other techniques. These techniques can be controlled by the way you write your measurement commands.

Implied Logical Operations

Inside the measurement command, up to eight layers can be defined in a single complex relationship. For example, with five layers you can write

```
measureParameter( area metal over poly over cut not_over buried_cut not_over tub )
```

Such a relationship is processed by looking at all the layers at once. The program does not do a series of two-layer logical operations to derive the required area. This is optimal unless parts of the relationship are used multiple times, in which case it would be better to extract that area separately with logical functions. In the example, if you had a number of relationships involving poly over cut, it would be more efficient to write

```
pcut = geomAnd( poly cut )
measureParameter( area metal over pcut not_over buried_cut not_over tub )
measureParameter( area metal2 over pcut over well )
```

This optimization has to be considered with the consolidated measurements.

Consolidated Measurements

Multiple measurement commands can be processed in one pass of the data. This capability is controlled by the system command optimizer, which decides if such consolidation is worthwhile.

The optimization decision is based on the number of common layers in a command. For example, if one measurement involves layers a, b, c, and d, and another measurement involves layers b, c, d, and e, then they are consolidated, and both measurements are made at the same time.

The exact optimization criteria is not given here since it is not a simple heuristic.
Any number of measurement commands can be consolidated provided these criteria are met:

- The optimizer decides that the consolidation is worthwhile.
- The resulting number of layers does not exceed the maximum allowed.
- The measurements are contiguous in the command stream; that is, not separated by other commands.

Separate groups of measurement commands in the command stream are optimized separately.

**Parameter Measurement Reference Commands**

The following section contains the parameter measurement reference commands.
**calculateParameter**

\[
\text{outValue} = \text{calculateParameter}( \text{expression} [\text{limit}] )
\]

**Description**

This function creates new device parameter measurement values based on calculations with previously measured or calculated values derived through the `measureParameter` and `calculateParameter` commands.

Each `measureParameter` command allows you to make a single device parametric measurement. This command lets you manipulate one or more of these simple measurements to form more complex parameters.

**Fields**

- **outValue**
  The resultant values are stored in a file which is represented as a numeric value. It cannot be processed by any layer manipulation commands. Only commands related to parameter manipulation can reference the layer name. Specifically, these are `calculateParameter` and `saveParameter`.

- **expression**
  This is the calculation expression relating the results of previous `measureParameter` or `calculateParameter` commands with mathematical operators. The operators can be any of the following:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Function</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>add</td>
<td>x + y</td>
</tr>
<tr>
<td>-</td>
<td>subtract</td>
<td>x - y</td>
</tr>
<tr>
<td>*</td>
<td>multiply</td>
<td>x * y</td>
</tr>
<tr>
<td>/</td>
<td>divide</td>
<td>x / y</td>
</tr>
<tr>
<td>log()</td>
<td>natural log</td>
<td>log(x)</td>
</tr>
<tr>
<td>sin()</td>
<td>sine in radians</td>
<td>sin(x)</td>
</tr>
<tr>
<td>cos()</td>
<td>cosine in radians</td>
<td>cos(x)</td>
</tr>
<tr>
<td>atan()</td>
<td>arctangent of x with range -pi/2 to pi/2</td>
<td>atan(x)</td>
</tr>
</tbody>
</table>
The precedence of the calculation operators follows normal mathematics conventions; that is, multiply and divide have higher precedence than plus or minus with left to right evaluation of operators with the same precedence. You can include parentheses to override the default evaluation order. To avoid conflict with names containing special characters (such as P+), you must use spaces to separate names from operators.

All the layer references from previous measureParameter or calculateParameter commands, used in a calculateParameter
command, must be made from a device recognition layer of the same type. You cannot mix measurements made for different device types. In addition, all measurements in a calculation for a device application must be made from the same device recognition layer.

Each parameter used in a calculateParameter command references data stored in a file which has to be opened by the Diva verification tool. The number of files available when this tool is running depends upon factors outside this tool’s control. To reduce the risk of exceeding the number of files available, a limit of ten parameters within a single calculateParameter command has been set (as a compile-time option). If you exceed this limit, the run does not start. If the required number of files is not available when the calculateParameter command is executed, the program recognizes the fact and stops the run.

If a calculation results in an inappropriate operation, such as dividing by zero, the program issues a warning message and aborts that particular calculation. Such situations are normally the result of a flawed calculation definition, as would be the case if a division is requested by a value that can be measured as zero or nonexistent.

For example

1. Measure area metal over poly.
2. Measure area metal over diff.
3. Divide the result of step 1 by step 2.

If a shape of metal overlaps poly but not diff, there is no second measurement. This results in a value of 0 and the calculation fails.

limit

This optional argument provides the ability to reject calculated values because they fall outside the specified limits. It can be used to significantly reduce the amount of data being created by rejecting all values which are insignificant.

The format of the limits specification can be one of the following:
low_limit op keyword op high_limit
keyword op low_limit
keyword op high_limit

The limits can be integer or floating-point values or mathematical expressions defining such values.

You can use the following operators:

<  
<=  
>  
>=  
==

The keyword can be either ignore or keep.

The following examples illustrate the format:

ignore < .001
keep >= .3
1 < keep < 5
1.3e-4 <= ignore <= 2.7e-3

Example

The following example illustrates the generation of the effective gate length for a device.

\[
\begin{align*}
    \text{av\_length} & = \text{measureParameter}( \text{length gate inside poly 0.5} ) \\
    \text{width} & = \text{measureParameter}( \text{length gate coincident poly 0.5} ) \\
    \text{bends} & = \text{measureParameter}( \text{bends\_all gate inside poly} ) \\
    \text{length} & = \text{calculateParameter}( \text{av\_length} - (0.1 \times \text{bends}) )
\end{align*}
\]

The calculation formulates the effective length from the average center line length minus half the width for every bend.
measureParameter

outValue = measureParameter( operator ( layer1 [function
layer2]... )[coeff][limit] )

Description

This command measures devices or the relationships between devices and other shapes. Those measurements might later be applied as parameters of the devices using the saveParameter command, or contribute to calculations for subsequent parameters using the calculateParameter command.

Prerequisites

You must process the first layer with an extractDevice or extractMOS device command.

Fields

outValue

The measurement results are stored in a file which is represented as a numeric value. It cannot be processed by any layer manipulation commands. Only commands related to parameter manipulation can reference the layer name, specifically calculateParameter and saveParameter.

operator

This argument defines the relationship to be measured. It consists of a single operator followed by a layer reference and optional function-layer reference pairs.

The following are possible operators:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>area</td>
<td>Area of the shape or combinations of shapes defined by the over and not_over functions.</td>
</tr>
<tr>
<td>perimeter</td>
<td>Perimeter of the shape or combinations of shapes defined by the over and not_over functions.</td>
</tr>
<tr>
<td>length</td>
<td>Length of edges conforming to the defined relationship. If the length operator is used, the function argument is required.</td>
</tr>
<tr>
<td>bends_all</td>
<td>Count of all bends in a shape.</td>
</tr>
</tbody>
</table>
The following are possible functions:

- **bends_full**: Count of full bends (>45 degrees) in a shape.
- **bends_part**: Count of partial bends (<=45 degrees) in a shape.
- **bends_rad**: Count of the angles of all bends measured in radians.
- **corners_all**: Count of all outside (convex) corners in a shape.
- **corners_full**: Count of all outside (convex) full corners (>= 90 degrees) in a shape.
- **corners_part**: Count of all outside (convex) partial corners (< 90 degrees) in a shape.
- **corners_rad**: Count of the angles of all corners measured in radians.
- **angles_all**: Count of all corners and bends in a shape.
- **angles_full**: Count of all full corners and bends (>= 90 degrees) in a shape.
- **angles_part**: Count of all partial corners and bends (< 90 degrees) in a shape.
- **fig_count**: Count of figures (shapes) of `inLayer2` enclosed inside `inLayer1`. A shape of `inLayer2` is enclosed in `inLayer1` if it does not cross the boundary of `inLayer1`. The edges of the shapes can be coincident.

  You can use this operator only in conjunction with the `enclosing` function.

- **layer1**: The first layer referenced must have been used as a device recognition polygon in a device extraction command, or derived from a recognition polygon by a selection function such as `geomOverlap`.

- **function**: The functions define the required relationship between the layers referenced before and after them. In these descriptions, the functions are defined relative to a relationship of `layerA function layerB`.

The following are possible functions:

- **butting**: Edge coincidence where the shapes of `layerA` do not overlap shapes of `layerB`. 
Diva Reference
Extracting Parameters (iLPE)

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>coincident</td>
<td>Edge coincidence where the shapes of layerA do overlap shapes of layerB.</td>
</tr>
<tr>
<td>over</td>
<td>For area and perimeter, equivalent to a logical AND of layerA and layerB. For length and bend operators, equivalent to inside plus coincident.</td>
</tr>
<tr>
<td>not_over</td>
<td>For area and perimeter, equivalent to a logical AND_NOT of layerA and layerB. For length and bend operators, equivalent to outside plus butting.</td>
</tr>
<tr>
<td>outside</td>
<td>Edges of layerA outside but not butting shapes of layerB.</td>
</tr>
<tr>
<td>inside</td>
<td>Edges of layerA inside but not coincident with shapes of layerB.</td>
</tr>
<tr>
<td>enclosing</td>
<td>Used in conjunction with the figCount operator to count the number of shapes on inLayer2 enclosed by the shape on inLayer1. You cannot use this function with any other functions or operators. This function applies only to two layers.</td>
</tr>
</tbody>
</table>

**Note:** The area and perimeter operators are shape based and can only measure relationships defined with the over and not_over functions. The length and bend operators are edge based and can measure relationships defined by any of the functions.

layer2
Subsequent layers can be derived layer references or original layer names in quotes.

The sequence of layers and functions is read from left to right. There is a limit of eight layers total for any single measurement.
The following figure illustrates the basic function relationships. The functions other than over and not_over define edges rather than areas.

The following figure illustrates the various bend operator options. The same principles apply to the corner operator options, although the angle measurement is applied to the outside (convex) corners rather than the inside (concave) corners.
The angle of a bend is measured as the deviation of one edge of the bend from being colinear with the other edge, as illustrated in this figure.

![Bend angle measurement](image)

The following figure illustrates a complete multilayer relationship.

![Complete multilayer relationship](image)

**coeff**

An optional coefficient. This is a floating-point number applied as a multiplier to the measured value to change its units. For example, it can be used to change an area measurement in square microns to a capacitance in farads. The choice of the coefficient depends upon the final destination of the data. For example, SPICE™ requires area measurements to be in square meters rather than square microns.

The default value for this coefficient is 1.0.

**limit**

This optional argument provides the ability to reject measured values because they fall outside the specified limits. It can be used to significantly reduce the amount of data being created by rejecting all measured values which are insignificant.

The format of the limits specification can be one of the following:
low_limit op keyword op high_limit
keyword op low_limit
keyword op high_limit

The limits can be integer or floating-point values or mathematical expressions defining such values.

Operators (op) you can use are as follows:

<
<=
>
>=
==

The keyword can be either ignore or keep.

The following examples illustrate the format.

ignore < .001
keep >= .3
1 < keep < 5
1.3e-4 <= ignore <= 2.7e-3

Examples

Since the number of permutations of this command is large, these examples illustrate some possibilities.

trans_length = measureParameter( length ( gate inside poly ) 0.5 )
trans_width = measureParameter( length ( gate coincident poly ) 0.5 )
trans_bends = measureParameter( bends_all ( gate inside poly ) )
poly_perim = measureParameter( perimeter( poly not_over diff )
ignore < .01 )

Data Storage

The following section discusses the saveParameter function.
saveParameter

saveParameter( measurement propname )

Description

This function saves device measurements as properties on the appropriate device instances in the extracted cellview.

Prerequisites

Measurements must have been made by a measureParameter command, or derived from them using the calculateParameter command.

Fields

- measurement: The layer resulting from the measureParameter or calculateParameter commands.
- propname: The name of the property, enclosed in quotes, that is created on the device and references the value of the measurement. The value type of the property is a floating-point number.

Examples

saveParameter( wl "wl" )
saveParameter( width "w" )
saveParameter( length "l" )
Extracting Parasitics (iLPE)

Introduction

During device and connectivity extraction, an extracted version of the circuit layout is produced. The extracted version contains instances of devices and nets in the form of a true electrical network.

The verification system provides tools for measuring parasitics, performing calculations, and storing the results in the extracted network. Parasitic measurements are saved in the extracted network either as parameter properties on devices or as parasitic devices between nets.

Devices in a circuit which are defined by intentional mask interactions are recognized and extracted by the extractDevice and extractMOS commands. These devices usually appear in the schematic of the circuit.

Devices that exist only as side effects of the mask layout and do not normally appear in the schematic are referred to as parasitic devices. A good example of a parasitic device is the capacitance between two shapes on interconnect layers that cross over each other. The parasitic value involved is derived from measurements of the crossover interaction.

The verification tools can measure a wide range of characteristics of the circuit topology and form parasitic devices from them with a range of options as to how the devices are connected in the circuit (between two nets, one net to ground). The measureParasitic, measureFringe, and multiLevelParasitic command definitions give details of the capabilities.

The commands available for manipulating parasitics are detailed in the command reference section of this chapter. The commands are as follows:

- measureParasitic
  Measures the values of topological relationships that occur due to mask layer interaction.

- multiLevelParasitic
  Measures parasitic capacitances from any or all interactions between multiple layers that are ordered by their positions relative to each other.
Diva Reference
Extracting Parasitics (iLPE)

- **measureFringe**
  Measures the values of topological relationships that occur when mask layers do not overlap.

- **complexParasitic**
  Combines the lateral functionality of measureFringe, and the vertical functionality of multilevelParasitic into a single rule. This combination provides greater accuracy by allowing the measurement of more environmental factors that contribute to capacitance.

- **calculateParasitic**
  Allows calculations to be performed on and between measured values.

- **saveParasitic**
  Saves measured or calculated values as parasitic devices inside the extracted network.

- **attachParasitic**
  Saves measured or calculated parasitic values as parameter properties on devices. The values were derived from measurements made of shapes and relationships outside the device and are associated with the device through some direct or indirect interaction.

**Netlisting Parasitics**

If your *schematic* view of the circuit does not contain parasitic devices and your *extracted* view does, you might have conflicting needs for generation of the netlist of the *extracted* view. For SPICE simulation, you do want to include the parasitic devices in the netlist; for LVS checking, you do not want to include them.

This is achieved by having two different views for each parasitic device. One view (called the *spice* view) has the netlisting control commands built in to generate the correct information in a SPICE netlist. The other view (called the *lvs* view) has information that causes it to be ignored by the netlister.

**Note:** For the *lvs* view, the *nlAction* property for parasitic devices must be set to *ignore* so that the devices are not netlisted. The *nlAction* property must be placed in the property list of the device’s cellview.

Two parasitic devices exist in the system library with the *lvs* view already defined. They are as follows:

- **pcapacitor**
  Equivalent to the normal device capacitor
Diva Reference
Extracting Parasitics (iLPE)

- pdiode
  
  Equivalent to the normal device diode

Both have a *symbol* view so they can be seen in the *extracted* view, a *spice* view so they can be netlisted in SPICE format, and an *lvs* view so they can be ignored during netlisting for an LVS run.
Measurement

The measurement process uses the Measure and Calculate commands.

Measure

The measurement commands associate each measurement with one of the following:

- Two electrical net numbers that form the parasitic device terminals
- The measured shape’s number if the measurement is to be attached to a device as a parameter

If different measurements in the same measure command result in the same two net numbers, they are summed together into a single measurement result.

The results from any measurement are given a name. This is a reference to those measurements which can be used in other processing commands.

The measurement commands are measureParasitic, measureFringe, and multiLevelParasitic.

Calculate

A calculation can be performed on the results of a single measurement (referenced by name), between the results of multiple measurements. All the measurement names referenced in the calculation must be associated with the same object. You cannot mix device measurements with parasitic measurements.

The command for calculation is calculateParasitic.

Here is an example:

\[ \text{calculateParasitic}( \text{cap1} + \text{cap2} ) \]

In this example, the object of both measurements is a parasitic value. Every value has two net numbers associated with it. The plus sign in the calculation says that any values in the measurement of cap1 and cap2 that have the same two net numbers are to be summed together to give a value result with the same two net numbers.
Measurement Functions

The specific measurement functions are detailed in the command reference section of this manual. Some, such as *area* and *perimeter*, are obvious and need no further explanation. Others, however, do need clarification.

Length

The term *length* when applied to a random polygon is meaningless. For a rectangle it could mean the length of the longest axis, but for any more complex shape, no simple definition is possible.

For the purposes of parameter and parasitic measurement, the definition of the length of a shape on a layer is defined as the length of edges of that shape relative to shapes on one or more other layers.

A simple example of this is the length of a transistor gate. In this context it is the length of the gate shape that is totally inside the polysilicon shape, divided by two. The division by two is required because the relationship defined measures both sides of the gate.

Several tools are provided to define the relationship. They are as follows:

```
over not_over inside outside butting coincident
```

The relationship can include up to eight layers.

Bends

The definition of bends has problems similar to those of the definition of length.

The first two figures are squares with a corner cut out. Does the first one have a bend in it? If you believe it does, consider the third figure. Does this have two bends in it?
Consider also this figure which represents an MOS transistor gate with the source and drain diffusions.

Both gates appear to have bends in them, but one has a bend in the length of the gate and the other has a bend in the width of the gate. The bend is relative to the diffusion layer.

As in the definition of length, the definition of bends depends upon a shape's relationship to shapes on other layers. The measurement command enables you to define this relationship to other layers as part of the bend determination.

By the program’s definition, a bend is created by an inside corner (concave corner) in a shape, so a count of bends is a count of the inside corners.

In addition to counting bends, you can also use the program for counting corners and angles. A corner is an outside corner (convex corner) in a shape, and an angle is any corner, outside or inside. Therefore, a count of angles is the same as a count of bends plus a count of corners.

**NULL Measurements**

The parameter and figure mode parasitic extraction measurements have these characteristics.

- If a value is measured to be 0, as is the case with bend counts, the value is treated as any other value. If there are no limits specified, the value is retained. If there are limits specified, the value is checked against them and retained if it conforms to those limits.

- If no measurement is made, as is the case of area of gate over metal when there is no metal under the gate, no measurement is created, and a value of zero is created if the limits specified request a value of zero to be retained. If the limits are not specified, or they exclude the value of zero, no measurement value will be created.

The difference between a value of zero and no measurement is in creation of the resultant property. A value of zero can be used to create a property with a value of zero. If no measurement is made, a property cannot be created.
Measurement Optimization

The measurement capabilities of the system are designed to be very flexible for general use. As a result, the code cannot be optimized as would be the case if the measurements were made from predefined layer relationships, as in the case of MOS transistor width and length in the extractMOS command.

However, some degree of run-time optimization is achieved through other techniques. Since these techniques can be controlled by the way you write your measurement commands, an explanation is required.

Implied Logical Operations

Inside the measurement command, up to eight layers can be defined in a single complex relationship. For example, with five layers, you can write the following:

```
area metal over poly over cut not_over buried_cut
not_over tub
```

Such a relationship is processed by looking at all the layers at once. The program does not do a series of two-layer logical operations to derive the required area. This is optimal unless parts of the relationship are used multiple times in which case it would be better to extract that area separately with logical functions. In the previous example, if you had many relationships involving poly over cut, it would be more efficient to write the following:

```
pcut = geomAnd( poly cut )
area metal over pcut not over buried_cut
not_over tub
area metal2 over pcut over well
```

Consolidated Measurements

Multiple measurement commands might be processed in one pass of the data. This capability is controlled by the system command optimizer which decides if such consolidation is worthwhile.

The optimization decision is based on the number of common layers in a command. For example, if one measurement involves layers a, b, c and d, and another measurement involves layers b, c, d and e, then they are consolidated, and both measurements are made at the same time.

The exact optimization criterion are not given here since it is not a simple heuristic.
Any number of measurement commands can be consolidated provided the following criteria is met:

- The optimizer decides that the consolidation is worthwhile.
- The resulting number of layers does not exceed the maximum allowed.
- The measurements are contiguous in the command stream, that is, not separated by other commands.

Separate groups of measurement commands in the command stream are optimized separately.

### Parasitic Measurement Reference Commands

The following section discusses the parasitic measurement reference commands.
**calculateParasitic**

outValue=calculateParasitic( expression [limit] )

**Description**

This is used to create new measurement values based on calculations with previously measured or calculated values derived from the `measureParasitic` or `calculateParasitic` commands.

Each `measureParasitic` command enables you to make a single measurement. This command enables you to manipulate one or more of these simple measurements to form more complex parasitic values.

**Fields**

- **outValue**
  The value results are stored in a file which is represented as a numeric value. It cannot be processed by any layer manipulation commands. Only commands related to parasitic manipulation can reference the layer name. Specifically, these are `calculateParasitic` and `saveParasitic`.

- **expression**
  This is the calculation expression relating the results of previous `measureParasitic` or `calculateParasitic` commands with mathematical operators. The operators can be any of the following:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Function</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>add</td>
<td>x + y</td>
</tr>
<tr>
<td>-</td>
<td>subtract</td>
<td>x - y</td>
</tr>
<tr>
<td>*</td>
<td>multiply</td>
<td>x * y</td>
</tr>
<tr>
<td>/</td>
<td>divide</td>
<td>x / y</td>
</tr>
<tr>
<td>log( )</td>
<td>natural log</td>
<td>log(x)</td>
</tr>
<tr>
<td>sin( )</td>
<td>sine in radians</td>
<td>sin(x)</td>
</tr>
<tr>
<td>cos( )</td>
<td>cosine in radians</td>
<td>cos(x)</td>
</tr>
<tr>
<td>atan( )</td>
<td>arctangent of x with range -pi/2 to pi/2</td>
<td>atan(x)</td>
</tr>
</tbody>
</table>
### Diva Reference
Extracting Parasitics (iLPE)

<table>
<thead>
<tr>
<th>Operator</th>
<th>Function</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>sqrt( )</td>
<td>square root</td>
<td>sqrt(x)</td>
</tr>
<tr>
<td>abs( )</td>
<td>absolute value</td>
<td>abs(x)</td>
</tr>
<tr>
<td>exp( )</td>
<td>exponential function (e to power x)</td>
<td>exp(x)</td>
</tr>
<tr>
<td>ceiling( )</td>
<td>smallest integer not less than x</td>
<td>ceiling(x)</td>
</tr>
<tr>
<td>floor( )</td>
<td>largest integer not greater than x</td>
<td>floor(x)</td>
</tr>
<tr>
<td>round( )</td>
<td>nearest integer (floor(x + 0.5))</td>
<td>round(x)</td>
</tr>
<tr>
<td>log10( )</td>
<td>base 10 logarithm; x must be &gt; 0</td>
<td>log10(x)</td>
</tr>
<tr>
<td>tanh( )</td>
<td>hyperbolic tangent of x</td>
<td>tanh(x)</td>
</tr>
<tr>
<td>sinh( )</td>
<td>hyperbolic sine of x</td>
<td>sinh(x)</td>
</tr>
<tr>
<td>cosh( )</td>
<td>hyperbolic cosine of x</td>
<td>cosh(x)</td>
</tr>
<tr>
<td>acos( )</td>
<td>cos⁻¹ of x in range 0 to pi</td>
<td>acos(x)</td>
</tr>
<tr>
<td>asin( )</td>
<td>sin⁻¹ of x in range -pi/2 to pi/2</td>
<td>asin(x)</td>
</tr>
<tr>
<td>tan( )</td>
<td>tangent of x</td>
<td>tan(x)</td>
</tr>
<tr>
<td>power( )</td>
<td>x raised to power y</td>
<td>power(x,y)</td>
</tr>
<tr>
<td>atan2( )</td>
<td>tan⁻¹ of y/x in range -pi to +pi</td>
<td>atan2(y,x)</td>
</tr>
<tr>
<td>fmod( )</td>
<td>floating-point remainder of x/y with same sign as x; y cannot be 0</td>
<td>fmod(x,y)</td>
</tr>
</tbody>
</table>

The precedence of the calculation operators follows normal mathematics conventions; that is, multiply and divide have higher precedence than plus or minus with left to right evaluation of operators with the same precedence. You can include parentheses to override the default evaluation order. The names have to be separated from the operators by spaces to avoid conflict with names containing special characters (for example, +P).
All the layer references from previous `measureParasitic` or `calculateParasitic` commands (used in a `calculateParasitic` command) must be of the same type. You cannot mix measurements made with the `figure` option of the `measureParasitic` command with node-based measurements made by any of the parasitic measurement commands.

Each parameter used in a `calculateParasitic` command references data stored in a file which has to be opened by the program. The number of files available when the program is running depends upon factors outside the program’s control. To reduce the risk of exceeding the number of files available, a limit of ten parameters within a single `calculateParasitic` command has been set (as a compile-time option). If you exceed this limit, the run does not start. If the required number of files is not available when the `calculateParasitic` command is executed, the program recognizes the fact and discontinues the run.

If a calculation results in an inappropriate operation, such as dividing by zero, the program issues a warning message and aborts that particular calculation. Such situations are normally the result of a flawed calculation definition, as would be the case if a division is requested by a value that can be measured as zero or nonexistent.

For example

```
x = area metal over poly
y = area metal over diff
calculate x/y
```

If a shape/node of metal overlaps poly but not diff, there is no y measurement that results in a value of 0. This calculation then fails.

This optional argument provides the ability to reject calculated values because they fall outside the specified limits. It can be used to significantly reduce the amount of data being created by rejecting all values that are insignificant.

The format of the limits specification can be one of the following:
low_limit op keyword op high_limit
keyword op low_limit
keyword op high_limit

The limits can be integer or floating-point values or mathematical expressions defining such values.

You can use the following operators:

<  
<= 
>  
>= 
==

The keyword can be either ignore or keep.

You can use the following limit specifications:

ignore < .001
keep >= .3
1 < keep < 5
1.3e-4 <= ignore <= 2.7e-3

Examples

The following examples illustrate various permutations of the command.

```
cap = calculateParasitic( c1 + c2 + c3 + c4 ignore < 0.1 )
cap = calculateParasitic( (log(c1) * 1.3e-6 ) + ( c2 * 0.054 ) + 1.4 ignore < 0.1 )
```
complexParasitic

complexParasitic(
    tableName
    (layer list)
    [grounded factor)
    model terminal1 terminal2 property
)

Description

The complexParasitic command provides a capability for extracting parasitic capacitance that is more accurate, although slower, than the multiLevelParasitic command. In addition to increasing the accuracy for total capacitance measurement in normal layouts, this command improves the accuracy for more specialized structures and for cross-coupled capacitance.

It is similar in its methodology to the multiLevelParasitic command, but it measures more layout relationships and uses a look-up table for capacitance derivation rather than equations. The complexParasitic command hides the details of the extraction, which make the syntax very simple, but requires the use of the Coefficient Generator to create the capacitance look-up table.

Unlike the multiLevelParasitic command, this command is completely self contained. One specification of this command initiates all the required capacitance extraction for all layer combinations and relationships (lateral and vertical). It also instantiates the resultant capacitances into the extracted view of the design, hence negating the need for the generation of intermediate storage files and use of the calculateParasitic and saveParasitic commands.

The section, Measurement Types for complexParasitic details the topological relationships being measured by this command.

Prerequisites

Each layer defined for capacitance extraction must be a connected layer or derived from a connected layer by a function that propagates connectivity. You can include resistive layers, but they may not be derived from a connected layer.

The number of layers defined for this command must match the number of layers defined in the capacitance look-up table created by the Coefficient Generator. You must make sure that the Coefficient Generator generated this capacitance look-up table for the same technology as the design being measured.
## Arguments

<table>
<thead>
<tr>
<th>Argument</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>tableName</strong></td>
<td>The name, in quotes, of the capacitance look-up table for the technology of the design being processed. The Coefficient Generator must generate this capacitance table.</td>
</tr>
</tbody>
</table>
| **layer list** | A list, in parentheses, of layer names separated by spaces. For example  

\[(substrate\ poly\ metal1\ metal2\ metal3)\]

The layers must be ordered from the lowest (usually *substrate*) first to the highest last. The number of names must match the number of layers defined in the technology file used to create the capacitance look-up table. The names need not be the same as those in the technology file, but they must be functionally equivalent. For a resistive layer, the layer name must be the name of the layer from which resistance was extracted. |
| **grounded** | Optional keyword that converts the measured parasitic values into two separate measurements between the measured nets and ground. Without this keyword each measurement between two nets is stored as a capacitance with the two nets as its terminal connections. If you specify the **grounded** keyword, each measurement between two nets is converted into two capacitors. Each capacitor will have one of the original nets for one of its terminal connections, and ground as the other terminal connection. |
| **factor** | An optional floating point number used to modify the capacitor values when the grounded option is used. Each grounded capacitance value will be multiplied by this factor. If not specified, the default of 1.0 will be used. |
| **model** | The parasitic capacitor model name consisting of a character string enclosed in quotes. The string must define the model name and optionally the view name and the library name. |

If you specify the **groundNet** command, the ground net for each grounded capacitor is the net specified by the **groundNet** command. Otherwise, the ground net is specified as “0!”. |
view name is omitted, the default is symbol. An instance of this model will be created in the extracted view of the design for each capacitor.

terminal1 The name, in quotes, of the first terminal of the parasitic capacitor model.

terminal2 The name, in quotes, of the second terminal of the parasitic capacitor model.

property The name, in quotes, of the property that is created on the capacitor instance to hold the capacitor value. The value type of the property is a floating point number.

Example

complexParasitic(
    "myTable" (substrate diffusion metal1 metal2 metal3)
    grounded 0.75
    "capacitor" "plus" "minus" "cap"
)

Measurement Types for complexParasitic

There are three different basic measurement types: vertical, single layer lateral and two layer lateral. Each measures different layer relationships. The Coefficient Generator determines the combinations of layers to be measured for each measurement type.

Vertical Measurements

The following vertical measurements will be made between two layers based on the area of overlap of the two layers not including the area shielded by intervening layers. This overlap area is referred to as the caparea.

<table>
<thead>
<tr>
<th>Numbers indicated in the figure</th>
<th>Corresponding description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Area of the caparea</td>
</tr>
<tr>
<td>2</td>
<td>Length of caparea edges over layer 1 (fringe up)</td>
</tr>
<tr>
<td>3</td>
<td>Length of caparea edges over layer 2 (fringe down)</td>
</tr>
</tbody>
</table>
### Numbers indicated in the figure  
**Corresponding description**

<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Length of <em>caparea</em> edges coincident with layer 1 and layer 2 edges (coincident fringe)</td>
</tr>
<tr>
<td>5</td>
<td>Length of <em>caparea edges</em> butting shielding shapes</td>
</tr>
<tr>
<td>6</td>
<td>Average un-shielded extension of layer 1 beyond the <em>caparea</em></td>
</tr>
<tr>
<td>7</td>
<td>Average un-shielded extension of layer 2 beyond the <em>caparea</em></td>
</tr>
<tr>
<td>8</td>
<td>Average separation of other layer 1 shapes from the <em>caparea</em></td>
</tr>
<tr>
<td>9</td>
<td>Average separation of other layer 2 shapes from the <em>caparea</em></td>
</tr>
<tr>
<td>10</td>
<td>Average separation of shielding shapes from the <em>caparea</em></td>
</tr>
<tr>
<td>11</td>
<td>Coverage of the capacitance area by shapes on the layer below layer 1</td>
</tr>
<tr>
<td>12</td>
<td>Coverage of the capacitance area by shapes on the layer above layer 2</td>
</tr>
</tbody>
</table>

**Figure 10-1  Vertical Measurements**
Single Layer Lateral Measurements

The following lateral measurements will be made between shapes on a single layer based on the area \((\text{caparea})\) formed by the separation between the shapes.

<table>
<thead>
<tr>
<th>Numbers shown in the figure</th>
<th>Corresponding description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Average lateral spacing between the shapes</td>
</tr>
<tr>
<td>2</td>
<td>Length for which the spacing applies</td>
</tr>
<tr>
<td>3</td>
<td>The average width of the shape on one side of the spacing.</td>
</tr>
<tr>
<td>4</td>
<td>The average width of the shape on the other side of the spacing.</td>
</tr>
<tr>
<td>5</td>
<td>Coverage of the capacitance area by shapes on the layer below the measured layer.</td>
</tr>
<tr>
<td>6</td>
<td>Coverage of the capacitance area by shapes on the layer above the measured layer.</td>
</tr>
</tbody>
</table>

Figure 10-2 Single Layer Lateral Measurements

Two Layer Lateral Measurements

The following lateral measurements will be made between shapes on two layers based on the area \((\text{caparea})\) formed by the separation between the shapes.

<table>
<thead>
<tr>
<th>Numbers shown in the figure</th>
<th>Corresponding description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Average lateral spacing between shapes</td>
</tr>
<tr>
<td>2</td>
<td>Length for which the spacing applies</td>
</tr>
</tbody>
</table>
### Numbers shown in the figure

<table>
<thead>
<tr>
<th>Number</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Average width of the layer 1 shape on one side of the spacing.</td>
</tr>
<tr>
<td>4</td>
<td>Average width of the layer 2 shape on the other side of the spacing.</td>
</tr>
<tr>
<td>5</td>
<td>Coverage of the capacitance area by layers between the measured layers (if applicable)</td>
</tr>
<tr>
<td>6</td>
<td>Average separation of shapes of layer 1 from the caparea</td>
</tr>
<tr>
<td>7</td>
<td>Average separation of shapes of layer 2 from the caparea</td>
</tr>
<tr>
<td>8</td>
<td>Coverage of the capacitance area by shapes on the layer below layer 1</td>
</tr>
<tr>
<td>9</td>
<td>Coverage of the capacitance area by shapes on the layer above layer 2</td>
</tr>
</tbody>
</table>

#### Figure 10-3 Two Layer Lateral Measurements

![Diagram of two layer lateral measurements](image_url)
measureFringe

outValue = measureFringe( layer1 [layer2] calculate
  [print1s "filename"] [grounded] [ML Flayer] [limit]
drc_command )

Description

This function measures the relationships between shapes where the relationship can be
defined with a DRC command. This means relationships that do not involve layer overlap or
interaction, such as fringe or sidewall capacitance, can be measured.

The program bases its measurement on the edges of the layers that conform to the defined
DRC relationship. It is limited to measuring the length and separation of those edges. Since
these two parameters are available in each measurement, the measureFringe command
must contain sufficient information to derive a single measurement result from them. To this
end, the command contains its own formula definition.

The two shapes involved in each DRC relationship, whether they be from one or two layers,
provide the net numbers that are associated with the measurement for the parasitic device
result. If both net numbers are the same, no measurement is made. All measurements with
the same two net numbers are summed together.

You can also use this function to pass on information about lateral shape relationships to the
multiLevelParasitic command so you can adjust the vertical capacitance measurement based
on the lateral capacitance configurations, or you can generate an additional lateral capacitor
to compensate for the vertical capacitance configurations.

Prerequisite

The layers referenced must be connected layers.

Fields

outValue

The measurement results are stored in a file which is represented as a numeric value. It cannot be processed by any
layer manipulation commands. Only commands related to parameter and parasitic manipulation can reference the layer
name. Specifically, these are calculateParasitic and saveParasitic.
layer1
The primary layer to be measured. This must be a connected layer.

layer2
An optional second layer to be processed in the measurement. If this layer is not specified, the DRC check defined must be sep (separation) so as to be applicable to a single layer. With this second layer defined, the DRC check is applied to both layers.

calculate
The DRC command finds pairs of edges that conform to the defined relationship. Pairs of edges are measured individually to determine their length and separation. This calculate statement derives a single measurement result from these two parameters. The statement consists of the keyword calculate followed by the formula in parentheses. Within the formula, the length and separation of the edges are represented by the symbols l and s.

The allowable functions in the formula are the same as those allowed in the calculateParasitic command. These functions are shown in the table.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Function</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>add</td>
<td>x + y</td>
</tr>
<tr>
<td>-</td>
<td>subtract</td>
<td>x - y</td>
</tr>
<tr>
<td>*</td>
<td>multiply</td>
<td>x * y</td>
</tr>
<tr>
<td>/</td>
<td>divide</td>
<td>x / y</td>
</tr>
<tr>
<td>log()</td>
<td>natural log</td>
<td>log(x)</td>
</tr>
<tr>
<td>sin()</td>
<td>sine in radians</td>
<td>sin(x)</td>
</tr>
<tr>
<td>cos()</td>
<td>cosine in radians</td>
<td>cos(x)</td>
</tr>
<tr>
<td>atan()</td>
<td>arctangent of x with range -pi/2 to pi/2</td>
<td>atan(x)</td>
</tr>
<tr>
<td>sqrt()</td>
<td>square root</td>
<td>sqrt(x)</td>
</tr>
<tr>
<td>abs()</td>
<td>absolute value</td>
<td>abs(x)</td>
</tr>
<tr>
<td>exp()</td>
<td>exponential function (e to power x)</td>
<td>exp(x)</td>
</tr>
<tr>
<td>ceiling()</td>
<td>smallest integer not less than x</td>
<td>ceiling(x)</td>
</tr>
<tr>
<td>floor()</td>
<td>largest integer not greater than x</td>
<td>floor(x)</td>
</tr>
</tbody>
</table>
The symbols \( s \) and \( l \) are the only names allowed in the calculation. Names generated by other `measureParasitic` and `calculateParasitic` commands are not allowed. Otherwise, all normal calculation capabilities are available.

The calculation is applied to each pair of edges before any summing of measurements having the same net numbers.

Typical examples of `calculate` commands would be:

\[
\text{calculate}( \frac{l}{s} ) \\
\text{calculate}( \frac{l}{\log(s) + 2.0} )
\]

Normal mathematical order of precedence applies.

If a calculation results in an inappropriate operation, such as dividing by zero, Diva verification issues a warning message and aborts that calculation.
printls

An optional keyword you use to request text file output of the measured length and spacing values. You use this keyword in conjunction with a file name.

printls "filename"

The text file with the name “filename” is generated in the current directory. It has the following format:

;Node name to number mapping.
vdd 1
gnd 9
q* 13
q 15
;length spacing net1 net2
5.500000 1.000000 1 13
2.500000 0.500000 1 13
2.500000 0.500000 1 15
9.500000 1.000000 9 15

The file consists of two sections. Each section is introduced by a single comment line preceded by a semicolon. The first section maps the user-defined labels to the internal net numbers. The second section lists each fringe measurement in terms of its length, spacing, and the two net numbers between which the measurement was made.
The *measureFringe* command operates in the following sequence:

1. Finds the next pair of edges meeting the DRC criteria
2. Measures the length and spacing between the edges
3. Calculates the capacitance between the edges
4. Determines the net numbers of the edges
5. Adds the new calculated capacitance value to the total for the two nets
6. If there are more edges, returns to step 1
7. If there are no more edges, filters the total capacitance for each pair of nets by the values in the limits option. If you did not specify a limits option, the command is complete.

This option lists the original length and spacing values measured in step 2 prior to any calculation, summation, or filtering. For any pair of net numbers, the program can create multiple entries in the file where each entry represents a different pair of edges of the shapes forming those nets.

If a file exists in the current directory with the same name as the “filename” specified with this option, that file is overwritten. To use this option with multiple *measureFringe* commands, you must specify a different file name for each usage; otherwise, each file generated overwrites the file generated by the previous command.

The program optimizer normally eliminates any command whose output is not used. If you specify this option in the *measureFringe* command, even if the *outValue* is not used elsewhere, the command executes and the text file is created.

**grounded**

Optional keyword that converts the measured parasitic value into two separate measurements between the measured nets and ground.

Without this keyword, each measurement between two nets is stored relative to those two nets. If you specify the keyword, each measurement between two nets is converted into two
measurements, both having the same value as the original measurement. Each measurement is stored relative to one of the nets and ground.

If you specify the *groundNet* command, the ground net is the net specified in the *groundNet* command. If you do not specify *groundNet*, the ground net is specified as 0!.

The summation of all values for any given net pair remains unaffected by this keyword. However, with the keyword, all measurements for any net for a single *measureFringe* command are summed together because the second net is always ground. You can then use the *calculateParasitic* command to sum all measurements for each net into a combined value for the net.

*limit*

Optional argument that lets you reject measured values because they fall outside the specified limits. It can be used to significantly reduce the amount of data being created by rejecting all measured values which are insignificant.

The check for limits is made after the measured values have been summed for the same net numbers.

The format of the limits specification can be one of the following:

```
low_limit op keyword op high_limit
keyword op low_limit
keyword op high_limit
```

The limits can be integer or floating-point values or mathematical expressions defining such values.

You can use the following operators:

<  
<=  
>  
>=  
===

The keyword can be either *ignore* or *keep*.

You can use the following limit specifications:
ignore < .001
keep >= .3
1 < keep < 5
1.3e-4 <= ignore <= 2.7e-3

ML
Keyword that introduces the name of the layer, Flayer, that passes on information to the multilevelParasitic command.

Flayer
Layer that passes on information to the multilevelParasitic command and supplies an output layer you can use as the input in subsequent multilevelParasitic commands.

Whenever this verification product finds a measureFringe layer configuration that matches the DRC specification, the relevant information on the associated edges and the spacing value are stored in the Flayer.

For further information on the use of the ML Flayer option, refer to the “multilevelParasitic” on page 400.

drc_command
The DRC check that defines the relationship to be measured. It uses the standard DRC format for specifying the check, dimensional range, and options. The only valid checks are sep, enc, and ovlp. The options diffNet and edge are always in effect. Examples of the command format are as follows:

sep < 2 parallel opposite
1 < sep < 3 app >s 4

The following figure illustrates how the length and separation measurements are derived from the edges extracted through the DRC command. The following example shows the sep command.
The length \( l \) is the average length of the two edges and the separation \( s \) is the average separation between them.

Whether the \( sep \) command is used for fringe or side-wall capacitance, or \( encl \) and \( ovlp \) are used for some other relationship, the result is always the measurements between two edges.

Examples

The number of permutations of this command is large. The following examples illustrate some possibilities.

\[
\text{fcap} = \text{measureFringe( metal calculate ( l / s ) sep < 3 parallel opposite )}
\]

\[
\text{mp} = \text{measureFringe( metal poly calculate( sqrt(l + 3) / log(s) ) ignore < 0.1 1 < sep <= 3 parallel )}
\]

\[
\text{lap} = \text{measureFringe( poly1 poly2 calculate( l * 1e-6) ignore < 1e-6 ovlp <= 1 )}
\]
measureParasitic

outValue = measureParasitic( operator ( layer1
   [function layer2]...) [coeff] [application] [grounded]
   [polarized] ) [limit] )

Description

This function measures shapes or the relationships between shapes to determine parasitic
characteristics. You can later store those measurements as parasitic devices using the
saveParasitic command, or you can use them in calculations using the calculateParasitic
command to derive more complex values for subsequent parameters or parasitics.

Prerequisites

If the net options are used, at least the first layer referenced must have been a connected
layer.

Fields

outValue

Results of the measurement are stored in a file which is
represented as a numeric value. It cannot be processed by any
layer manipulation commands. Only commands related to
parasitic manipulation can reference the layer name. Specifically,
these are calculateParasitic and saveParasitic.

operator

This argument defines the relationship to be measured. It
consists of a single operator followed by a layer reference and
optional function-layer reference pairs.

The possible operators are as follows:

area

Area of the shape or combinations of shapes defined by
over and not_over functions.

perimeter

Perimeter of the shape or combinations of shapes defined
by over and not_over functions.

length

Length of edges conforming to the defined relationship.

bends_all

Count of all bends in a shape.
bends_full  Count of full bends (>45 degrees) in a shape.
bends_part  Count of partial bends (<=45 degrees) in a shape.
corners_all  Count of all outside (convex) corners in a shape.
corners_full  Count of all outside (convex) full corners (>= 90 degrees) in a shape.
corners_part  Count of all outside (convex) partial corners (< 90 degrees) in a shape.
angles_all  Count of all corners and bends in a shape.
angles_full  Count of all full corners and bends (>= 90 degrees) in a shape.
angles_part  Count of all partial corners and bends (< 90 degrees) in a shape.
fig_count  Count of figures (shapes) of inLayer2 enclosed inside inLayer1. A shape of inLayer2 is enclosed in inLayer1 if it does not cross the boundary of inLayer1. The edges of the shape can be coincident.

You can only use this operator in conjunction with the enclosing function.

layer1  The first layer reference must be a connected layer if the application is one of the net options.

function  The functions define the required relationship between the layers referenced before and after them. In these descriptions, the functions are defined relative to a relationship of layerA function layerB.

Possible functions are as follows:

butting  Edge coincidence where the shapes of layerA do not overlap shapes of layerB.
coincident  Edge coincidence where the shapes of layerA do overlap shapes of layerB.
over  For area and perimeter, equivalent to a logical AND of layerA and layerB. For edge-based operators, equivalent to inside plus coincident.
not_over

For area and perimeter, equivalent to a logical AND_NOT of layerA and layerB. For edge-based operators, equivalent to outside plus butting.

outside

Edges of layerA outside but not butting shapes of layerB.

inside

Edges of layerA inside but not coincident with shapes of layerB.

The area and perimeter operators are shape based and can only measure relationships defined with the over and not_over functions. The length and bends operators are edge based and can measure relationships defined by any of the functions.

enclosing

Used in conjunction with the figCount operator to count the number of shapes on inLayer2 enclosed by the shape on inLayer1.

You cannot use this function with any other function or operators. This function applies only to two layers. You can use this function only with the figure option.

layer2

Subsequent layers can be derived layer references or original layer names in quotes. In some cases, dependent upon the node options requested, the layer must be derived and have nodal information.

The sequence of layers and functions is read from left to right. There is a limit of eight layers total for any single measurement.

The following figure illustrates the basic function relationships.
**Note:** The functions other than *over* and *not_over* define edges rather than areas.

The following figure illustrates the various bend operator options. The same principles apply to the corner operator options, although the angle measurement is applied to the outside (convex) corners rather than the inside (concave) corners.

The angle of a bend is measured as the deviation of one edge of the bend from being colinear with the other edge as illustrated in this figure.
The following figure illustrates a complete multilayer relationship.

A over B not_over C not_over D

**coeff**

An optional coefficient. This is a floating-point number applied as a multiplier to the measured value to change its units. For example, it can be used to change an area measurement in square microns to a capacitance in farads. The choice of the coefficient depends upon the final destination of the data. For example, SPICE requires area measurements to be in square meters rather than square microns. The default value for this coefficient is 1.0.

**application**

The application is used to define how the measurements are to be processed. Primarily there are two modes; one for creating parasitic devices and one for measuring general shapes, usually used for creating parameters on devices (see the attachParasitic section in this chapter). The parasitic device application has a number of alternative modes used to define how the parasitic device terminals are to be connected.

**figure**

Merged layer shapes are being measured, regardless of devices or interconnect. The results are stored per shape and can be used by the attachParasitic command to add parameters to devices measured from any directly or indirectly associated shapes.

**two_net**

Each parasitic measurement must be associated with two electrical nets representing the terminals of the parasitic device. The first net is derived from `layer1` (the first layer referenced in the command). The second net is derived from the next layer referenced in the command, which is preceded by a function of `butting`, `coincident`, `over`, or `inside`. 
**Note:** If a layer is preceded by *not_over* or *outside*, it cannot provide subsequent nets because a measurement is made if the layer is not there.

If the second net is the same as the first net, that measurement is ignored.

If you do not specify a second layer in the command, or if all subsequent layers are preceded by *not_over* or *outside*, a default second net number is allocated. The second net is the net name you specify with the *groundNet* command. If no *groundNet* is specified, the net number is 0!

The net number 0! is used because 0 is a net number that doesn’t occur during normal extraction, so it cannot conflict with any other net number. The ! adopts the convention used by the schematic editor to designate a global net.

All measurements made within a single *measureParasitic* command that have the same pair of nets allocated are summed together to form a single value for the net pair. If you use the *polarized* option, the values are only summed if the nets occur in the same order (see the *polarized* option). You can process the final results by specifying the *calculateParasitic* command or by saving them as parasitic devices using the *saveParasitic* command.

**one_net**

This application is the same as *two_net* except the measurement is associated only with the net specified by *layer1* (the first layer referenced in the command). The second net is always defined by the *groundNet* command. If you do not specify *groundNet*, the net is named 0!, regardless of the other layers and functions referenced.

All measurements made within a single *measureParasitic* command that have the same net allocation are summed together to form a total value for the net.

**three_net**

The net numbers for this application are selected in the same way as the *two_net* application, with the following addition. It selects a third net number from the next layer encountered which is preceded by a function other than *not_over* or *outside*. If the third net number is the same as the first net number, the measured value is ignored.
Here is an example:

If you want the area of layer A over layer B, but want the parasitic device to be connected between layer A and the background layer, you would write the measurement as A over background over B, and use the `two_net` application. However, if the measurement doesn’t apply when layers A and B are the same net, then a spurious measurement is made. The `three_net` application enables you to reject the measurement in this case.

```
A over B wanted as a capacitor to bkgnd
```

```
A is same net as B so no capacitor wanted
```

Optional keyword that converts the measured parasitic value into two separate measurements between the measured nets and ground.

Without this keyword, each measurement between two nets is stored relative to those two nets. If you specify the keyword, each measurement between two nets is converted into two measurements, both having the same value as the original measurement. Each measurement is stored relative to one of the nets and ground.

If you specify the `groundNet` command, the ground net is the net specified in the `groundNet` command. If you do not specify `groundNet`, the ground net is specified as 0!.

The summation of all values for any given net pair remains unaffected by this keyword. However, with the keyword, all measurements for any net for a single `measureFringe` command are summed together because the second net is always ground. You can then use the `calculateParasitic` command to sum all measurements for each net into a combined value for the net.
The polarized option is not meaningful if you use it with the grounded option. If you specify both of these options, it will be flagged as illegal.

The one_net option has the same effect as the grounded option, so you do not need to specify both. However, if you do specify both one_net and grounded, it will not be flagged as illegal.

Additional option valid only with the parasitic device applications of one_net, two_net, and three_net. Without this option, all measured values in a single measureParasitic command are summed if they have the same net numbers, regardless of which layers contributed to the numbers. For example, if you measure the area of layer A over layer B and have two results, one with layer A as net 1 and layer B as net 2, and the other with layer A as net 2 and layer B as net 1, the measurements are summed into a single value between nets 1 and 2.

With this option, the measurements are only summed if the nets are the same numbers as the same layers. In the previous example, the result is two measurements, one between nets 1 and 2 and the other between nets 2 and 1. This facilitates the creation of directional parasitic devices such as diodes. The saveParasitic command allows you to define which layer is to form which terminal of the parasitic device result.

This optional argument provides the ability to reject measured values because they fall outside the specified limits. It can be used to significantly reduce the amount of data being created by rejecting all measured values that are insignificant.

For the parasitic measurements, the check for limits is made after the measured values have been summed for the same net numbers.

The format of the limits specification can be one of the following:

- low_limit op keyword op high_limit
- keyword op low_limit
- keyword op high_limit

The limits can be integer or floating-point values or mathematical expressions defining such values.
You can use the following operators:

<  
<=  
>  
>=  
==

The keyword can be either ignore or keep.

You can use the following limit specifications:

ignore < .001  
keep >= .3  
1 < keep < 5  
1.3e-4 <= ignore <= 2.7e-3

Examples

The number of permutations of this command is large. The following examples illustrate some possibilities.

cap = measureParasitic( area ( poly over metal )  
    0.03 two_net )

diode = measureParasitic( area ( ndiff over pbase not_over ntub ) two_net polarized )
multiLevelParasitic

outValue = multiLevelParasitic(
    layers( layer1 layer2 ...)
    cap( layerA layerB c1 c2 [c3[c4]]
        [shield( layerS c5)
            [fringe( layerAB layerF [vertical( ...s...)]])
        [grounded]
        [limit]
    )
)

Description

The multiLevelParasitic function measures the areas and edge lengths of multilayer shape relationships in order to create parasitic capacitance. This function differs from the measureParasitic command because it can handle layer priority with shorted and intervening layers for any stack of layers.

You can use the saveParasitic command to store the measurements later as parasitic devices, or you can use the calculateParasitic command for calculations to derive more complex values for subsequent parasitics.

Prerequisites

Each layer that forms a terminal of a capacitor must be a connected layer or derived from a connected layer by a function that propagates connectivity.

Fields

outValue

Results of the measurement. The results are stored in a file that is represented as a numeric value. The result cannot be processed by any layer manipulation commands. Only commands related to parasitic manipulation (calculateParasitic and saveParasitic) can reference the layer name.

The results of the measurement are a separate capacitance value for each pair of electrical nets encountered during the layer-to-layer measurement process. Values are not kept separate for each pair of layers involved.

layers

A list of the layers to be processed. This argument has the following format:
layers( layer1 layer2 ........layerN )

**Note:** This command can process a maximum of 16 layers at one time. This limit includes layers specified in the layers list, plus any special layers referenced in the fringe option.

The order of the layers defines the priority. The first entry is considered the lowest layer in the layout and the last entry is considered the highest layer in the layout. The list must contain the initial keyword *layers*. The list must contain at least two layers. If a layer is to be used as the terminal of a capacitor, it must be a connected layer or derived from a connected layer by a function that propagates connectivity.

For those layer pairs for which you provide coefficients, the program measures the capacitance providing that

- The two layers overlap.
- The two layers are not on the same electrical net.
- No shapes on other layers are between the two layers.

The following diagram illustrates how the layers and capacitance measurements are related.

The area of capacitance between *layer1* and *layer4* (labelled A) is the overlap area of these two layers minus any area of the intervening *layer2* and *layer3*. This is equivalent to

layer1 and layer4 andNot layer2 andNot layer3
If layer1 and layer4 are on the same electrical net, the capacitance labelled A is not created. The same definition applies to all other pairs of layers.

Here is an example:

```
layers( Diff Poly Metal1 Metal2 )
```

cap

Consists of one or more definitions of the electrical coefficients between pairs of layers. There is one definition for each pair of layers for which capacitance is to be measured. This argument has the format

```
cap( layerA layerB coeff1 coeff2 [coeff3 [coeff4]] [shield()] [fringe()])
```

You must specify the initial keyword cap. The entries layerA and layerB must be two of the layers defined in the previous layer list. The coeff entries define the values to be applied as coefficients to convert area and length measurements into true capacitance values. You must provide at least one of these coefficients for the definition to be valid. You can leave unwanted coefficients blank if they trail defined coefficients; otherwise, you must set them to zero or nil.

**Note:** The order of layerA and layerB in the cap argument is irrelevant. The layers are referenced in the order they appear in the layer definition. In the following definitions, layer1 refers to the first definition in the layer list (the lower layer). To avoid confusion, keep the order of layers in the cap argument the same as the order in the layers definition.

coeff1

The first coefficient coeff1 applies to the area formed between layer1 and layer2. This coefficient applies regardless of how that area is formed, as shown in the following diagram.
The second and third coefficients apply when an edge of one layer forming the capacitance area is directly over the other layer, as illustrated in the following diagrams.

**coeff2 and coeff3**

The coeff2 applies to edges of layer1 over an area of layer2. The coeff3 applies to edges of layer2 over an area of layer1. If coeff3 is left blank, the same value as coeff2 is used.

**coeff4**

The fourth coefficient applies when the edge of a capacitance area is formed by the coincidence of the edges of the two layers, as illustrated by the following diagram.

**shield**

You can use this option to adjust the total capacitance between two layers when there are intervening (or shielding) layers.
forming the edge of the capacitance area, as shown in the following figures.

The area capacitance is shown as $c_1$ in both figures. The value of the area capacitance should be less than if it was based on the area of overlap. The intervening edges of other layers reduce the value of $c_1$. You can specify a negative coefficient in the shield option to obtain this effect.

The coefficient you provide must account for the effect of the shielding layer on both the upper and lower capacitance layers. In Figure A, the coefficient must provide for the adjustment of capacitors $c_2 + c_4$ or $c_3 + c_5$.

You can provide one shield option entry for each intervening layer. The effect of multiple shielding layers forming a single shielding edge, as shown in Figure B, is cumulative.

The shield option has the following syntax:

```
shield( layer coeff )
```
The following diagram illustrates a plan view of an overlap area with the appropriate coefficients applied.

The `fringe` option has the following syntax:

```
fringe( capLayer MLlayer vertical( ...s...) lateral( ...s...))
```

You can use either or both of the `vertical` sub-options and `lateral` sub-options.

**vertical**

Lets you modify the vertical edge measurement coefficients to compensate for the existence of shapes lateral to the measured shapes.

**lateral**

Lets you generate an additional lateral capacitor to adjust the capacitor previously generated by the `measureFringe` command, to compensate for the existence of the vertical shape relationships.
The following diagram illustrates the possible relationships.

- **capLayer**: Either layer1 or layer2 of the cap argument.

- **MLlayer**: A pseudo layer generated by a prior *measureFringe* command referencing the *capLayer*. This pseudo layer passes information between the *measureFringe* and the *multiLevelParasitic* commands.

- **vertical**: Provides a user-defined calculation based on the variable s, which this tool uses to calculate the change in coefficient to be applied to the vertical edge capacitance. This coefficient is added to any other vertical component coefficients before the final capacitance value for an edge is calculated.

- **lateral**: Provides a user-defined calculation based on the variable s, which this product uses to calculate a coefficient used to generate a new capacitor between the lateral shapes. This capacitance represents a change (delta) from the original capacitance derived from the *measureFringe* command and should be added to that value by using the *calculateParasitic* command.

**Note:** The vertical adjustment due to the fringe layer is automatically added into the vertical capacitance measured by this *multiLevelParasitic* command because both
measurements are available.

However the lateral adjustment due to the fringe layer cannot be automatically added to the main fringe measurement because the main fringe values are only available in the output of the `measureFringe` command.

To manually add the lateral fringe adjustment to the main fringe measurement, add the output of this `multiLevelParasitic` command to the output of the `measureFringe` command. This is done with the `calculateParasitic` command.

A typical sequence is:

```plaintext
lateralCap = measureFringe (..........)
verticalCap = multiLevelParasitic (..........)
totalCap = calculateParasitic ( lateralCap +
           verticalCap )
saveParasitic (totalCap ..........
```

**Note:** Although the capacitance is calculated between the `multiLevelParasitic` area and the lateral `measureFringe` shape, you can still use the `grounded` option to apply the value as a capacitance to ground for both nodes.

This verification tool supplies the separation value of the lateral shapes as the variable `s`. You can provide any calculation commands. The result is a positive value unless you explicitly apply a negative sign to one of the arguments. Because lateral-vertical relationships usually reduce the vertical and lateral capacitance, the calculations can result in a negative coefficient.

A calculation is made only when an edge of a capacitance area abuts an area that previously formed a `fringe` capacitor. The value of `s` for the calculation will be the specific spacing value used for that `fringe` calculation.

You can specify a separate `fringe` argument for each of the two layers in the `cap` argument.

If both layers in the `cap` argument have lateral relationships (and associated `fringe` sub-arguments with the `lateral` statement) and both apply to a common edge of a capacitance area, both calculations are made and both resultant coefficient values are added to the normal edge coefficient.
**Note:** This only applies when the edges of \textit{layer1} and \textit{layer2} forming the capacitance area are coincident.

If the lateral spacing between shapes is equal to zero, no lateral or vertical adjustments are made. The following diagram illustrates the two scenarios where zero separation can occur.

In Figure A, there is no interaction between the zero separation fringe capacitor and the vertical edge capacitor.

In Figure B, the layer forming the zero width separation is acting as a shield layer so there is no vertical edge capacitor to be adjusted, and there is no adjustment made to the zero separation fringe capacitor.

**grounded**

Optional keyword that converts the measured parasitic value into two separate measurements between the measured nets and ground.

Without this keyword, each measurement between two nets is stored relative to those two nets. If you specify the keyword, each measurement between two nets is converted into two measurements, both having the same value as the original measurement. Each measurement is stored relative to one of the nets and ground.

If you specify the \texttt{groundNet} command, the ground net is the net specified in the \texttt{groundNet} command. If you do not specify \texttt{groundNet}, the ground net is specified as 0.

The summation of all values for any given net pair remains unaffected by this keyword. However, with the keyword, all
measurements for any net for a single measureFringe command are summed together because the second net is always ground. You can then use the calculateParasitic command to sum all measurements for each net into a combined value for the net.

limit

Optional argument that lets you reject measured values because they fall outside the specified limits. You can use it to significantly reduce the amount of data being created by rejecting all measured values that are insignificant.

The check for limits is made after the measured values have been summed for the same net numbers.

The format of the limits specification can be one of the following:

low_limit op keyword op high_limit
keyword op low_limit
keyword op high_limit

The limits can be integer or floating-point values or mathematical expressions defining such values.

You can use the following operators:

<
<=
>
>=
==

The keyword can be either ignore or keep.

You can use the following limit specifications:

ignore < .001
keep >= .3
1 < keep < 5
1.3e-4 <= ignore <= 2.7e-3

The following is an example of the cap option:

cap( metall diffusion 0.74 0.12 0.23 0.14
    shield( poly -0.05)
fringe( metal1 Fmetal vertical( -0.15 / s ) )

Assuming that metal1 and diffusion are in the same order in the layers list

- The area coefficient is 0.74.
- The coefficient for edges of metal1 over diffusion is 0.12.
- The coefficient for edges of diffusion over metal1 is 0.23.
- The coefficient for coincident edges of metal1 and diffusion is 0.14.
- Layer poly is a shield with a coefficient of -0.005 between metal1 and diffusion.
- Layer Fmetal is previously specified in a measureFringe command.
- Any metal1 edge coefficients will be adjusted by -0.15/s where s is the separation to the lateral layer defined in the measureFringe command.

Examples

The number of permutations of this command is large. The following examples illustrate some possibilities.

cap = multiLevelParasitic(
   layers( diff poly metal )
   cap( diff poly 0.35 nil )
   cap( poly metal nil 0.13 )
   ignore < 0.1 )
cap = multiLevelParasitic(
   layers( diff poly metal )
   cap( diff poly 0.35 nil )
   cap( diff metal 0.2 nil ))

Data Storage
**attachParasitic**

`attachParasitic( measurement propname device_layer [attach_layer] [shared] )`

**Description**

The *figure* option of the *measureParasitic* command enables you to measure the parasitic properties of general circuit shapes other than device recognition polygons. This command attaches these measurements to devices as properties. The association of the measurement and the device is through direct interaction between them, or indirect interaction through another layer.

This function is similar to the *saveParameter* command in that it creates properties containing measured values on device instances. In this case, however, the measurements are parasitic values derived from the *measureParasitic* command using the *figure* option, or from the *calculateParasitic* command for the same *figure* measurements rather than direct device measurements using the *measureParameter* command.

The following figure illustrates various modes of association. The device figure is D, and the measured figure is F.

![Diagram](image)

- **Direct association between a gate and its source and drain**
- **Indirect association between an npn device and collector through the tub**
- **Various associations either directly or through a reference figure**
Prerequisites

Measurements must have been made by a `measureParasitic` command with the `figure` option, or derived from them using the `calculateParasitic` command.

Fields

- `measurement` The value file reference for the measurement made with the `measureParasitic` or `calculateParasitic` commands.

- `propname` This argument defines the property name, which is created on the device to contain the measured value. It can have one of three forms.
  
  - A single property name enclosed in quotes. All separate measurements which are associated with a single device are summed into a single property value.
    
    Example: "sdcap"

  - A list of property names. Each separate measurement associated with a single device is allocated to the next unused property in the list. If there are more measurements than properties, the remaining values are summed into the first property. If there are more properties than measurements, the remaining properties are unused.
    
    Example: "source_cap" "drain_cap"

  - A list of property name and terminal name pairs in parentheses. Each measurement is associated with the property name corresponding to the terminal having the same electrical net number as the measured figure. Multiple measurements for the same terminal are summed into the same property name.
    
    Example: ( "source_cap" "S" ) ( "drain_cap" "D" )

- `device_layer` The layer reference for the device recognition polygon in the `extractMOS` or `extractDevice` command used to create the devices to which this `attachParasitic` command is to apply.

  This entry can also be a layer that is derived from a device recognition layer by a selection function such as `geomInside` or `geomGetTexted`. For each shape on this layer, the property is attached to the original device from which the shape was derived.
attach_layer

An optional layer reference or list of layer references enclosed in parentheses, defining the layer to be used to associate the measured figures with the device recognition figures. A measured figure is associated with the device figure if they both have some interaction with a common attach figure. Association can be by touching, lapping, or nesting. If a list of attach layers is defined, they are merged prior to the association.

If an attach layer is not defined, the measured figure must have a direct association with the device figure.

It is possible to have multiple associations with multiple attach figures, device figures, or measured figures. All such associations are broken down into single associations. If one measured figure is associated with multiple devices, its value is attached to each device. If a single measured figure is associated to a device through multiple attach figures, only one association is formed. If multiple measured figures are associated with a single device, each value is attached according to the property name definition of the command.

shared

This optional keyword allows the value of a single measured figure to be proportioned out according to the number of devices it attaches to. The measured value is divided by the number of devices attached to it, and the value result is stored on each device.

Without this option, the complete measured value is stored on every device to which it attaches.

Examples

Attach the source-drain area to the device as a single capacitance.

    attachParasitic( sdarea "sdcap" gate )

Attach the source-drain area to the device as two capacitors. Also share the value between all associated devices.

    attachParasitic( sdarea "scap" "dcap" gate shared )

Attach the source-drain area to the device as two capacitors correctly ordered to correspond to two device terminals. Also share the value between all associated devices.
Diva Reference
Extracting Parasitics (iLPE)

attachParasitic( sdarea ( "scap" "S" ) ( "dcap" "D" )
gate shared )

Attach a collector area as a capacitance to an npn device by association through the device tub.

attachParasitic( coll_area "coll_cap" npn tub )
saveParasitic

saveParasitic( measurement terminal1 terminal2 propname model )

Description

This function saves net measurements as parasitic device instances in the extracted cellview. The existence of a measurement between two nets causes a device to be created between those nets. The value of the measurement is stored as a property on the device.

Prerequisites

Measurements must have been made by a measureParasitic command with one of the net options, or derived from them using the calculateParasitic command.

Fields

- **measurement**: A single value file resulting from a measureParasitic command or calculateParasitic command, or a list of these layers enclosed in parentheses. If multiple value files are defined, a single parasitic device with multiple properties is created for each pair of nets.

- **terminal1**: The name of the parasitic device first terminal, in quotes. When saving values measured with the polarized option, this terminal corresponds to the first net layer referenced in the measureParasitic command.

- **terminal2**: The name of the parasitic device second terminal, in quotes. When saving values measured with the polarized option, this terminal corresponds to the second net layer referenced in the measureParasitic command.

- **propname**: The name of the property, enclosed in quotes, that is created on the device and references the value of the measurement. The value type of the property is a floating-point number. If a list of measurements is being saved, this argument must be a list of property names in parentheses. This list corresponds one-to-one with the list of measurements. The first measurement is saved on the parasitic device with the first property name, the second measurement with the second property name, and so on.
The parasitic device model name consisting of a character string enclosed in quotes. The character string must define the model name and the view name, and optionally, the library name. If the view name is omitted, the default name is `symbol`.

The defined model must be a two terminal device and be available in a library accessible to the program so that the terminal configurations can be verified and instances of the device can be placed in the extracted view of the circuit. A typical model definition might be “capacitor symbol.” If you specify a library name, the library is searched, otherwise, the library that contains the layout cellview is searched. If that search fails, all available libraries are searched in arbitrary order.

The following example shows model definitions.

- "pcap lvs mylib"
  Use model `pcap lvs` from library `mylib`.

- "pcap lvs"
  Search for model `pcap lvs` in the library that contains the layout cellview. If that search fails, all available libraries are searched in arbitrary order.

- "pcap"
  Search for `pcap` symbol in same library, as example above.

- "pcap lvs mylib"

**Examples**

The first two examples illustrate the creation of a single parasitic device per net pair with a single property.

```plaintext
saveParasitic( cap "PLUS" "MINUS" "c" "pcapacitor" )
saveParasitic( area "ANODE" "CATHODE" "a" "pdiode" )
```

The following examples each produce a single parasitic device per net pair, but with multiple properties.

```plaintext
saveParasitic( ( length width ) "PLUS" "MINUS"
  ( "W" "L" ) "pcapacitor" )
```
saveParasitic( ( sourcec drainc ) "PLUS" "MINUS"
( "sc" "dc" ) "pcapacitor" )

ConclCe Command

Verify ⇒ ConcICe

Reduces your large RC network to a smaller, electrically equivalent network, the Assura™
RC network reducer option (ConclCe) view. You can use this view with any tool that can take
an extracted view, such as interconnect analysis tools. For more information see the Assura
ConclCe Help.

Prerequisites

You must have an extracted view of your layout created by this tool.

ConclCe in Diva ignores parasitic capacitance measured by measureResistance. Use
measureFringe and multiLevelParasitic to measure parasitic capacitance.

The ConclCe command does not support parasitic capacitances with multiple value
properties on them. Use a separate saveParasitic command for each separate capacitance
instance.

The multiLevelParasitic command is usually used to make adjustments to values generated
by measureFringe, which can create negative capacitors. To prevent this problem, combine
all measurements from measureFringe and multiLevelParasitic in a calcParasitic statement
and save the results with saveParasitic.
ConcICe in Diva Form

ConcICe View Name sets the name of the extracted view. If you iterate reductions using different accuracy levels and coupling factors, you can apply a different view name to each iteration.

Accuracy is an integer representing the number of capacitors left between the two end points of a wire segment after reduction. The default is 2. An end point can be an instance pin, a pin on a pad, or a branch node. The number of capacitors does not include the junction capacitors on the two end points. The number of resistors left on the segment will generally be one more than the number of capacitors. For details, see the “About Reduction Accuracy” section in the Assura ConcICe Help.

Coupling Factor is a floating-point number between 0 and 2 that is used as a multiplier to model the effects of coupling capacitance, even though it is lumped to ground in the reduced circuit. You set one factor that applies to all the coupling capacitance in the circuit you are reducing. The default is 0. For details, see the section, “How Coupled Capacitance is Lumped to Ground” in the Assura ConcICe Help.

Resistor Model Data displays the library, model type, view type, and property name (r for resistor) of parasitic resistors for which measureResistance has stored parasitic resistance values. Models listed here will be reduced. You can delete a resistor type if you do not want that type reduced, or add a resistor type that you do want reduced.

Capacitor Model Data displays the library, model type, view type, and property name (c for capacitor) of parasitic capacitors for which saveParasitic has stored parasitic capacitance values. Models listed here will be reduced. You can delete a capacitor type if you do not want that type reduced, or add a capacitor type that you do want reduced.
Using ConclCe

To run ConclCe on your extracted view, see the “Running ConclCe in Diva” section of the Assura ConclCe Help.

Viewing Reduction Results

When you graphically display the ConclCe view of your network, it looks approximately the same as the extracted view. During probing, you might see some slight differences due to the reduced number of resistors. For example, the resistance of the interconnect shown in the following figure was distributed to five sections. Each section represented a resistor. When the interconnect is reduced to three resistors, two of the connectors disappear.

Extracted view: interconnect distributed to five resistors

ConclCe view: interconnect reduced to three resistors

These connections disappear after reduction
Extracting Parasitic Resistance (iPRE)

Introduction

What Is Parasitic Resistance Extraction?

The Parasitic Resistance Extraction (PRE) function lets you convert one or more interconnect layers to a resistance-capacitance network. You do not have to perform any preprocessing to prepare the interconnect layers for extraction, as would be the case if you were extracting devices using the `extractDevice` and `extractMOS` commands.

Resistance extraction provides you with a reasonably accurate representation of the interconnect parasitics without incurring excessive processing overhead. Accuracy is obtained by a thorough topological analysis and by inclusion of factors such as bend resistance, transition resistance, and contact resistance, and through the handling of complex shapes, all angles, and multiple terminations to contacts and device terminals.

Distributed capacitance of the interconnect can be included in the extraction with your own choice of the resistance-capacitance (RC) model as well as the various coefficients and factors.

How Do I Invoke It?

Resistance extraction is triggered through the `measureResistance` command, which must precede the `geomConnect` command in the input stream. Resistance extraction is performed as part of a normal layout extraction run in which the circuit network is extracted from the layout and stored in an extracted view of the circuit.

What Are the Results?

During circuit extraction processing without parasitic resistance, an *extracted* view of the circuit is produced in which every electrical net formed by the layout is represented as a single entity in the data.
Parasitic resistance works on one connected layer at a time. Any part of an electrical net that is composed of the specified layer is broken down into a resistor-capacitor network. The parts of the net that are not composed of resistance layers are left as before.

The result is that each single entity that is an original net becomes a network. The original net no longer exists. If the original net was labelled, the part of the original net that overlaps the label uses that label for the net name. The additional nets created have names based on the original net name and a sequence number. You can control the exact format of the name used for the additional nets by defining the SKILL function `ivUserNetNameFormatter`. If the SKILL function is not defined, the Diva verification tool uses a default format of `<original-name>:<sequence-number>`. An example of the Skill function that mimics the default behavior is:

```plaintext
procedure( ivUserNetNameFormatter( name number )
    get_string( concat( name ":" number ) )
); end iVUserNetNameFormatter
```

The original devices which were recognized by the `extractDevice` and `extractMOS` commands are unchanged. New devices representing the resistors and capacitors are added.

The layers used for resistance extraction are removed from the circuit to form the resistors themselves. Therefore, they are no longer available for other parasitic measurements, such as cross-coupling capacitance.
Resistance Extraction Capabilities

The Extraction Process

Several separate functions are performed during parasitic resistance extraction. Processing is split into two sections.

- The first section is performed before the circuit interconnect extraction and is the preparation of the data. It is performed for each layer defined to be used for resistance extraction.

- The second section is performed after the interconnect extraction and is the actual recognition and measurement of resistors and capacitors.
Preconnect

The resistance layer is split into two separate layers. The first layer is a replacement for the original interconnect layer and consists of the areas formed by the contacts and pins plus special areas formed from device recognition shapes see the “Device Terminals” on page 427. This layer forms the terminals of the resistors. The second layer is all of the original layer minus the terminal layer and forms the shapes to be converted into resistors.

To enable the capability of combining resistance extraction and cross-coupling or fringe capacitance extraction, you can also specify that a special additional layer be derived that consists of a concatenation of the terminal and resistor layer. This layer appears to be almost identical to the original layer but has the correct nodal information to allow parasitic capacitance to be measured. You can use this special layer as input to the parasitic measurement commands.

You can also request that the resistor bodies be fractured into smaller pieces to allow more accurate distribution of the cross-coupling capacitance inside the resistor network.

Postconnect

The resistor layer is processed to extract the resistance and capacitance network it contains. Each contact or device terminal that was cut out to form the resistance layer acts as a terminal to the extracted network. The devices and nets forming the extracted network are added to the extracted view of the circuit, supplementing the devices and interconnect already extracted through device and connectivity extraction steps.

The extraction process consists of breaking the resistor layer down into small pieces. Each piece can have its resistance and capacitance calculated with compensation for bends and transitions. These resistances and capacitances form an initial network into which contact resistance is added. The network output is then reduced to its simplest form.

Each resulting device in the network is an entry that has a resistance value and a capacitance value. You can use the system netlisting capabilities to translate these devices into networks of their own, such as Pi or Tee R-C networks.

Resistor Formation

Forming resistors from polygon shapes requires several steps. The individual polygons on the resistor layer are fractured into pieces. Each piece has a maximum of four sides and no external angles less than 180 degrees (the piece has no reentrant angles). Each piece has its edges identified as external edges of the original shape, as connections to other polygon
pieces, or as connections to contacts and device terminals. This concept is illustrated for a simple polygon. Complex polygons follow the same process.

![Diagram of polygon pieces with resistors](image)

Each polygon piece has possible current flow between all contact edges, terminal edges, and connections to other pieces. For each possible current flow path, the shape is analyzed and the resistance of that path is calculated. All the resistance paths are then combined to form a network. There can be from zero to four resistors in each polygon piece. As far as current flow is concerned, the pieces with zero resistors are dead ends. The following figure illustrates examples of resistance networks inside polygon pieces.

![Examples of resistance networks inside polygon pieces](image)

The determination of resistance for each path in a polygon piece includes compensation for bends, steps, and transitions. Even though each individual piece does not have these configurations, the information from the original polygon, necessary for the calculations, can be deduced from the pieces.

During the fracturing process, the external edge length and area of each polygon piece is measured and associated with the piece. If you supply a calculation as part of the resistance extraction command, the capacitance of each polygon piece is calculated and distributed among the resistors calculated from the piece.
**Contact Resistance**

The program considers three components of contact resistance during the resistance network extraction.

- The resistance relative to current flow around the contact
- The resistance of contact edges
- The resistance of contact area

**Current Flow**

You conventionally apply a compensating factor to the resistance along a net based on the width of the contact to that net and the width of the net itself. This approximation is intended to compensate for the change in resistance created by current flow around the contact.

This program’s resistance extraction methodology makes contact compensation unnecessary since it extracts the complete network around the contact.

The following figure illustrates an example of a resistance network around a single contact. This form of network extraction is applied in all contact configurations, whether there be single, multiple, or arrays of contacts.

![Resistence Network Example](image)

**Edge Resistance**

Contact edge resistance is considered when current is flowing across the edge from the net to the contact or vice-versa.

You conventionally consider only the edge of the contact that is perpendicular to the main direction of the net to which it connects. This is an approximation because you have no information as to the current flow around the contact.
Since the program does have the current flow information around the contact in the form of the extracted network, it can utilize the resistance of all edges of the contact. The program therefore measures the length of each contact edge and applies to it the coefficient you supplied for contact edge resistance with the assumption that the resistance value of an edge is inversely proportional to its length.

You can define a different edge resistance coefficient for each type of contact.

The following figure illustrates the edge resistance of a contact in isolation from other resistances.

![Edge Resistance Diagram]

**Area Resistance**

Resistance of the contact area is considered as a resistance to the flow of current through the contact. The coefficient you supply is used to calculate the resistance inversely proportional to the contact area.

You can define a different area resistance coefficient for each different type of contact.

The following figure illustrates area resistance of a contact in isolation from other resistances.

![Area Resistance Diagram]

Since the contact edge and area resistance extraction capabilities are both optional, the network output can be formed in one of four ways, as illustrated in this figure.
In all cases, the resistors on the side of the contact away from the current flow direction have little effect on the total resistance, but they do accurately model the true circuit.

**Possible Contact Configurations**

- All
- No area
- No edge
- No edge or area

**Device Terminals**

Connections to resistors are formed from contacts, pins, and device terminals formed from the device recognition shapes. For pin and contact connections to resistors, the Diva verification product uses the complete area of the connection shape as the connection to the
resistor. For device terminals, this tool has to create special connection shapes using the device recognition shape as the base.

There are two types of device terminals. A resistor shape can butt against a device recognition shape to form a terminal as in the source/drain of an MOS transistor. A resistor shape can also pass across a device recognition shape to form a terminal as in the gate of an MOS transistor.

To form connectivity for the device terminal recognition, each device terminal has to be represented by a shape on the interconnect layer. If this verification tool converted all the interconnect layer to resistor layer there would be no terminals left for connectivity, so this tool creates its own terminals.

For the source/drain type of connection, this tool creates a terminal at each butting interface between the resistor layer and the device recognition shape. This terminal is a narrow sliver along the butting edges. Its existence lets this tool recognize both the resistor and device terminals.

For the gate connection, this tool creates a very narrow sliver across the device between the source and drain edges. This sliver has area, so it behaves as a normal terminal to the device during device extraction. However it is small enough that it has virtually no effect on the resistance along the path through the gate.
For both these connection types, the final interconnect layer has no shape that represents the full device recognition shape. This means you cannot use the original resistor layer to make device parameter measurements. For example, if you convert gate polysilicon into resistors you cannot measure gate width by measuring the length of polysilicon coincident with the gate region. You must measure the length of the gate region butting the source/drain layer.

If there is a contact or pin over the gate region, and the area of that contact or pin does not intersect with the gate sliver terminal, the gate has two connections and is flagged as badly formed.

**Bend Resistance**

You can provide a bend resistance factor to the program so that it can more accurately calculate the effect of bends in a path on the resistance of that path. You provide a factor that is defined for right-angled bends where the path width does not change from one side of the bend to the other. For bends in which the paths on either side of the bend are of different width, the factor is modified to cover the angle of the bend plus the transition effect from one path width to another.

If you want a 90-degree bend to be considered as if the current always flows through the center line, you would have to provide a factor of 1. The default factor applied by the program if you do not provide one is 0.56. For bends other than 90 degrees, the program uses a portion of the factor you supply based on the angle between the paths on either side of the bend. The smaller the angle, the smaller the factor used.

The following figure illustrates sample bend configurations for which the program compensates.
Transition Resistance

A transition is a sudden change in the width of a conducting path. Generally, the term transition is used when the one or both sides of a path form a step. In the examples, the discussion is limited to a step on a single side, but the principle applies in both cases.

The transition effect on current flow is to form a dead corner through which no current actually passes. Since all the resistive material of the path is not in use, the resistance formed by the path is higher than would be the case if all the material were used.

The figure shows the current flow pattern and examples of how the resistance could be calculated.

The simple solution does not consider the transition, and results in a calculated resistance value higher than the real value. The alternate solutions show how the program methodology of cutting the original shape ensures that the calculated resistance is as close to the real resistance as possible. The appropriate shape cutting is automatic, requiring no user information or intervention.

Network Reduction

The resistance extraction process described so far results in a large number of individual resistors connected in a complex network. Even a simple “dog-bone” resistor can result in
over 36 resistors representing a single resistance path. Although this is an accurate representation of the resistive circuit in the original shapes, it would consume an unacceptable amount of storage space and be time-consuming to simulate.

To overcome this, the program reduces the network to a minimum configuration. A dog-bone shape should reduce to a single resistor. The reduction process uses a series of transformations in an iterative sequence. Each sequence reduces the number of resistors. The program repeats the process until no more reduction takes place.

The reduction process does not reduce the accuracy of the network. The transformations used give exact equivalent circuits. No approximations are made.

In most cases, the reduced network includes the minimum number of resistors required to represent the circuit. For some circuits, the program might produce a less-than-minimum result when the network reaches a configuration that cannot be reduced further by the tools available.

The resistor configuration produced by the program might not match your concept of what that configuration should be. The result, however, is meaningful and accurate.

You can also eliminate resistors from the network after the reduction process completes by using the ignore option. The ignore option replaces resistors below the value you define with short circuits and replaces resistors above the value you define with open circuits.

**Cross Coupling and Fringe Capacitance**

**Measurement**

You can specify an output layer for the `measureResistance` command. This layer is the original layer you defined for resistance but contains separate polygons for the resistor bodies and contacts, pins, and device recognition polygons that form the terminals of the resistors. It is a composite of the resistor layer and the remaining interconnect layer.

You treat the output layer like any other derived layer containing unmerged data. For most situations, this verification tool treats the layer as unconnected. However, you can use the layer to measure parasitic capacitance with the following commands:

- `measureFringe`
- `multiLevelParasitic`

In these cases, the language parser treats the resistance output layer as if it is connected. You use the layer in these commands just like any other connected layer.
The output layer from `measureResistance` must not be used with the `complexParasitic` command. This new command requires the input layer of `measureResistance` to be used. The `complexParasitic` function will automatically locate any `measureResistance` command which processes the layer, and cause `measureResistance` to produce the data needed by `complexParasitic`.

Any parasitic measurements made between the resistance output layer and any other layers are evenly distributed to the terminal connections of the resistors. For example, in the following figure:

- The resistor connected between terminals A and B.
- The resistor crosses net C with a cross coupling capacitance of 10.
- Two capacitors are formed:
  
  A to C = 5  
  B to C = 5

Any capacitance to nets directly connected to the resistor body are considered as capacitors to the same net and are ignored.

Measurements can be made between two or more resistor bodies, and the resultant values are distributed between all the resultant terminals. For example, in the following illustration:

- Resistor 1 (R1) is between nets A and B.
- Resistor 2 (R2) is between nets C and D.
- The value of 20 is measured by R1 crossing R2.
- These capacitors are formed:
A to C = 5
A to D = 5
B to C = 5
B to D = 5

You can manipulate parasitic measurements made from a resistor body like any other measurement by using the `calculateParasitic` and `saveParasitic` commands.

When using `measureFringe` on resistor bodies, it is recommended that you always use the `shielded` and `opposite` options in the `drc` command. This is recommended whenever you use `measureFringe`, but is even more important when you use the command on resistor bodies because the layer you are measuring consists of the concatenation of the resistor bodies and the resistor terminals. This concatenation results in *internal* edges at the interface between the two layers. If your resistance layer originally terminated at a gate (like the source/drain of an MOS device), this butting edge is included in the parasitic measurements and might cause false capacitance values.

If any resistor shape has no terminals, it is not included in the parasitic measurements because there are no terminal nodes to which the resultant capacitors can attach.
Distribution

For resistors or networks derived from larger polygons, the even distribution of capacitance on the original resistor terminals might be inaccurate. To improve the accuracy, you can fracture the original resistor bodies into smaller pieces. You can provide a command line option to control the size of these pieces.

This verification tool fractures resistor bodies for better distribution of resistance and capacitance by creating extra terminals along the path of the resistor. These extremely narrow terminals cut across the resistance path and have little impact on resistance and capacitance calculations. When you measure cross-coupling capacitance on resistors where fracturing has been applied, separate measurements are made for each fractured resistor piece and are distributed only to the terminals of that piece. Consider the following example:

A path has been cut into four pieces and has a cross coupling-capacitance across the second piece. The original terminals of the shape are A and B. The extra terminals created by the distributed cutting are D1, D2, and D3.

The resultant network shows the cross-coupling capacitance value distributed only to the terminals of the second piece.

The fracturing applies to all resistor body shapes, but the actual cuts are made only in parallel-edged horizontal and vertical sections of the resistor. The extra terminals are narrow and have a minor effect on the resistance and capacitance values. For example, a resistance of 10 might become 9.99.

This verification tool has been designed to provide consistent results when fracturing shapes that you place at different orientations. However, it is possible that in some circumstances rotated shapes are cut differently.
This product maintains the length of the fractured segments in a straight, parallel-edged path, close to the distribution factor you provide. However, this product does not attempt to maintain the exact length of segments that form steps, bends, or branches.

**Resistor Body Capacitance**

As described under *Resistor Formation*, during the creation of the resistor pieces from the original shape, the area and external edge length of the pieces is used to calculate a capacitance value for those pieces. The area is the complete area of the piece, and the edge length is the length of any edges of the piece which were also edges (or partial edges) from the original shape. These values are used in the calculation you supplied in the original *measureResistance* command to generate a capacitance value for the piece.

When the resistor network is calculated for each piece, that capacitance value is distributed among those resistors. During the resistor network reduction phase of this verification, the capacitance values are consolidated along with the resistors. The result is a single value of capacitance associated with each resistor in the reduced network. As with the resistance values, no accuracy is lost during the reduction process.

**R-C Models**

Although this description of the resistance extraction process has, up to now, used the word *resistor* to describe the devices extracted by the program, this is not exactly true. What has been extracted is two terminal devices of undefined type that have both a resistance value and capacitance value. It is up to you to give these devices a model name and to provide a device library element for them.

For the purposes of netlisting these devices, you can either set up the netlist control to have each of your own device types generate a netlist entry directly, or you can have each device be represented by a schematic containing a subnetwork that forms a netlist down to individual
resistors and capacitors. Within this subnetwork, you can choose the required resistor and capacitor network formation.

Netlisting R-C Models

The major difference between normal netlisting R-C models and normal netlist control is that the properties containing the values of resistance and capacitance are not on the instances of the resistors and capacitors, but on the instance of the higher level cell.

Each lower level device in the schematic can have associated with it only a portion of the value assigned to that property on the instance of the cell. For example, if the property value on the cell for capacitance is 20, and the schematic cellview of it contains two capacitors, each capacitor needs to have a value of 10 generated for it in the netlist.

To facilitate netlisting for this situation, the following procedure must be adopted.

- Create a cell called *pirescap* for the device to be extracted from the resistance program.
- This cell needs two cellviews, one called *symbol* containing a picture of the circuit used to display the device in the extracted cellview, and one called *schematic* containing interconnected instances of resistors and capacitors, which is used by the netlister. The resistance extraction program creates instances of this cell and adds two properties to each. For example
  
  \[
  r = 12.65 \\
  c = 134.9
  \]

- The *pirescap* schematic cellview contains a circuit consisting of a single resistor with a capacitor to ground at either end. The single resistor is an instance of a device called *pires* and the two capacitors are instances of devices called *picap*. Both devices have cellviews called “lvs” with similar properties. The capacitor is used in this example.
The first two entries have the type of NlpExpr and are the same as those on a normal capacitor. The third entry defining the property has the type of ilExpr. The Postamble entry references the capacitor element entry which is in the nlpglobals cellview. The capacitor element entry references the property entry. Normally, the property entry is found in nlpglobals, but in this case, the one shown in the capacitor itself takes precedence. The property entry references a SKILL function that must be provided by the user.

```
procedure( resPrintCBy2( )
    prog(( str tmp )
        tmp = fnlSearchPropString( "c" nil )
        if (tmp, then
            tmp = evalstring( tmp ) / 2.0
        else
            tmp = 0.0
        )
        sprintf( str "\ c %f " tmp )
    return( str )
)
```

The fnlSearchPropString function searches through the hierarchy leading to the device instance, looking for a property called c. In this case, it finds it on the parent instance of the capacitor, namely, the instance of pirescap. The value is returned as a string and is converted to a number by the evalstring function and divided by 2.

The resultant value is then printed into a string along with the remaining text required by the final property (in this case, the enclosing quotes and the c character) and returned, which has the effect of printing it in the netlist. In the example, a similar routine is needed for the resistance, but without dividing its value by 2.

The resistance extraction program is working with devices that have two terminals. The only additional nets that can be introduced into the schematic are global signals such as power and ground.
Multiple Layer Extraction

Your run command stream can have more than one resistance extraction command. Each command applies to a different interconnection layer, and there are no interactions between the devices created from one command and the devices created from another.

Since the extraction and reduction process works on a shape-by-shape basis, the terminations of each shape (the contacts and device terminals) remain the terminations of the network generated for that shape. For example, a single interconnection line formed from two layers joined by a single contact causes two resistors to be created, one for each layer, with a junction net at the contact between them. The circuit reduction does not proceed across that contact.

Unexpected Results

There is one situation that causes unexpected results if you are not careful. As explained, if you request extraction of two or more layers, and any two of them are directly connected by a contact, the network reduction is broken at that point. The manner in which the contact is cut from the resistor prior to resistance extraction creates a new electrical net at the contact area.

If the connection between the two layers is made at any point by multiple contacts, each contact area becomes a separate electrical net. Since they are independent nets, resistance extraction causes discrete resistors to be created between them. This can be seen in the
The figure shows how a single contact between two resistance layers causes a break in the network, and how two contacts result in a more complex network.

As the number of individual contacts forming a single connection between two resistance layers increases, so does the complexity of the network. The lowest number of resistors between three contacts is three on each layer, totalling six. The lowest number of resistors between five contacts is ten on each layer, totalling twenty. As the numbers of resistors increases, the problems of subsequently processing those devices increases.

You reach a point where you won’t want such a network formed. There are techniques you can use to overcome the problem. The results are not quite as accurate as having the full network, but the degradation is minimal.

The paragraphs detail two techniques to handle the problem. Both rely on the same principle but approach it in different ways. Each has its advantages and disadvantages. Both cause all the contacts at a single connection to be connected to a single net, allowing the network reduction process to fully function.
A Simple Approach

The simplest way to make all contacts at a single point have a common net is to generate a new layer from the intersection of the two resistance layers and include it in the `connect` command.

```
short = geomAnd( metal1 metal2 )
MeasureResistance( metal1 "resistor" 1.0 "r" )
MeasureResistance( metal2 "resistor" 1.0 "r" )
geomConnect ( via( via short metal1 metal2) )
```

In this example, a layer called `short` is created as the `and` of `metal1` and `metal2`. During the preparation for resistance extraction, the `metal1` and `metal2` layers is split into the areas wanted for resistance, and the areas under the contacts. During the connection process, if there are no contacts over a shape on the `short` layer, no connections are made. Where there are contacts over a `short` shape, the remaining contact areas of `metal1` and `metal2` connect as normal, and each area connects to the same `short` shape, forming a single electrical net.

This figure shows two examples of multiple contacts. The first figure illustrates a case where the approach gives a reasonable result. The second figure, however, illustrates a case where the result becomes significantly inaccurate. There is a large resistance between the contacts which disappears if the contact areas are on the same electrical net.

A Better Approach to get More Accurate Results

This approach requires more user-supplied functions and absorbs more processing time, but provides more accurate results.

```
close = drc( via sep < 3 opposite )
short = geomOr( via close )
MeasureResistance( metal1 "resistor" 1.0 "r" )
MeasureResistance( metal2 "resistor" 1.0 "r" )
geomConnect( via( via short metal1 metal2) )
```

The `drc` command creates shapes from the areas between close contacts. The value used for the separation depends upon your own understanding of the technology and definition of
when contacts can be considered common and when they cannot. The shape results are then combined with the contacts themselves to produce the same short layer as in the previous example, which is then processed the same way in the connect command. In the previous figure, the first figure shows contacts shorted together (providing they are closer than the drc separation value). The second figure shows contacts remaining separated with the network resistor result between them.

**Using Shorts to Reduce Large Contact Array Resistor Networks**

You must grow and shrink the contacts by an amount sufficient to merge contact arrays and then use the resultant layer as the "short" layer. This has the effect of giving all contacts in the array a common node, but it does not bypass the measurement of the network around the node or the contact resistance measurement. The program will still need time to create the network inside the contact array, but the time to reduce the network down is significantly reduced and the resultant network is much simpler.

This provides an intermediate solution between leaving the contacts as they are (which is expensive in run time and creates a very large resistor network) and growing and shrinking the contacts themselves which eliminates the contact resistance accuracy.

**Command Reference**

The following section discusses the measureResistance command.
measureResistance

[outLayer=] measureResistance( reslayer model rescoeff [bf] resprop [cap] [ignore] [save] [contact][distribute][capLimit])

Description

The measureResistance function extracts a resistance-capacitance network from an interconnect layer. The extraction process allows for sheet resistance, bend resistance, transition resistance, terminal resistance, contact area resistance, contact edge resistance, area capacitance, edge capacitance, and user defined R-C models.

Prerequisites

The layer to be processed for resistance extraction must be referenced in a geomConnect command. The measureResistance command must precede the geomConnect command.

Fields

outLayer

An optional output layer consisting of the original resLayer separated out into the resistor bodies and the resistor terminals (cuts, pins, gates, and device recognition shapes).

If you specify this option, you can treat the outLayer as any other unconnected derived layer, except that it is unmerged. For the purposes of parasitic extraction, you can treat it as a connected layer, except for the complexParasitic command, which must be given the resLayer.

The following parasitic measurement commands accept the outLayer as a connected layer:

- measureFringe
- multiLevelParasitic

Any parasitic capacitance created with a shape on the resistor output layer is distributed evenly between the terminal nets of the resistor network extracted from the shape.

If you do not specify this argument, no output layer is created.
The layer to be processed for resistance extraction. This must be a derived layer used in a `geomConnect` command. The layer is used for both input and output. On input, the layer is the originally derived interconnect layer. The resistance extraction process removes those areas of the layer which becomes resistors, leaving only those areas which form contacts and device terminals. It is these remaining areas which compose the output layer.

The area of the input layer that forms the resistor bodies can be saved separately, if required for reference, using the `save` option of the `measureResistance` command.

The device model name consisting of a character string enclosed in quotes. The model can represent a low-level device such as a resistor or a higher level device that netlists down into a resistor and capacitor network.

The character string must define the model name and the view name. If the view name is omitted, the name `symbol` is defaulted. The defined model must be available in a library accessible to the program so that the terminal configurations can be verified and instances of the device can be placed in the extracted view of the circuit.

The defined model must be a two-terminal device in which the terminals are interchangeable (nonpolarized). Because of this, the terminal names are not required to be specified in this model definition since they can be deduced directly from the model.

"pi-res-cap symbol"

A floating point or integer value representing the resistivity of the interconnect layer being processed, in ohms per square.

An optional bend factor for adjusting the values of resistors having bends. The value is the resistance in squares around a single 90-degree bend, replacing the theoretical measured value of 1.

The bend factor is automatically adjusted for bends having angles other than 90 degrees and for bends whose legs are of different width. The bend factor is also applied to step transitions.
between one width and another in straight resistors to improve the accuracy of resistance calculation.

If you do not provide a bend factor, the program defaults to a 0.56 value.

resprop

The name of the property to be attached to the device to contain the value of its resistance. This must be a text string enclosed in quotes.

cap

An optional argument to specify the parameters for capacitance measurement. If not defined, no capacitance extraction is performed. The options are

( cap capprop (equation) )

cap

This is a keyword introducing the capacitance definition.

capprop

The name of the property to be attached to the device to contain the value of its capacitance. This must be a text string enclosed in quotes.

"c"

equation

This equation converts the area and perimeter of the capacitor into a single capacitance value. It uses the symbols a (area) and p (perimeter). No other symbol is allowed in the equation. The full range of mathematical functions used in the calculateParasitic and measureFringe commands are available.

Since the capacitance calculation is performed on resistor pieces, and the perimeter measured is from the original shape’s external edge perimeter, it’s possible that some resistor pieces have no original edge perimeter, resulting in a p value of zero. Do not perform a division using the p symbol.

Here is an example of the cap option.

( cap "c" 0.35 * a + 1.02 * p )

ignore

This optional argument lets you remove resistors from the circuit if their values fall outside the limits you specify.
This argument has the following syntax:

( [value op] keyword op value )

value
Any positive integer or floating point value.

op
One of the following operators:

<
less than

<=
less than or equal to

>
greater than

>=
greater than or equal to

keyword
You can define two keywords:

keep
Keeps any resistor that meets the limits you specify.

ignore
Removes any resistor that meets the limits you specify.

For example, the following syntax removes all resistors whose value is less than 1.

( ignore < 1 )

The following example removes all resistors whose value is greater than 1,000.

( ignore > 1000 )

The following example removes all resistors whose value is less than 1 and more than 1,000.

( 1 < keep < 1000 )

Any resistor that is removed because its value is lower than the limit you specify is replaced by a short circuit. Any resistor that is
removed because its value is greater than the limit you specify is replaced by an open circuit.

You cannot specify a range of limits using the ignore keyword. For example, the following syntax is invalid:

\[( 1 < \text{ignore} < 1000 \)\]

Any capacitance associated with the removed resistors is distributed around the resistors on the resultant nets.

**Note:** For versions prior to 4.4.3, there were limitations when using the keep or ignore options. A resistor was never removed, regardless of its value, if one of its nets was connected to a contact, device terminal, or pin on the original net. This was true even if the value of the resistor would otherwise cause it to be ignored.

In the 4.4.3 release and all subsequent releases, resistors can now be removed based on their values regardless of their connectivity.

The program automatically ignores very small resistors during network reduction, whether you specify the ignore option or not. The program applies the ignore option after the network reduction.

**save**

This optional argument allows you to save the shapes representing the resistor bodies in the extracted cellview.

\[( \text{save layerName fracture} \)\]

**save**

This is a keyword introducing the save layer definition.

**layerName**

The name of the graphics layer, in quotes, on which the resistor body shapes are to be stored.

**fracture**

This is a keyword that creates fracturing lines, which lets you see how the geometry is cut up before analysis.

This is an example of a save option.

\[( \text{save "poly_res" fracture} \)\]

The shapes saved on this layer are for display purposes only. They are not associated with the electrical network in any way.
They are a subset of the original connectivity layer used to create the resistors. The remaining shapes from the connectivity layer remain on that layer and can be viewed by saving the connectivity layer in the normal way.

**contact**

This optional argument lets you specify parameters for contact resistance measurement. Without this argument, no contact resistance is measured. Any number of these arguments can be specified for different contact and terminal layers. The same process is applied to contacts and device terminals.

( contact layer acoeff lcoeff )

**contact**

Keyword introducing the contact resistance definition.

**layer**

The layer reference for which the contact resistance is to be measured. The layer must appear in a `geomConnect` command.

**acoeff**

A floating point or integer value that defines the coefficient to be applied to the contact to convert its area into a single resistance value in the resultant network. The resultant value becomes inversely proportional to the area. If this coefficient is not required, its value must be set to 0.0.

**lcoeff**

A floating point or integer value that defines the coefficient to be applied to the contact to convert its edge lengths into individual resistors in the resultant network. Each edge forms a resistor between the contact and the surrounding interconnect, whose value is inversely proportional to the edge length. If this coefficient is not required, its value must be set to 0.0.

( contact cut 0.03 2.5 )

**capLimit**

This optional argument lets you limit the number of terminals used when distributing parasitic capacitance values created between a resistor body and another shape.

( capLimit [N] )

The required value N is the maximum number of resistor network terminals used when distributing parasitic capacitance. The
terminals used are selected by the software on a first-come, first-used basis.

The primary purpose of this option is to reduce parasitic capacitance runtimes and disk usage. Improvements in data handling have greatly reduced the impact of large numbers of resistor network terminals. We no longer recommend the use of this option.

distribute

This optional argument lets you fracture the resistor bodies into smaller pieces to obtain more accurate resistance and capacitance distribution. The distribute argument has the following form:

( distribute [N] )

The distribute keyword tells this tool to fracture the resistor shapes into smaller pieces.

The optional value $N$ is any integer greater than 1. The value is directly related to the size of the fractured pieces that result. If you specify the distribute keyword but do not specify a number, a default value of 12 is used. If you do not specify a keyword, the resistor bodies are not fractured.

The value of $N$ is allowed to be zero for the special case of disabling the distribute option.

The value $N$ defines the ideal length in squares that each fractured piece should be if a single straight path is fractured. For irregular shapes and branched shapes, there may not be any unbroken paths of this length, but this product still fractures the shape. The larger the number, the larger the pieces are.

Fracturing introduces new terminals in the circuit. These extremely narrow terminals span the resistor bodies (hence cutting them into pieces) and have virtually no impact on the resultant resistance and capacitance of the pieces. These terminals are positioned only in horizontal or vertical sections of the resistor body.

**Note:** For versions prior to 4.4.3, there were limitations when using the distribute option in conjunction with the keep or ignore options. The splitting of the resistor bodies into two or more pieces using the distribute option creates new terminals. For
purposes of resistor removal, these new terminals are treated as if they were part of the original net. The result is that pieces created by the `distribute` option cannot be removed, even if the value of the resistor would otherwise cause it to be ignored.

This implies that there can be situations where the entire resistor in a branch is removed when the `distribute` option is *not* used, but *none* of the pieces are removed when the `distribute` option is used.

This above problem has been fixed in the 4.4.3 release. All resistor pieces created by the `distribute` option can now be removed based on their values using the `keep` or `ignore` options.

**Examples**

Since the number of permutations of this command is large, these examples illustrate one minimal command and one using all options.

```plaintext
measureResistance( poly "resistor" 30.5 "r" )
measureResistance( metal "res_pi_cap" 30.5 0.5 "r" )
( cap "c" (1.5e-12 * a + 2.7e-13 * p ) )
( ignore <= 0.15 )
( save "resist_body" )
  ( contact cut 0.1 2.5 )
  ( contact via 0.08 1.4 )
  ( distribute 20 )
)```

```plaintext
```
Checking Electrical Rules (iERC)

Introduction

The Electrical Rules Check (ERC) program verifies the structure and integrity of a network without doing a full Layout Versus Schematic (LVS) comparison. Because its primary processing is based on the circuit netlist, you can use it on extracted views or schematics.

ERC does not provide the accuracy that the LVS does, but it does provide valuable information that helps you to remove many errors from your circuit prior to LVS checking.

ERC works as follows:

- Generates a netlist for the circuit
- Verifies the circuit using the verification rules in your ERC rules file and the run options you specify in the ERC form when you select the ERC command from the Verify menu
- Verifies device parameters using an internal feature that combines parameters during series and parallel reduction
- Outputs error files that can be used to view and probe errors
- Displays and explains the errors it found

The ERC program is controlled by the system simulation environment (se). The simulation environment is normally set up by your system administrator. See Appendix D, “Simulation and Environment Control” for more information.

You can run ERC using the ERC command from the Verify menu or using the ivERC SKILL function.

You control ERC processing by entering verification rules in the ERC rules file and by choosing options in the ERC form. The verification rules commands define the type of verification checks you want to run. The form options define requirements for a specific run.
Prerequisites

The prerequisites for ERC are

- You must define ERC verification rules in a text file.
- You must extract a network so that ERC can generate the netlist it needs.
  
  To create the layout network, run an extraction on the layout using the *Extract* command from the Verify menu to get an extracted view.

  To create the schematic network, run a check on the schematic using *Edit – Check – Current Cellview*.

- You must initiate ERC from a window containing either a *schematic* cellview or an extracted layout cellview.

Program Functionality

These sections describe the verification rules commands available in ERC and their overall function.

Defining External Connections

The following commands define the circuit connections. Many other commands use this information. In some cases, the circuit connection definition is essential to the functioning of the command and in other cases, it eliminates spurious errors. For example, the pull-up and pull-down commands cannot function without power and ground definition. The commands can produce spurious errors if inputs are not defined. The commands are as follows:

```plaintext
setPower
setGround
setInput
setOutput
```

Tracing Paths

The following commands define the flow characteristics of devices. The pull-up and pull-down commands use these characteristics to determine how the tracing of the circuit proceeds through devices. For example, a MOS transistor has its source and drain defined in a *twoWayPath* command so that the pull-up and pull-down processing can trace every net through the source and drain to power and ground.
By selectively defining which nets are power and ground, and what the device paths are, you can use these commands for generalized path tracing.

```plaintext
oneWayPath
twoWayPath
```

**Pull-Up and Pull-Down Tracing**

Pull-up and pull-down tracing starts at every net and goes through each device path looking for power and ground. By selectively defining which nets are power and ground, you can use these commands for generalized path tracing instead of conventional pull-up and pull-down tracing.

```plaintext
checkPullUp
checkPullDown
checkPullUpAndDown
```

**Counting Connectivity**

The following commands select devices and nets based on the number of connections to them. The first commands check for nonfunctional devices and nets because they have one or no connections. The other commands look for specific counts of connections.

Because input and output nets connect externally to the circuit, those nets are treated as if they have at least one connection to a device terminal.

```plaintext
checkFloatingDevices
checkFloatingNets
checkOneNetDevices
checkOneTerminalNets
checkDeviceNetCount
checkFanOut
```

**Checking Connectivity**

The following commands select devices by their direct connections, or lack of direct connections, to specific nets.

```plaintext
checkConnected
checkNotConnected
```
Forming Gates

The following commands let you combine MOS transistors into gates and test the circuit result for feedback loops. In addition, a simple gate-level netlist is created. The program recognizes a preset number of gate configurations. Any devices not forming gates can be seen as gate errors.

During the formation of gates, the program can consolidate individual device parameters into gate parameters using functions you define.

```plaintext
  reduceDevice
displayGateErrors
displayFeedBackLoops
```

Testing Properties

The following commands let you test the parameters of individual devices and gates:

```plaintext
  testDeviceProperty
testGateProperty
```

Program Output

All the errors found by the ERC check are placed in error output files, which the program uses to display and explain errors.

The output files are stored in subdirectories of the run directory. When you use the ERC form, you must specify the run directory where you want to put your results. The program then creates the run directory (if one does not already exist). When you run an ERC check, the resulting error output files are put in the appropriate directories.

The outputs from ERC fit into these categories:

- The `probe.err` error file that you can read to find the name of the net or device causing an error
- Working and information files that you can display by using menu commands
- Control and cross-reference files that you cannot access
- A copy of the rules file under the name `divaERC.rul`
probe.err Files

This text file contains the error output of the ERC program. It is stored in the layout or schematic subdirectories of the run directory. The Diva verification tool highlights the nets and devices referenced in this file when you select Display Errors in the ERC form.

You can also display the contents of this file by selecting Show Run Info in the ERC form.

Working and Information Files

The following files contain information that you can display using the Show Run Info form, which you access from the ERC form.

netlist Text file containing the netlist of the circuit. You can view this file by selecting the Netlist option in the Show Run Info form.

si.out Text file containing all messages and warnings output from the program. You can view this file by selecting the Output option in the Show Run Info form.

si.log Text file containing information pertaining to the program execution and its environment. It also includes summary information of the results of the functions you requested. You can view it by selecting the Log option in the Show Run Info form.

ERC Functions

To make its checking process easier, ERC lets you simplify your circuit by reducing devices and forming gates. It also combines the parameters of the reduced devices so that you can compare them. This section gives a detailed explanation of network reduction and parameter consolidation.

Reducing Networks

ERC gives you three options for reducing (simplifying) your network:

- Series
  Reduces all series devices of the same type to a single device. It takes into account the polarity of devices where this is appropriate. Consolidates parameters if requested.

- Parallel
Reduces all parallel devices of the same type to a single device. It considers the polarity of devices where this is appropriate. Consolidates parameters if requested.

- **MOS**

Recognizes and builds logic gates from groups of MOS transistors. It builds the logic by iteratively combining transistors in logical AND and OR configurations, followed by the combining of pull-up and pull-down logic functions. Consolidates parameters if requested. Any devices not consolidated into logic gates are left as discrete devices.

### Reducing Series Devices

You can combine devices of the same type that are connected in series by “series reduction.”

To be combined in series, the device type must have two terminals. Two devices must be connected by a single net. That net must not be connected to any other device and must connect one of the terminals from each of the devices being combined. If the two terminals of the device type are permutable, it doesn't matter which two are connected. If the two terminals of the device type are not permutable, then the connection must be between unlike terminals.

The following figure illustrates typical series configurations.

![Series Configurations](image)

- Resistors
- Diodes that can be reduced
- Capacitors
- Diodes that cannot be reduced

### Reducing Parallel Devices

You can combine devices of the same type that are connected in parallel by using “parallel reduction.”

You can connect devices in parallel regardless of the number of terminals they have. However, you must connect each terminal of each device directly to a terminal of the same type on another device of the same type. Terminals are of the same type if they are defined as permutable. The nets connecting the terminals can have other connections.
The following figure illustrates typical parallel configurations.

If you use for one device `permuteDevice` commands specifying both `series` and `parallel` options, both options are applied iteratively until no more reduction is possible.

The following figure illustrates complex configurations that reduce to single devices when both series and parallel options are applied.

**Reducing MOS Devices**

You can group together MOS transistors if they are of the same type and form a logical function. Such groups form the logic section of NMOS gates and the load and logic sections of CMOS gates.

In normal MOS layout, the gate structures do not have to match the schematic exactly for the circuit to function. Most gates have alternative forms. Take for example a *three-input AND* structure as shown in this figure.
If the schematic has the original transistors in the order A-B-C and the layout has them in the order B-C-A, the comparison of the circuits fails. The MOS grouping process combines the three separate transistors into a single device with three permutable terminals. With this single device, the terminal ordering doesn't matter and the layout device matches the schematic device, which causes the original devices to match.

This verification tool groups MOS devices using device pairs in exactly the same way as in series and parallel reduction. In this case, however, there are two separate criteria controlling the grouping.

The first grouping criteria is an AND configuration as used in the previous example. If a transistor source/drain connects to the source/drain of another transistor of the same type, and that connecting net has no other connections, then the devices are grouped.

The second grouping criteria is an OR configuration. In this configuration, two transistors have a common net for both their source and drain.

The following figure illustrates AND and OR configurations.
The following figures illustrate how the logic of a complex gate is permutable and how the grouped result is the same for either.

During the MOS grouping, certain transistor configurations can be encountered that function electrically as if they are in parallel but that do not conform to the conventional concepts of parallel. These transistors are merged together as if they are parallel. They are referred to as pseudoparallel.

The following figure illustrates two of these configurations.

If you specify both series and parallel consolidation, this tool applies these two functions iteratively until no further reduction is possible. If you request MOS reduction, series and parallel reduction for those devices is implied.
Defining Device Pin Permutability

To correctly perform reduction, ERC needs to know the permutability of device terminals (whether they can be interchanged). For example, source and drains of an MOS transistor are permutable. This means they can be interchanged. This concept extends to larger, more complex devices, where the permutations can be more complex and less obvious.

You define the pins of a device as permutable or fixed by attaching a property to its cell. This property defines whether the pin can be reduced or whether its relationship must stay unchanged. The system netlister adds this information to the netlist so the ERC program can use it.

The property name is

\texttt{permuteRule}

The recognized keywords in the property are \texttt{p} for permutable and \texttt{f} for fixed. An example of the property contents is

\[( p \ 1 \ 2 )\]

This specifies that terminals 1 and 2 on this device are interchangeable.

This is a more complex example.

\[(p (p \ 1 \ 2 ) (p \ 3 \ 4 ))\]

This specifies that terminals 1 and 2 are permutable, as are terminals 3 and 4. Also, the pair of 1 and 2 are permutable with the pair of 3 and 4.

The following figure illustrates more complex permutability.

```
1
2
3
Macro block
4
5
6
(p(f(p \ 1 \ 2) \ 3)(f(p \ 4 \ 5) \ 6))
```
This specifies that terminals 1 and 2 are interchangeable but fixed relative to 3. Terminals 4 and 5 are interchangeable but fixed relative to 6. Group 1, 2, 3 is permutable with group 4, 5, 6.

**Consolidating and Testing Parameters**

When you reduce a group of devices to one device or form a gate, you must also combine the parameters (properties) of the devices involved. For ERC to correctly consolidate the parameters, you must tell it what to do by supplying a parameter consolidation SKILL routine.

You also can test the parameters (such as width, length, or area) of the reduced devices and gates to ensure they meet your design standards. For ERC to correctly test these parameters, you must supply a test SKILL routine.

**Combining Series/Parallel Parameters**

You can use the *reduceDevice* command to reduce groups of serial and parallel devices of the same type to a single device. Each time you reduce a pair of devices to a single device, the device parameters must be consolidated into a single parameter. This results in a final correct parameter for each device generated for a group of series/parallel devices.

How this consolidation is done depends upon the technology, the application, the type of reduction, the device type, and any other factors you want to introduce. The only way the program can correctly consolidate parameters is for you to tell it what you want it to do. You do this by supplying a routine, written in SKILL, for each type of reduction and device type you require.

The SKILL routine you write must accept as input two lists of parameters, one list for each device. Each list is in the netlist and contains all the parameters for that device. The routine determines if the parameters you want consolidated are actually present in the list and then does the necessary processing to combine them. This product passes the resulting new list of parameters back to the ERC program.

The result can be the resultant parameters only, or a combination of the new ones plus uncombined originals. For example, if ERC is consolidating two resistors in series, it passes the SKILL routine the string of parameters attached to each of them. The SKILL routine, in this case, selects the R parameter and sets $R_{new} = R_1 + R_2$. It then returns $R_{new}$ as the new value of $R$ to be attached to the consolidated device.

You must place the SKILL routines that perform parameter reduction before any of the other ERC commands in the *ercRules( )* section of the verification rules file.
Series/Parallel Example

In the following example, the comments clarify the application and do not describe the basic SKILL structure or syntax. For information on SKILL, refer to the Design Framework II SKILL Functions Reference.

The contents of the procedures (routines) can be as complex as you want, and you can use any of the parameters found in the property strings associated with the devices. For example, you can calculate the width and length resulting from combining two transistors by using their original widths and lengths plus parameters such as bend counts, as well as any constants or factors.

The example shows procedure for combining parameters during a parallel reduction. The numbers at the beginning of the line are for reference in the comments below only and do not appear in the final code.

```skill
1  procedure( parallelMOS( m1, m2 )
2      prog( ( mt )
3          mt = ncons( nil )
4          if( ( m1->L && m2->L )
5              mt->L = ( m1->L + m2->L ) / 2.0
6          )
7          if( ( m1->W && m2->W )
8              mt->W = m1->W + m2->W
9          )
10         return( mt )
11      )
12    )
```

Line 1 Defines the procedure name parallelMOS for a parameter consolidation routine in a reduceDevice statement. It also defines two arguments, m1 and m2, which the routine expects to receive. These two arguments are each lists of all the parameter properties on the devices being combined.

Lines 2 and 3 Sets up a local variable as a list (initially a null) that contains your resulting parameter list for the combined device.

Line 4 Checks to see if both the m1 and m2 lists contain a reference to a parameter called L. If they do, line 5 calculates a resulting value as (L1+L2)/2 and stores it in the new list mt with the parameter name of L.

Line 7 Checks to see if both the m1 and m2 lists contain a reference to a parameter called W. If they do, line 8 calculates a resulting
value as \( W_1 + W_2 \) and stores it in the new list \( mt \) with the parameter name of “\( W \)”

**Line 10**

Returns the resulting list \( mt \) back to calling program, where it is associated with the combined device.

**Combining MOS Parameters**

During generation of the gate-level netlist, two forms of reduction take place: AND and OR. This serves to combine device parameters to form pull-up and pull-down parameters. All devices of the same type and previously reduced structures forming an AND logic configuration are combined into higher level structures. The same is true for the OR configuration. By iterating these processes, the pull-up and pull-down portions of a gate can be reduced to single structures. Later these are combined to form the complete gate.

The following figure illustrates the logic of the AND/OR invert gate.

![Logic Diagram](image-url)

The figure shows the steps for converting the device-level logic into the gate pull-down structure. At each step, ERC reduces two components configured in either an AND or OR configuration to one. At each reduction, the appropriate routine combines the parameters of the two components into one set of parameters. The form of this combination depends on the routine you provide. If width and length are being combined, you might want to calculate an overall average. For beta ratios, you might select the worst- or best-case combination.

**MOS Example**

The following example illustrates both \( orMOS \) and \( andMOS \) routines for combining parameters.

The numbers at the beginning of the line are for reference in the comments below only and do not appear in the final code.
procedure( orMOS( m1, m2 )
  prog( ( mt )
    mt = ncons( nil )
    if( ( m1->l && m2->l && m1->w && m2->w )
        if( ((float( m1->w ) / float( m1->l ))) >
            (float( m2->w ) / float( m2->l )))
      then
        ( mt->w = m1->w )
        ( mt->l = m1->l )
      else
        ( mt->w = m2->w )
        ( mt->l = m2->l )
    )
    return( mt )
  )
)
procedure( andMOS( m1, m2 )
  prog( ( mt )
    mt = ncons( nil )
    if( ( m1->l && m2->l && m1->w && m2->w ) then
      ( mt->w = ( float( m1->w ) + float( m2->w ))
        / 2.0)
      ( mt->l = m1->l + m2->l )
    )
    return( mt )
  )
)

The basic structure is the same as the previous series/parallel example. In the first procedure, orMOS (line 1), the routine checks for parameters w and l in both lists (line 4). If all exist, it checks the w/l ratio of one to the w/l of the other (line 5). The returned list (line 13) has its parameters set to those of the input list having the highest ratio. The concept here is that the final structure contains the parameters for the path (one or the other) that has the highest w/l ratio.

In the second procedure, andMOS (line 16), after checking the existence of all parameters (line 19), the routine sets the resultant w to the average of the two input "w" values (line 20), and the resultant l to the sum of the input l values (line 21). This simulates a structure that has one device in series (AND) with the other.
Combining Gate Parameters

When you combine gate parameters, the order of the input parameters is critical. The first input is always the load structure (pull-up) parameter list, and the second is always the logic structure (pull-down) parameter list.

The other consolidations normally create a parameter list containing exactly the same parameter names as their input. For gate parameter consolidation, however, you can create a completely different parameter list from the input.

Gate Example

This example follows the structure of those given previously, but creates the parameter betaRatio from the input w and l parameters.

```
1  procedure( gateMOS( m1, m2 )
2      prog( ( mt )
3          mt = ncons( nil )
4          if( ( m1->l && m2->l && m1->w && m2->w )
5              then
6                   mt->betaRatio = ( float(m1->w) / float(m1->l ) /
7                                 ( float(m2->w) / float(m2->l ) ) )
8                  )
9          )
10    )
```

After testing for the existence of the required parameters in the pull-up and pull-down inputs (line 4), the routine calculates a new parameter, called betaRatio, from the ratio of the pull-up (input 1) w/l over the pull-down (input 2) w/l (line 5).

Testing Gate and Device Properties

The testing functions are simpler than the reduction consolidation functions in that they only have to process the parameters of a single device or gate; they do not have to generate a parameter list.

For each device whose type is specified in a testDeviceProperty or testGateProperty command, the SKILL routine you specify uses a single argument, which is the parameter list of the device. If the parameter list does not exist, your routine can either treat this as an error condition or not. Inside the routine you can do whatever processing you wish on any or all parameters in the list. You can test their values, combinations of values, or the existence of the parameters.
By testing, you can decide whether the parameters are acceptable or not. If the parameters fail your test, you must return \textit{t} (for true), or a text string which is appended to the error message associated with that device. If the parameters pass your tests, you must return \textit{nil}. The error text string automatically contains the relevant information of the device being tested, so the appended message can best be used to define the value against which the test failed.

The following example illustrates a simple test in which the parameters \textit{w} and \textit{l} are tested against a predefined ratio.

```plaintext
1  procedure( testMOS( m1 )
2      prog( ( )
3          if( ( m1->L && m1->W ) then
4              if( ( m1->W / m1->L ) != 3
5                  return( "Wanted 3." )
6              )
7          else return ( t )
8      )
9      return( nil )
10 )
```

Line 1 Defines the procedure name \textit{testMOS}, which is the name used for the function in the \textit{testDeviceProperty} command. It also defines the single argument \textit{m1}, which is a list of parameter properties. This list contains all the properties available on the device being tested.

Line 3 Checks to see if the parameter list contains both \textit{L} and \textit{W} before continuing. If it does not, line 7 returns the value \textit{t} (TRUE). The error message associated with this return automatically tells you if no parameter list exists. If you want to be more precise and specify that a particular parameter does not exist, then you must define a separate \textit{if} test for each parameter name and provide a separate return text for each case.

Line 4 Tests the ratio of \textit{W} divided by \textit{L} against the constant 3.

If the comparison fails, line 5 returns the text string “Wanted 3.” If the comparison succeeds, line 9 returns the value \textit{nil} (FALSE).

The following example illustrates a simple gate property test.

```plaintext
1  procedure( testGate( m1 )
2      prog( ( )
3          if( ( m1->betaRatio )
```

June 2000 465 Product Version 4.4.6
In this example, line 3 checks for the existence of the parameter \textit{betaRatio}. Line 4 checks the parameter against the constant 1.3. If it is greater, an error is generated by returning \textit{t} in line 5; otherwise, it is accepted by returning \textit{nil} in line 8.

\section*{ERC Commands}

This section describes the syntax for the commands used to define circuit connections and trace paths.
oneWayPath

oneWayPath( device terminal1 terminal2 )

Description

Specifies that during path tracing for pull-up and pull-down detection, this device is to be traced only from the net connected to terminal1 to the net connected to terminal2.

Fields

device  Name of the device to which this path definition applies. The name is a text string enclosed in quotation marks.
terminal1  Device terminal name from which the path tracing can start. The name is a text string enclosed in quotation marks.
terminal2  The device terminal name to which the path tracing goes. The name is a text string enclosed in quotation marks.

Example

The following example illustrates a diode in which the path tracing can go from anode to cathode but not from cathode to anode:

oneWayPath( "diode" "anode" "cathode" )
setGround

setGround( netName ...)

Description

Defines a list of nets that represent the ground connections to the circuit. These nets are used in the pull-down path tracing and are required for the recognition of gates during reduceDevice command processing.

Fields

netName

Name of a net that represents a ground connection to the circuit. You can define any number of nets. Each net is a character string enclosed in quotation marks.

Example

The following example declares VSS and gnd! as ground supplies for the ERC and device reduction programs:

    setGround( "VSS" "gnd!" )
**setInput**

`setInput( netName ...)`

**Description**

Defines a list of nets that represent the input connections to the circuit. These nets are used to eliminate false errors in the pull-up and pull-down path tracing. In addition, they are used to count single connections when looking for nets and devices that have one or zero connections.

**Fields**

`netName` The name of a net that represents an input connection to the circuit. You can define any number of nets. Each net is a character string enclosed in quotation marks.

**Example**

The following example declares all specified net names as input nets:

```
setInput( "ph1" "ph2" "clock" "in1" "in2" )
```
setOutput

setOutput( netName ...)  

Description

Defines a list of nets that represent the output connections to the circuit.

You use these nets to count a single connection when looking for nets and devices that have one or zero connections.

Fields

netName  
Name of a net that represents an output connection to the circuit. You can define any number of nets. Each net is a character string enclosed in quotation marks.

Example

The following example declares all specified net names as output nets:

    setOutput( "out1" "out2" "trigger" "clock" )
setPower

setPower( netName ...)

Description

Defines a list of nets that represent the power connections to the circuit. These nets are used in the pull-up path tracing and are required for the recognition of gates during reduceDevice command processing.

Fields

netName

Name of a net that represents a power connection to the circuit. You can define any number of nets. Each net is a character string enclosed in quotation marks.

Example

The following example declares VDD and vdd! as power supplies for the ERC and device reduction programs.

setPower( "VDD" "vdd!" )
twoWayPath

twoWayPath( device terminal1 terminal2 )

Description

Specifies that during path tracing for pull-up and pull-down detection, this device can be traced from the net connected to terminal1 to the net connected to terminal2 and in the opposite direction from the net connected to terminal2 to the net connected to terminal1.

Fields

device
Name of the device to which this path definition applies. The name is a text string enclosed in quotation marks.

terminal1
Name of the device terminal name at which the path tracing can start or end when passing through the device. The name is a text string enclosed in quotation marks.

terminal2
Name of the device terminal where the path tracing goes when passing through the device. The name is a text string enclosed in quotation marks.

Examples

The following examples illustrate typical devices that have two-way trace paths:

twoWayPath( "nfet" "S" "D" )
twoWayPath( "resistor" "PLUS" "MINUS" )

Tracing and Connectivity Commands

This section describes the syntax for the commands used to trace pull-ups and pull-downs and check connectivity.
checkConnected

checkConnected( device terminal net )

Description

Finds devices based on the connection of specific device terminals to defined nets. ERC selects each device of the type specified if any of the specified terminals connect to any of the specified nets.

All selected devices are referenced in the error file with a message. You can display these devices by selecting Display Errors in the ERC form.

Fields

device Name of a device, consisting of a text string enclosed in quotation marks. Only connections to this device are considered.

terminal Name of a device terminal, consisting of a text string enclosed in quotation marks or a list of device terminals surrounded by parentheses. The terminals must belong to the device previously defined. Only connections to these device terminals are considered.

net A single net name enclosed in quotation marks or a list of net names surrounded by parentheses. If any of the device terminals connect to any of these nets, the device is selected.

Examples

The following example flags all nfet devices whose gates are connected to vdd.

    checkConnected( "nfet" "G" "vdd")

The following example flags all nfet devices whose sources or drains are connected to vdd or VDD.

    checkConnected( "nfet" ( "S" "D") ("vdd" "VDD") )

The following example flags all pfet devices whose source, drain, or gate terminals are connected to vss or gnd.

    checkConnected( "pfet" ( "S" "D" "G") ( "vss" "gnd") )
checkDeviceNetCount

checkDeviceNetCount( device terminal limit )

Description

Finds devices based on the number of distinct nets connected to them through specific terminals. The limit argument defines the number of nets. If the number of nets meets these limits, the device is selected.

A single net connected to more than one of the terminals of a device is counted only once. A net is counted only if it connects to more than one device terminal, or if it connects a device terminal to an I/O of the circuit. If a terminal of a device is floating (it does not connect to another device terminal or an I/O of the circuit), it is considered unconnected.

All selected devices are referenced in the error file with a message. To display these devices, select Display Errors in the ERC form.

Fields

device Name of the device to which this command applies. The name is a text string enclosed in quotation marks.

terminal Name of a device terminal. The name is a text string enclosed in quotation marks. The terminal must belong to the device previously defined. Only nets connected to this device terminal are counted. This can also be a list of terminal names surrounded by parentheses.

limit Limit specification. The options are as follows:

- lower_limit operator keyword operator upper_limit
- keyword operator upper_limit
- keyword operator operator lower_limit

You can use the following keywords:

- keep
- ignore

You can use the following operators:
You can use the following limit specifications:

- `keep < 3`
- `2 < ignore < 5`
- `ignore > 5`
- `keep == 3`

**Examples**

```python
checkDeviceNetCount( "nppn" "e" keep < 3 )
checkDeviceNetCount( "nppn" ( "e" "c" ) 2 < ignore < 5 )
checkDeviceNetCount( "nppn" ( "e" "b" "c" ) keep > 5 )
checkDeviceNetCount( "nppn" ( "S" "D" ) keep == 1 )
```
checkFanOut

checkFanOut( device terminal limit [usage net] )

Description

Finds nets based on the number of device terminals of a specific type connected to them. You define the number of terminals with the limit definition in the command. If the count of terminals meets these limits, the net is selected.

You can extend the command to specify exactly which nets are to be checked or which nets are not to be checked.

All selected nets are referenced in the error file with a message. You can display these nets by selecting Display Errors in the ERC form.

Fields

device Name of a device, consisting of a text string enclosed in quotation marks. Only connections to this device are counted for any net.

terminal Name of a device terminal, consisting of a text string enclosed in quotation marks. The terminal must belong to the device previously defined. Only connections to this device terminal are counted for any net. This can also be a list of terminal names surrounded by one set of parentheses.

limit Limit specification. The options are as follows:

lower_limit operator keyword operator keyword operator upper_limit

keyword operator upper_limit

keyword operator1 lower_limit

You can use the following keywords:

keep
ignore

You can use the following operators:
You can use the following limit specifications:

- keep < 10
- 2 < ignore < 8
- ignore > 5
- keep == 3

**Usage**

Optional keyword that precedes a net name or list of net names. The options are as follows:

- `for`
- `except`

The `for` keyword precedes a list of nets that are to be checked. The `except` keyword precedes a list of nets that are not to be checked. Without the `usage` keyword, all nets are checked.

**Net**

A single net name enclosed in quotation marks or a group of net names surrounded by parentheses. This list is applied according to the preceding keyword `for` or `except`.

**Examples**

The following example flags all nets that connect to more than three gate terminals (terminals named `G`) of `n` channels (devices whose model name is `nfet`).

```
checkFanOut( "nfet" "G" keep > 3 )
```

The following example flags only nets named `ph1` and `ph2` that connect to more than ten gates of `nfet` devices.

```
checkFanOut( "nfet" "G" keep > 10 for ("ph1" "ph2") )
```
The following example flags all nets that connect to more than one pad source or drain.

    checkFanOut( "pfet" ( "S" "D") keep > 1 )

The following example flags all nets except vdd and gnd that connect to more than five gates of nfet devices.

    checkFanOut( "nfet" "G" keep > 5 except ("vdd" "gnd") )

The following example flags all nets, except a1, a2, and a3, that connect to more than two but less than five e or c terminals on npn transistors.

    checkFanOut( "npn" ("e" "c") 2 < ignore < 5 except ("a1" "a2" "a3") )
checkFloatingDevices

checkFloatingDevices( )

Description

Finds all devices that have no connections to the remainder of the circuit. Although each of the device’s terminals connects to a net, none of those nets connect to terminals of other devices.

All floating devices are referenced in the error file with a message. You can display floating devices by selecting Display Errors in the ERC form.

Example

    checkFloatingDevices( )
checkFloatingNets

checkFloatingNets( )

Description

Finds all nets with no connections to device terminals in the circuit. A reference to a net in a setInput or setOutput command is considered a single connection to the net.

All floating nets are referenced in the error file with a message. You can display floating nets by selecting Display Errors in the ERC form.

Example

    checkFloatingNets( )
checkNotConnected

checkNotConnected( device terminal net )

Description

Finds devices based on the absence of connection of specific device terminals to defined nets. If none of the specified terminals connect to any of the specified nets, ERC selects each device of the type specified.

All selected devices are referenced in the error file with a message. You can display these devices by selecting Display Errors in the ERC form.

Fields

device

Name of a device, consisting of a text string enclosed in quotation marks. Only connections to this device are considered.

terminal

Name of a device terminal, consisting of a text string enclosed in quotation marks or a group of device terminals surrounded by parentheses.

The terminals must belong to the device previously defined. Only connections to these device terminals are considered.

net

A single net name enclosed in quotation marks or a list of net names surrounded by parentheses. If none of the device terminals connect to any of these nets, the device is selected.

Examples

The following example flags all pfet devices whose source or drain terminals are not connected to vdd.

    checkNotConnected( "pfet" ( "S" "D") "vdd" )

The following example flags all cap devices that have neither terminal connected to vdd or gnd.

    checkNotConnected( "cap" ( "PLUS" "MINUS") ( "vdd" "gnd" ) )
checkOneNetDevices

checkOneNetDevices( )

Description

Finds all devices that have only one connection to the remainder of the circuit. Although each of the device’s terminals connects to a net, only one of those nets connects to a terminal of another device.

All single net devices are referenced in the error file with a message. You can display these nets by selecting Display Errors in the ERC form.

Example

    checkOneNetDevices( )
checkOneTerminalNets

checkOneTerminalNets( )

Description

Finds all nets that have only one connection to a device terminal in the circuit. A reference to a net in a setInput or setOutput command is considered a single connection to the net.

All single terminal nets are referenced in the error file with a message. You can display these nets by selecting Display Errors in the ERC form.

Example

    checkOneTerminalNets( )
checkPullDown

checkPullDown( )

Description

Starts a trace at every net in the circuit other than the power nets and determines whether that trace can reach the nets defined as ground. It traces through devices as defined by the oneWayPath and twoWayPath definitions. The command has no arguments.

If you use the usual definitions of ground, with the source and drain of MOS transistors defined in a twoWayPath definition, this command performs the conventional pull-down checking.

All nets that cannot be traced (pulled down) are referenced in the error file with a message. You can display these nets by selecting Display Errors in the ERC form.

Prerequisite

You must first define ground connections and device paths.

Example

The following example checks all nets, except those declared as inputs by setInput, for a path to a declared ground supply.

    checkPullDown( )
checkPullUp

checkPullUp( )

Description

Starts a trace at every net in the circuit other than the ground nets and determines whether
that trace can reach the nets defined as power. It traces through devices as defined by the
oneWayPath and twoWayPath definitions. The command has no arguments.

If you use the usual definitions of power, with the source and drain of MOS transistors defined
in a twoWayPath definition, this command performs conventional pull-up checking.

All nets that cannot be traced (pulled up) are referenced in the error file with a message. You
can display these nets by selecting Display Errors in the ERC form.

Prerequisite

You must first define power connections and device paths.

Example

The following example checks all nets, except those declared as inputs by setInput, for a path
to a declared power supply.

    checkPullUp( )
checkPullUpAndDown

cHECKPULLUPANDDOWN( )

Description

Starts a trace at every net in the circuit other than the power and ground nets and determines whether or not that trace can reach the nets defined as power and ground. It traces through devices as defined by the oneWayPath and twoWayPath definitions. The command has no arguments.

If you use usual definitions of power and ground, with the source and drain of MOS transistors defined in a twoWayPath definition, this command performs conventional pull-up and pull-down checking.

All nets that cannot be traced to power or to ground (pulled up or down) are referenced in the error file with a message. You can display these nets by selecting Display Errors in the ERC form.

Prerequisite

You must first define power connections and device paths.

Example

The following example checks all nets, except those declared as inputs by setInput, for a path to declared power and ground supplies.

cHECKPULLUPANDDOWN( )

Gates and Parameters Commands

This section describes the syntax for the commands used to form gates and test parameters.
displayFeedBackLoops

displayFeedBackLoops( )

Description

Requests display of all loops in the circuit after the circuit has been reduced to the gate level using the reduceDevice command with the MOS keyword.

A loop is formed when the output of a gate can be traced, through inputs of other gates, back to the input of the original gate. ERC detects all loops, whether they are isolated, nested or crossing. Each loop is given a loop number and a subloop number. The subloop number is used when a single loop has multiple paths within it.

All nets forming loops are referenced in the error file with a message. You can display nets forming loops by selecting Display Errors in the ERC form.

Prerequisites

This command is applicable only after using a reduceDevice command with the MOS keyword.

The following figure illustrates the formation of loops and subloops.

Example

displayFeedBackLoops( )
displayGateErrors

displayGateErrors()

Description

Requests display of all transistors that have failed to become part of a gate during device reduction initiated with the `reduceDevice` command with the `MOS` keyword.

If individual transistors do not form part of any of the recognized gates, they are left in the circuit and form part of the netlist. This command simplifies the isolation of those devices in cases where you assumed the devices formed gates. It is probable that gates have not been formed because the transistors are not correctly connected, or they form a gate type not recognized by the program.

All selected devices are referenced in the error file with a message. You can display these devices by selecting `Display Errors` in the ERC form.

Prerequisites

This command is applicable only after a `reduceDevice` command with the `MOS` keyword.

Example

    displayGateErrors()
reduceDevice

reduceDevice( type device [key] [spFunction]
 [andFunction orFunction])

Description

Requests that the circuit netlist be simplified (reduced) by combining series and parallel devices, creating logic gates from MOS transistors, or both.

During reduction, you can consolidate the original parameters defined as properties of the devices into new parameters by using SKILL functions you write. To create logic gates, you can define SKILL functions that calculate the gate-level parameters from the parameters of the load and logic portions of the gate.

If this command produces a reduced logic gate network, it writes out the network in text format in a form that can be directly used or translated for use by other programs.

Prerequisites

For full reduction, you must specify the permutability of device terminals in the device definitions in the netlist. To help this command create logic gates, use the setPower and setGround commands.

Fields

type

Defines the type of reduction to be performed. The options are as follows:

series

Merges devices of the same type connected in series into a single device. You can specify two-terminal or MOS devices. If the two device terminals are permutable, this command merges the devices regardless of the way they are connected. If the two terminals are not permutable, the command merges the devices only if unlike terminal types are connected.

parallel

Merges into one all the terminals of two devices of the same type that have common nets.

MOS

Creates logic gates out of groups of transistors by recognition of pairs of devices in AND and OR configurations. It can handle NMOS or CMOS technologies and is limited to creating gates of
these types. These definitions show the terminal ordering of the resultant gate. The \textit{nand}, \textit{nor}, and \textit{complex} gates are illustrated by single examples, but the full range of configurations is available.

\begin{verbatim}
  inverter  out in
  nmos xfer gatedrain in source
  cmos xfer gatedrain nin pin source
  tristate  out common nin pin
  nand3     out in1 in2 in3
  nor3      out in1 in2 in3
  aoi321    out in3a in3b in3c in2a in2b in1a
  oai321    out in3a in3b in3c in2a in2b in1a
\end{verbatim}

For a MOS device to be reduced correctly, its terminal ordering in the netlist must conform to SPICE conventions of (S G D B), and in the device definition, it must have the correct source and drain permutation statement (p S D).

The keyword also triggers the creation of a netlist of the reduced circuit in the \textit{si.out} file.

\textbf{Note:} The MOS reduction significantly reduces the number of devices and the output netlist size accordingly.

If you specify multiple reductions for the same device, all are applied iteratively. For example, if you request series and parallel reduction for resistors, both are applied to resistor networks, reducing them as far as possible.

The following figure illustrates normal series and parallel reduction.

\begin{figure}
\centering
\includegraphics[width=0.8\textwidth]{parallel_reduction.png}
\caption{Parallel reduction}
\end{figure}

MOS reduction also implies parallel reduction plus reduction of pseudoparallel device combinations. In these cases, the devices are not truly connected in parallel but behave electrically as if they were.
The following figure illustrates pseudoparallel reduction during MOS reduction.

![Diagram of pseudoparallel reduction](image)

**device**
Name of a device consisting of a text string enclosed in quotation marks. The reduction process requested is applied to all devices of this type.

**key**
The options are as follows:

**logic**
Used only during MOS reduction to specify that the type of the device in this command forms the logic part of the gate. In MOS, this is usually the *nfet* or its equivalent.

ERC recognizes gates in two parts. The load, which is normally the pull-up-to-power section, and the logic, which is normally the pull-down-to-ground section. During reduction, both sections are formed separately and then paired together with the output of the gate being the common net between the sections.

**netlist**
Used for series and parallel reduction. The key "netlist" is added to one of the *reduceDevice* commands. This keyword triggers the creation of a netlist of the reduced circuit in the *si.out* file. Used if the MOS reduction has not been applied but a netlist is wanted after series and parallel reduction only.

**spFunction**
Optional name of a user-defined SKILL routine used to consolidate the device parameters (stored as properties on the devices) during series and parallel reduction.

This function is usually not used for MOS processing. The only exception is for pseudoparallel configurations.

The first function name encountered after the *series* or *parallel* keyword is assumed to be the *sp_function*.
andFunction  Optional name of a user-defined SKILL routine used to consolidate the parameters (stored as properties on the devices) during MOS AND reduction.

This function cannot be used for series and parallel processing.

The first function name encountered after the MOS keyword is assumed to be the andFunction.

During MOS reduction, ERC combines devices by recognizing either an AND or an OR configuration. An AND configuration occurs when two devices of the same type have a common source/drain net with no other connection to that common net, and their other source/drain connections do not have common nets.

orFunction  Optional name of a user-defined SKILL routine used to consolidate the parameters (stored as properties on the devices) during MOS OR reduction.

This function cannot be used for series and parallel processing. This is the second function after the MOS keyword.

The first function after the MOS keyword is considered to be the andFunction.

During MOS reduction, ERC combines devices by recognizing either an AND or an OR configuration. An OR configuration occurs when two devices of the same type have both their source and drain terminals connected to common nets, but not their gate terminals.

Examples

The following example reduces all resistor devices in series.

```
reduceDevice( series "resistor" )
```

The following example reduces all capacitor devices in parallel and calls a SKILL function to combine their parameters.

```
reduceDevice( parallel "capacitor" series_cap )
```

The following example reduces all nFET devices and references them as logic forming.

```
reduceDevice( MOS "nfet" logic )
```

The following example reduces all pFET devices and calls SKILL functions to consolidate parameters during reduction of AND and OR configurations.

```
reduceDevice( MOS "pFet" mos_and mos_or )
```
testDeviceProperty

testDeviceProperty( device testFunction )

Description

Lets you check the values of parameters stored on the devices as properties. ERC does this checking after doing any reduction. This means that for series and parallel reduction, ERC checks the parameters after they have been processed by the series and parallel consolidation functions.

For MOS reduction, the original parameters are maintained and checked on the original devices except in those cases where pseudoparallel reduction invoked the parallel consolidation function. ERC uses the AND and OR consolidation functions to create parameters for the resultant logic and load structures. This command does not check these parameters.

If you want to do checking prior to a reduction, you must make a separate run with the reduction commands omitted.

All devices failing the parameter test are referenced in the error file with a message. You can display these devices by selecting Display Errors in the ERC form.

Fields

device

Name of a device, consisting of a text string enclosed in quotation marks. ERC applies the requested parameter testing to all devices of this type.

testFunction

Name of a user-defined SKILL routine used to perform the parameter testing. All devices of the specified type are processed when you specify this function.

Examples

The following example flags all nfet devices whose properties fail the user-defined SKILL evaluation.

    testDeviceProperty( "nfet" testNfetParams )

The following example flags all resistor devices whose properties fail the user-defined SKILL evaluation.
testDeviceProperty( "resistor" resistor_WL_check )
**testGateProperty**

```plaintext
testGateProperty( createFunction [testFunction] )
```

**Description**

Provides data for the following processes:

- Defines the SKILL routine you used to create gate parameters after the gate level reduction has been performed. During gate level reduction, ERC creates load and logic sections, each gathering parameters from the `AND` and `OR` functions provided by the `reduceDevice` command. ERC passes these parameters to the SKILL routine, which then creates a set of parameters to be associated with the final gates created from the load and logic sections.

- Defines the SKILL routine to be used to test the values of the resultant gate parameters created by this routine.

All devices failing the parameter test are referenced in the error file with a message. You can display these devices by selecting *Display Errors* in the ERC form.

**Fields**

- **createFunction**: Name of a user-defined SKILL routine used to create the gate parameters.

- **testFunction**: Name of an optional user-defined SKILL routine used to perform the gate parameter testing. All gates are processed if you specify this routine.

**Example**

The following example calls a user-defined SKILL function (*create_gate_prop*) that creates gate-level parameters associated with the previously combined parameters that resulted from the device reduction. The second function (*test_gate_prop*) begins a user-defined check on the final parameters.

```plaintext
testGateProperty( create_gate_prop test_gate_prop )
```
Comparing Layout to Schematic (iLVS)

Introduction

The Layout Versus Schematic (LVS) program compares two versions of a circuit and isolates any differences. You can use it to compare two layouts, two schematics, or a layout and a schematic.

LVS works as follows:

- Generates a netlist for each representation of the circuit.
- Compares the circuit versions using the verification rules commands you enter in the LVS rules file and the run options you specify from the menu.
- Compares device parameters using an internal feature that combines parameters during series and parallel reduction.
- Lets you provide correspondence points to improve error isolation. Correspondence points identify equivalent devices or nets in each representation of the circuit.

**Note:** The LVS program normally does not need correspondence points and gives good results in most circuits without them.

- Outputs error files that can be used to view and probe errors.
- Displays and explains the errors it found.

The LVS program is controlled by the system simulation environment (se). The simulation environment is set up by your system administrator and normally you do not need to know about it. However, to compare symbolic cellviews or macro cells, you must alter the environment defaults. See Simulation and Environment Control for more information.

You can run LVS using the LVS command from the Verify menu or using the ivLVS SKILL function. You can also run LVS in the UNIX environment.

The LVS comparison algorithm uses a combination of “signature analysis” and “circuit tracing” algorithms to provide the best possible matching of the two circuits with minimum additional input from users. However, there are situations where the circuits are ambiguous.
and the normal processing cannot clearly identify the matching. In these situations, users can introduce the device parameters into the algorithm to act as a “tie-breaker.” For more information, see the “Parameter Analysis” section and the parameterMatchType function description in this chapter.

Prerequisites

The prerequisites for LVS are as follows:

■ You must define LVS verification rules in a text file.
■ You must extract networks for both cellviews so that LVS can generate the netlists it needs.

To create the layout network, run an extraction on the layout using the Extract command from the Verify menu to get an extracted view.

Note: Sometimes the extracted view contains devices, such as parasitic devices, that you do not want to include in the netlist for LVS checking. For any devices that you do not want to include in the netlist, you must set the lvs view property nlAction to ignore. The nlAction property must be placed in the property list of the device’s cellview.

To create the schematic network, run a check on the schematic using Edit – Check – Current Cellview.

■ You must initiate LVS from a window containing either a schematic cellview or an extracted layout cellview.

Capabilities

You control LVS processing by entering verification rules commands in the technology file and by choosing options in the LVS form, which you access by selecting the LVS command from the Verify menu. The verification rules commands define the type of verification checks you want to run. The LVS form options define requirements for a specific run.

The verification rules commands can do the following:

■ Permute (interchange or combine) devices
■ Ignore device terminals
■ Ignore extraneous devices
■ Analyze parameters

The LVS form offers the following options:
Diva Reference
Comparing Layout to Schematic (iLVS)

- Device fixing
- Correspondence points

Additional capabilities are as follows:
- Network terminal processing
- Macro cell processing

Backannotation

You can backannotate the names of the nets and instances in the schematic view of a circuit to the extracted view of the circuit after LVS runs.

You can perform backannotation by executing the Diva verification `lvsbx` stand-alone in the UNIX environment. This function is described in detail at the end of this chapter.

Verification Rules Functions

This section discusses the functions performed by the LVS verification rules commands. You can find the syntax for these commands at the end of this chapter.

Permutating Devices

The `permuteDevice` command lets you reduce (combine) all devices of the same type to a single device. This command lets you reduce series, parallel, and MOS devices, and define functions which combine the device parameters during this reduction process.

Permutation eliminates physical differences where the physical differences have no effect on the functional equivalence of the circuits. Simple examples of this are the reduction of series and parallel devices into single devices, and the pin permutability of MOS transistor sources and drains.

If a terminal name exists in both cellviews, devices cannot be permutated across that terminal. No series, parallel, or MOS permutation absorbs a terminal.
Series Reduction

You can combine devices of the same type that are connected in series by using “series reduction.” To combine in series, your device type must have two terminals, be a MOS device, or have a three-terminal resistor configuration.

For two terminal devices the two devices must be connected by a single net. That net must not be connected to any other device and must connect one of the terminals from each of the devices being combined. If the two terminals of the device type are permutable, it doesn’t matter which two are connected. If the two terminals of the device type are not permutable, then the connection must be between unlike terminals.

The following figure illustrates typical series configurations.

- Resistors
- Diodes that can be reduced
- Capacitors
- Diodes that cannot be reduced

For MOS devices to be combined in series, they must share a common gate and a common source or drain region.

You can specify three terminal resistors and capacitors to be reduced in series. They must be connected in this manner.
This verification tool has no way of knowing from device names which devices are correctly configured for this form of permuting. To detect this configuration, it therefore applies some basic device structure and connectivity checks.

This form of series reduction is only applied if

- The device has three terminals
- The first and second terminals are defined as permutable

\[ d \text{ res3 t1 t2 t3 ( p t1 t2 )} \]
- The two devices connected in series have a common net on their third terminal

**Parallel Reduction**

Devices of the same type that are connected in parallel can be combined by “parallel reduction.”

Devices can be connected in parallel regardless of the number of terminals they have. Each terminal of each device must be connected directly to a terminal of the same type on another device of the same type. Terminals must be of the same type if they are defined as permutable. The nets connecting the terminals can have other connections.

The following figure illustrates typical parallel configurations.

If the same device has *permuteDevice* commands specifying both *series* and *parallel* options, both are applied iteratively until no more reduction is possible.

The following figure illustrates complex configurations that reduce to single devices when both series and parallel options are applied.
MOS Reduction

MOS reduction is limited to MOS transistors that can be grouped together if they are of the same type and form a logical function. Such groups form the logic section of NMOS gates and the load and logic sections of CMOS gates.

In normal MOS layout, the gate structures do not have to match the schematic exactly for the circuit to function. All gates have alternative forms. Take for example a three-input AND structure as shown in this figure.

If the schematic has the original transistors in the order A-B-C and the layout has them in the order B-C-A, the comparison of the circuits fails. The MOS grouping process combines the three separate transistors into a single device with three permutable terminals. With this single device, the terminal ordering doesn’t matter and the layout device matches the schematic device, which causes the original devices to match.

The MOS grouping is done with device pairs in exactly the same way as series and parallel reduction. In this case, however, there are two separate criteria controlling the grouping.

The first grouping criteria is an AND configuration as used in the previous example. If a transistor source/drain connects to the source/drain of another transistor of the same type, and that connecting net has no other connections, then the devices are grouped.

The second grouping criteria is an OR configuration. In this configuration, two transistors have a common net for both their source and drain.
The following figure illustrates *AND* and *OR* configurations.

![AND and OR Configurations](image)

The following figures illustrate how the logic of a complex gate is permutable and how the grouped result is the same for either.

![Logic Illustrations](image)

During the MOS grouping, this product can encounter certain transistor configurations that function electrically as if they were in parallel but that do not conform to the conventional concepts of parallel. They are referred to as pseudoparallel. This verification tool merges
these transistors together as if they were parallel. The following figure illustrates two of these configurations.

![Device Pin Permutability Diagram](image)

**Device Pin Permutability**

The concept of pin permutability within a device is used throughout the LVS program. For example, source and drain of an MOS transistor are permutable. This means they can be interchanged. This concept extends to larger, more complex devices, where the permutations can be more complex and less obvious.

You define the pins of a device as permutable or fixed by attaching a property to its cell. This property defines whether the pin can be reduced or whether its relationship must stay unchanged. The system netlister adds this information to the netlist so the LVS program can use it.

The property name is

permuteRule

The recognized keywords in the property are `p` for permutable and `f` for fixed. An example of the property contents is

( p 1 2 )

and specifies that terminals 1 and 2 on this device are interchangeable.

A more complex example is

( p ( p 1 2 ) ( p 3 4 ) )

which specifies that terminals 1 and 2 are permutable, as are terminals 3 and 4. Also, the pair of 1 and 2 are permutable with the pair of 3 and 4.
The following figure illustrates more complex permutability.

This specifies that terminals 1 and 2 are interchangeable but fixed relative to 3. Terminals 4 and 5 are interchangeable but fixed relative to 6. Group 1, 2, 3 is permutable with group 4, 5, 6.

**Ignoring Device Terminals**

The *ignoreTerminal* command forces the program to do its comparison without considering specific device terminals.

During comparison of two networks, it is sometimes useful to be able to ignore specific device terminal types and continue processing as if they did not exist. For example, MOS transistors can be represented with either three or four terminals. The schematic can show the fourth terminal connected to either power or ground. In CMOS this is the substrate or Pwell which, in turn, has contacts to power and ground.

During layout of small areas of the circuit, the Pwell is included, but a connection of power to that Pwell might not be included. A four-terminal verification would show a mismatch with the schematic that has the back-gate of the N-channel transistors connected to power. Larger areas of the layout might include the Pwell contact, but not include a contact to substrate, so the “ground” connection is not made to the back-gates of P-channel transistors. Only in the full layout are all back-gate connections correct.

It is difficult to modify the schematic each time such a situation occurs. The easiest solution is to reference four terminal devices in the schematic and to tell the LVS program to ignore the back-gates during verification of the smaller circuit areas. On the full circuit verification, the back-gate can be included.
Ignoring Extraneous Devices

The `pruneDevice` command tells the program to ignore devices in the circuit that are electrically nonfunctional.

Certain technologies and layout techniques create extraneous devices that are not functional in the final circuit. Examples of this are unused gate-array devices and unused options. These devices are not normally defined in the circuit schematic and appear as unmatched in an LVS run.

For a small number of such devices, the results might be acceptable; but for any significant number, the true matching of the networks is masked. To allow the LVS program to work correctly, you must remove the unmatched devices before performing the comparison.

You can remove (prune) these devices by requesting specific modes of cleanup for specific device types. The program locates all the conditions you specify and removes the devices from further processing.

There are two forms of pruning. One eliminates any device if it is connected only to a single net. The other is more complex and is designed for MOS transistors. If the devices have terminal connections that do not contribute to the circuit flow, such as MOS back gates and other bulk connections, you can optionally ignore these during device pruning.

The following figure illustrates various devices that are pruned because they are connected to a single net. The open net ends represent unconnected terminals or nets with no other connection.

![Pruned Devices Diagram]

The following figure illustrates various transistor configurations that can be pruned. Some devices appear to be connected in such a way that they are not pruned, but the iterative
processing removes some devices in the first pass, leaving the remaining devices badly connected. In subsequent passes these devices get pruned.

If you specify parallel reduction, parallel devices are reduced prior to extraneous device removal.

Examples

    pruneDevice( MOS "nfet" )
    pruneDevice( general "capacitor" )
Removing Unwanted Devices

The `removeDevice` command lets you remove devices from the network during an LVS run prior to the circuit matching. This command lets you control the removal through a SKILL function that can operate on the device parameters. It also lets you define which terminals of the device to short together and which to leave open.

For capacitors, this device removal exists to some degree by using netlisting control to determine whether a device should be generated in the netlist or not. However, for series devices such as resistors, netlist control cannot merge the remaining nets together. Removing a resistor implies shorting together the original resistor terminals to form a single net from the two original nets. Other devices require a combination of both shorting and opening of nets. For example, a three-terminal resistor has the source and drain shorted but the back-gate left open.

The devices removed by the `removeDevice` command are treated as if they had been removed by a prune request, and you can see them by highlighting the pruned devices.

Device removal does not affect cross probing. If a removed device is cross probed, there is no matching device. If a net is cross probed, this product determines whether the net was merged with others due to device removal and highlights all affected nets in both the schematic and layout.

Parameter Analysis

The `compareDeviceProperty` command lets you request comparison of device properties (parameters) after the devices have been matched.

LVS can compare device parameters defined in the schematic with those extracted from the layout. The schematic parameters are either those specified on the schematic itself for each device instance, or those specified via the netlister as being default values for each device type. The extracted parameters are those derived from the layout by measurement in the `extraction` program and represent the true values.

The parameters are extracted by the program from the circuit netlist. You can generate them directly in the netlist or via the simulation environment netlisting process, which is driven by the device parameters and the `nlpglobals` control file. The format of these parameters is defined in the netlist format section of the “Simulation and Environmental Control” appendix.

There are two parts to the parameter processing in the LVS program: the consolidation of parameter values for series and parallel devices, and the comparison of parameters. You can control both parts completely on a device-type basis.
Combining Parameters

To help with a comparison, the *permuteDevice* command reduces groups of devices of the same type to a single device. For example, to compare three resistors in series on the layout with two resistors in parallel on the schematic, both are reduced to single devices that can be compared by their type and connections. The program does this by reducing devices two at a time iteratively, and alternatively series and parallel, until all series and parallel relationships have been reduced.

Every time LVS reduces a group of devices to a single device, you must combine the parameters for each device to produce a single set of parameters. This enables the parameters to be correctly compared after device matching. The consolidation process is device-, configuration-, technology-, and application-dependent, so you must specify it directly on a device and configuration basis. You cannot combine devices processed by the *MOS* option of the *permuteDevice* command.

*combineFunction Option*

To consolidate, you can use the *combineFunction* option of the *permuteDevice* command. This option specifies the name of a SKILL routine that combines the parameters of reduced devices. The routine uses two lists of parameters, one for each device. Each list contains all the parameters for that device. The routine extracts the parameters required for consolidation and combines them. The result, a new list of parameters, is passed back to the LVS program.

The list of parameters you pass back can be the resultant parameters only or a combination of the new ones plus uncombined originals. For example, if the program is consolidating two resistors in series, it uses a SKILL function that has the list of parameters for each resistor. The SKILL function might, in this case, select the R parameter and set $R_{new} = R_1 + R_2$. It then returns $R_{new}$ to the program as the new value of $R$ to be attached to the consolidated device.

The following shows example input:

```
permuteDevice( parallel resistor resParCombine )
permuteDevice( series resistor resSerCombine )
permuteDevice( parallel nfet nfetParCombine )
```

Details of the *permuteDevice* command format can be found in the “*permuteDevice*” section.
Example

Just as devices are combined two at a time, their parameters can be combined two at a time. The following example illustrates the process.

\[ \begin{array}{c}
A & B & C \\
Pa & Pb & Pc
\end{array} \xrightarrow{f(Pa,Pb)} \begin{array}{c}
X & C \\
f(Pa,Pb) & Pc
\end{array} \xrightarrow{f(Pa,Pb,Pc)} \begin{array}{c}
Y
\end{array} \]

If you have three devices A, B, and C in series, A and B are combined to form X, and then X is combined with C to produce Y. The parameters of A and B (Pa and Pb) are passed to the series device parameter consolidation function, which can do whatever it wants with them, returning a set of parameters that are stored with X. The parameters of X and C (Pc) are then passed to the function which generates the parameters of Y.

This accumulation of parameters works only if the function returns parameters as properties with the same names that were passed to it. For example, if the properties on a device are W and L, the result of the function should generate W and L. To obtain consistent results, the function should also be designed to be independent of the order in which the devices are combined.

Parameter Comparison

After devices on the schematic are matched with devices on the layout, you can compare their parameters. LVS compares the parameters of the actual devices matched. If these are the result of series or parallel consolidation, then LVS compares the combined parameter. However, when you want to display devices with mismatched parameters, this product highlights the original devices.

To compare the parameters, LVS uses a SKILL routine that you provide. You must give a separate routine for each device type you want to compare.

Use the LVS command `compareDeviceProperty` to define which devices you want to compare and what comparison routine you want to use. Refer to the “compareDeviceProperty” section for more information on this command. The `compareDeviceProperty` command works as follows:

- Identifies the devices to be compared.
- Identifies the comparison routine to use.
- Passes a parameter string from each device to the comparison routine. Each list contains all the parameters for that device.
If a device does not have parameters, LVS passes a null string. In the case of the schematic, LVS passes the default string associated with the device type (see netlist format in Appendix D, “Simulation and Environment Control”) if it exists; otherwise, LVS passes a null string.

- The routine compares the lists and returns a true or false switch, where t (true) indicates that the comparison has failed. You can replace the true switch with a message string that can be used by `Explain Error` in the LVS form.

  When the routine returns a message string, you must limit the length of the string to less than 32,000 characters.

- If the comparison fails, LVS stores the devices-in-error in the `audit.out` error file. `Display Errors` in the LVS form uses this file to highlight errors on the schematic and layout.

  If LVS is comparing combined devices, it compares the combined parameters, and updates the error file with references to all the original devices forming the consolidation.

You can view the devices in error graphically by selecting the Display Errors `UnMatched parameters` option in the LVS Error Display form. This option highlights those devices. You can determine the corresponding device in the alternate cellview (schematic or extracted) by using the cross-probing capability.

### Parameters for Ambiguity Resolution

Normally, LVS matches circuitry using only the device types and their interconnections (LVS verifies on circuit connectivity). When an ambiguous area of circuitry is encountered, the program has a choice in what can be matched. This approach might be acceptable for the connectivity, but it can result in mismatched device parameters where the ambiguous branches of the circuit actually contain different device parameters.

The ideal situation is for LVS to take into account the device parameters in its matching algorithm when ambiguous circuits are encountered. However, the parameters should not be used when matching straight connectivity.

Since the program has no way of knowing which device parameters to use to aid the verification, users must provide feedback to the program in terms of the parameters to be considered and their effect. For example, users must specify whether parameters should be within a certain percentage of each other or whether they should be identical to form a match.

To resolve this issue, LVS lets you define a command that instructs it to use parameters as part of the LVS signature process. The form of the command is

```plaintext
parameterMatchType( "device" function )
```
Explanations

The device is the name of the device type whose parameters are to be used. The function is the name of a SKILL function that provides feedback to this verification product on parameter information.

The main matching capability in LVS is based on signature analysis. For ambiguous areas of a circuit, the program needs to add the device parameters into its signature. However, the program does not know which devices to process and which parameters from all the possible parameters on a device to use. Also, the program cannot reasonably handle floating point parameter values, which is a critical situation. If one device parameter is 1.0 and another is 1.00000001, and the program takes these numbers literally, it would consider them to be mismatched. This can also be extended if you consider “close enough” to be correct. For example, if one parameter is 100 and the other parameter is 99, this might be considered a match.

The above command tells the program which devices it should look at and gives you control of the parameter selection and value interpretation.

**Note:** Do not confuse this command with the *compareDeviceProperty* command, which is used to compare device parameters after the devices are matched.

The program passes the parameter list from the device to your function. Your function must return a single integer value. This integer value can be derived any way that you want and normally represents the value of the key parameter or parameters being considered as critical to the matching. The returned value can be considered as a classification of the device into a subtype. Only devices of the same type and subtype are matched.

**Warning**

The program goes through two stages as follows:

1. It tries to match without considering parameters.

2. If it finds ambiguity and parameter matching is requested, the program adds the value returned from your function into each device signature. The program then tries to match again.

If the second matching is unsuccessful, the program does not return to the original mode for the ambiguous devices. The program does not assume that all ambiguous devices are equivalent and does not make a random matching. If this occurs, the ambiguous part of the circuit remains unmatched.

Be careful with the value returned from the function. If you are only considering device lengths of 2, 3 or 4, you can just use the length as the returned value. However, if you are considering
parasitic capacitance values as the matching criteria, you must be sure that the function
returns the same integer value for all capacitance values that you want matched. If you want
a tolerance on the matching of parasitic values, you should consider rounding, truncating, or
even multiplying the parasitic value by a factor before these operations.

The returned value should be considered a device subtype and not a value. Devices with the
same subtype are matched. The user's function creates a subtype from the device parameter
list.

Limitations

Using the parameters to aid matching only works where the parameters are known. This is
true for individual basic devices and for results of series and parallel reduction where the
user's parameter reduction function has been called.

It is not true for permutable MOS devices in AND and OR configurations. The program
reduces these groups of devices to higher-level structures and matches these. At this stage,
the individual parameters of the component devices are not available for controlling the
matching.

SKILL Routines Format

You must place the SKILL routines that perform parameter reduction and comparison in the
lvsRules function, before the various LVS control commands (permuteDevice,
ignoreTerminal, pruneDevice).

In the following examples, the comments clarify the application of the LVS program but do not
describe the basic SKILL structure or syntax. For information on SKILL refer to the Design
Framework II Help.

Both the permuteDevice and compareDeviceProperty SKILL routines process two parameter
lists. The first parameter list in compareDeviceProperty is always from the layout
representation, and the second list is from the schematic. The order is important in
comparisons where some check other than equality is required.

The contents of the routines can be as complex as you want, and can use any of the
parameters in the property strings associated with the devices. For example, you can
calculate the width and length resulting from combining two transistors using their original
widths and lengths plus parameters such as bend counts, plus any constants or factors you
want.
You can also use the `permuteDevice` function to determine whether series or parallel reduction should take place for the two devices being processed. This is achieved by the function returning a flag that tells the program not to perform the reduction.

**Example 1**

The following is an example of an SKILL function that performs a parameter combination in the `permuteDevice` command. The numbers at the beginning of the line are for reference in the comments only and do not appear in the final code.

```skil
1 procedure( parallelMOS( m1, m2 )
2 prog( ( mt )
3     mt = ncons( nil )
4     if( ( m1->L && m2->L )
5         mt->L = ( m1->L + m2->L ) / 2.0
6     )
7     if( ( m1->W && m2->W )
8         mt->W = m1->W + m2->W
9     )
10    return( mt )
11 )
```

**Line 1**

Defines the procedure name `parallelMOS` which is the name used for the function in the `permuteDevice` statement. It also defines two arguments, `m1` and `m2`, which it expects to receive when the LVS program calls the function. These two arguments are each lists of parameter properties and contain all the properties available on the devices being combined.

**Lines 2 and 3**

Set up a local variable as a list (initially a null), which contains your resulting parameter list for the combined device.

**Line 4**

Checks to see if both the `m1` and `m2` lists contain a reference to a parameter called “L.” If they do, line 5 calculates a resulting value as (L1+L2)/2 and stores it in the new list “mt” with the parameter name of “L.”

**Line 7**

Checks to see if both the `m1` and `m2` lists contain a reference to a parameter called `W`. If they do, line 8 calculates a resulting value as W1+W2 and stores it in the new list `mt` with the parameter name of `W`. 
Example 2

The following is an example of a compareDevice SKILL routine which compares two parameters and returns an error status if the parameters do not conform to the required limits.

```plaintext
1  procedure( compareCap( m1, m2 )
2    prog( )
3      if( ( m1->C && m2->C )
4        if( abs( m1->C - m2->C ) > 0.01 * m2->C
5          return( t )
6      )
7    )
8    return( nil )
9  )
10 )
```

Line 1 Defines the procedure name compareCap which is the name used for the function in the compareDeviceProperty statement. It also defines two arguments, m1 and m2, which it expects to receive when the LVS program calls the function. These two arguments are each lists of parameter properties and contain all the properties available on the devices being compared. The first (m1) contains the properties for the layout (extracted) device, and the second (m2) contains the properties for the schematic device.

Line 3 Checks to ensure that both lists contain the parameter to be compared, in this case C.

Line 4 Compares the parameters and, in this case, tests to see if the extracted C is within 1% of the schematic C. If it is not, the function returns t (true) which indicates to the LVS program that an error must be generated. If it is, within 1% the function returns nil (false) which indicates that no error has occurred, in which case the LVS program takes no further action.

Line 8 Returns the result, in this case, the value nil. To indicate an error, it might return the value t or a text string. This text string is associated with the error in the database and displayed when you select Explain Error on the Error Display form, which you access from the LVS form.
Two ways of creating a text string are as follows:

```c
return ( "Parameter mismatch on MOS devices" )
```

This text string is fixed for the compare routine and does not contain parameter values.

```c
sprintf( x , "Param mismatch: %g to %g" ,
float( m1->C ) , float( m2->C ) )
return( x )
```

This text string contains the values of the parameters compared.

**Note:** These constructed messages should not exceed more than 32,000 characters in length.

**Device Fixing**

The *Apply Device Fixing* option in the LVS form lets you control the permutability of devices on an instance-by-instance basis. Permutability normally applies to all individual devices of the type you specify. There are times, however, when you might not want global permutability. You can have the majority of your circuit obey the global permutability requests but force selected individual devices to be compared without permutability.

Some specific gates within a circuit are valid only if they are formed on the layout with exactly the individual transistor relationships defined in the schematic. Alternatively, some gates might function correctly only if specific devices in them maintain positions relative to others. For example, the clocking device in an NMOS NAND gate might need to be directly connected to ground while the other devices might be interchangeable.

You can fix series, parallel, and MOS permutability. A fixed device does not take part in device permutability processing and must, therefore, have direct correspondence between the layout and schematic.

**Prerequisites**

Device fixing has the following requirements:

- You must define the devices to be fixed in the schematics cellview.
- You must set a global switch to turn device fixing on.

Only one of the circuit representations (schematic or layout) has to be fixed since the other must conform to it; fixing both might lead to unresolved conflicts. We choose the
schematic as the representation to be fixed since LVS considers it the master to which the layout must conform.

You must manually add control information to the netlist if you run LVS on a netlist generated in some way other than through the netlister. The netlisting program translates any fixing information provided on the schematic into specific control lines in the netlist. If the netlist is not generated by the system netlister, it does not have these lines.

The fix Property

You can fix devices in the schematic by adding a property to device instances. The property has the following form:

```
fix = " terminalNameList "
```

*fix* is the name of the property. It must be of type *nlpExpr*.

*terminalNameList* contains the names of one or more terminals of the device instance to which you want to attach the property.

The device instance to which you attached the property can be a single low-level device such as a transistor, a higher level device such as a gate, or a high-level structure such as a counter. You cannot attach the property to the complete circuit.

All of the lowest level devices within the instance that are connected to the named terminal are considered fixed. For example, if in a CMOS NAND gate, you reference the terminal “*in1*” in a *fix* property, both the *n* and *p* devices gated by *in1* are fixed.

For a higher level device, such as a counter stage containing gates, which in turn contain transistors, the fixing applies to all transistors connecting to the named terminals, throughout the hierarchy. If the fixed instance is contained within another level of hierarchy, all placements of that higher level instance contain fixed lower level instances.

Correspondence Points

The Correspondence Points field lets you define correspondence points to aid the LVS processing. You can use terminals as correspondence points and/or provide a correspondence point file.

A correspondence point is a predefined matching of a single location in the layout and schematic that you supply and that is available to the program to help in its comparison. Normally in the LVS program correspondence points are not required; but in some cases of unresolved ambiguity, they help the program make a decision.
If you supply a correspondence point between two items that do not actually match, the program might immediately recognize the discrepancy and ignore the point; or it might initially accept the point and propagate errors to the surrounding circuit. The latter occurs if the incorrectly labeled nets have the same number of connections.

You can define correspondence points in two ways. Both can be applied at the same time.

- **By using terminal names**
  
  The program can use any terminal (pin) names on the schematic and extracted layout as correspondence points. It automatically transfers the names from the schematic or extracted representation to the netlist as `t` statements.
  
  If you choose the *terminals* option in the LVS form, these statements are used as correspondence points. If a terminal name on the layout corresponds to a terminal name on the schematic, then LVS assumes they refer to the same net. This feature does not apply to net names.

- **By creating a correspondence point file**
  
  You can specify your own correspondence points by creating correspondence point files using the LVS Correspondence Points form.

Because the application of incorrect correspondence points can mislead the program, first try running the program with none defined. If ambiguity causes the comparison to fail, you can apply circuit terminal names. If the run without terminal correspondence compares correctly, check your labeling by probing your terminals to confirm that the names used correspond to the names of the nets that the LVS program has matched.

### Creating a Correspondence Points File Using a Form

You can create correspondence points using the Correspondence Points form, which you access by selecting *Create* in the LVS form. You select pairs of items, one each from the extracted and schematic cellviews, to be stored in a correspondence points file. It is a standard text file containing net and device names. You can create any number of files with different names and then specify which one you want in the LVS form.

When using the Correspondence Points form, you must use the extracted and schematic cellviews you defined in the LVS form. Each cellview must be open in a window. You can be at any level of hierarchy within the cell.

Correspondence point files are a normal part of your computer file system and are not changed or deleted by running LVS. A temporary `lvs.cpoint` file is left in your run directory after the run. This contains the correspondence point file data, with net and device numbers instead of names.
The name `lvs.cpoint` is a reserved name and should not be designated as a user-defined correspondence file name.

**Note:** If you do not put the correspondence points file in the current working directory, you need to provide either a full path to the correspondence points file, or put the LVS run directory in your SKILL search path list.

**Creating a Correspondence Points File Manually**

You can also create correspondence point files by manually editing a file. You must know the names of the devices and nets you want matched. To edit the correspondence file manually, you need to understand that:

- Each line of the file describes a single correspondence point
- Each line has one of the following formats:
  ```plaintext
  ( #net_path_name ) ( #net_path_name )
  ( $device_path_name ) ( $device_path_name )
  ```

  The left-hand name refers to the layout, and the right-hand name refers to the schematic.

  The “#” and “$” define nets and devices, respectively. There must be no space between the character and the name.

  There must be one or more spaces between the opening and closing parenthesis and the names, and one or more spaces between the layout and schematic definitions. The number of spaces before the layout definition is not significant.

**Network Terminal Processing**

LVS uses network terminals (as distinct from device terminals) in Network Terminal Processing in several ways.

- Terminals are created by placing pins - not labels - in your layout view. During the extraction process, the Diva verification product attaches the pin name information to the net in the resulting extracted view. This product in turn uses the pin name information from the extracted view as terminals during an LVS comparison.

  **Note:** Dracula makes use of labels as terminals, not pins. If you trying to verify a design in this verification tool that was previously verified in Dracula, you must also place pins on the nets in the layout where you originally used labels.
Terminal names automatically give additional information. The program generates error messages for terminals that do not match. See the “Unmatched Terminals” section for more information.

If a terminal name exists in both cellviews, devices cannot be permuted across that terminal. No series, parallel, or MOS permutation absorbs a terminal.

If you select the terminals option in the LVS form, LVS treats all terminals with the same name in both cellviews as correspondence points. This forced matching of the terminals does not force the nets to which they are attached to be matched if they do not actually match. However, forced matching does help to resolve ambiguities.
Comparing Macro Cell

The LVS program works on all device types, regardless of whether they are low-level transistors or high-level macro blocks. The comparison works from two netlists, each derived from a different representation of the circuit. In the normal case, one is from the extracted layout and one from the extracted schematic. It is the control of the netlister that is special for macro cells.

The netlister normally defaults to the *spice* or *schematic* representation when encountering a device to translate. For macro cells, the default has to be overridden by use of the simulator control commands for the switching lists and stopping lists in the file `.simrc` in your home directory. (Refer to Appendix D, “Simulation and Environment Control” for information on the `.simrc` file.) These commands must reference the *layout* or *abstract* cellviews. The *spice* and *schematic* cellviews for the higher level cells do not normally contain connections to power and ground, whereas the *layout* and *abstract* cellviews do.

In addition, the cellview being referenced must contain the `NLPModelPreamble` and `NLPElementPostamble` properties that control the netlister output. If you put these on the *layout* cellview, you can automatically propagate them to the *abstract* cellview by using the `abgen` program.

When processing the schematic, the netlister follows its normal procedure and assumes global signals of the same name are connected. For the extracted cellview, however, this function is disabled so that true breaks in the global signals are not hidden.

With these changes in place, the processing of LVS remains unchanged. All its normal features and facilities are available.

Output Files

This tool puts all the errors it finds in error output files which it uses to display and explain errors.

The output files are stored in subdirectories of the run directory. When you use the LVS form, you must specify the run directory where you want to put your results. The program then creates the run directory (if one does not already exist) and the *layout* and *schematic* subdirectories. When you run an LVS comparison, the resulting error output files are put in the appropriate subdirectories.

The outputs from LVS fit into the following categories:

- Error files that you can read to find the name of the net, device, or terminal causing an error
Diva Reference
Comparing Layout to Schematic (iLVS)

■ Working and information files that you can display by using menu commands
■ Control and cross-reference files that you cannot access
■ A copy of the rules file under the name divaLVS.rul

Error Files

A series of text files containing the error output of the LVS program is stored in the layout and schematic subdirectories of the run directory.

This tool highlights the nets and devices referenced in these error files when you select the appropriate entry in the LVS Error Display form. However, with a complex design you might want to display only one net or device at a time. You can do this by using the *Probe* command and entering the net or device name.

One way of finding the net or device names is to look them up in the error files. Each file entry consists of two lines. The first gives the name of the object in error and indicates if it is a net (N), device (I), or terminal (T). The second line is a comment line explaining the error.

    N <netName> [comment]
    <explanation>
    I <deviceName> [comment]
    <explanation>
    T <terminalName> [comment]
    <explanation>

The error file names and their contents are as follows:

- netbad.out  Lists all unmatched nets
- devbad.out  Lists all unmatched devices
- prunenet.out Lists all ignored nets
- prunedev.out Lists all ignored devices
- mergenet.out Lists all merged nets
- termbad.out Lists all unmatched terminals
- audit.out Lists all unmatched parameters

Working and Information Files

The following files contain information that you can display using the Show Run Info form, which you access from the LVS form.
### Diva Reference

**Comparing Layout to Schematic (iLVS)**

<table>
<thead>
<tr>
<th>File Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>schematic/netlist</td>
<td>Text file containing the netlist of the schematic circuit. View this file by selecting the Schematic Netlist option in the Show Run Info form.</td>
</tr>
<tr>
<td>layout/netlist</td>
<td>Text file containing the netlist of the extracted circuit. View this file by selecting the Layout Netlist option in the Show Run Info form.</td>
</tr>
<tr>
<td>si.out</td>
<td>Text file containing all messages and warnings output from the program. View this file by selecting the Output option in the Show Run Info form.</td>
</tr>
<tr>
<td>si.log</td>
<td>Text file containing information on the program execution and its environment. It also includes summary information of the results of the functions you requested. View this file by selecting the Log option in the Show Run Info form.</td>
</tr>
</tbody>
</table>
Displaying Errors

When LVS has completed running, you can display the errors or cross reference the matching nets and devices. All error display is in the form of probes (highlighting).

You can display errors by selecting Error Display in the LVS form. You also can manipulate displayed errors using the Probe command from the Verify menu. By using the LVS output error files, you can identify specific nets or devices that you want to probe. You can find a description of the output error files in the Output Files section. You can probe nets and devices using the Probe command from the Verify menu.

The following sections discuss the display options available in the LVS Error Display form.

Unmatched Nets

The nets option under the Unmatched heading highlights all nets that failed to match in LVS. The number of connections made to the net is displayed when you use select Display Errors. Unmatched nets do not include merged nets.

Merged Nets

A second class of unmatched nets is the merged nets. In some cases, LVS recognizes that two nets need to be “shorted” together to improve the comparison. This occurs when many of the instances attached to the nets have already been matched. Explain Error in the LVS Error Display form lists the names of the two merged nets. LVS highlights not only the error net, but also the net with which it was merged.

When two nets need to be merged in the extracted representation, it indicates the possibility of an open. When two nets are merged in the schematic representation, it indicates the possibility of a short in the layout.

Unmatched Instances

For the selected window, LVS highlights all instances (devices) that appear in that network and could not be matched with instances in the alternate network. For schematic device instances where the device displayed is not the lowest level in the hierarchy (for example, a transistor is in error, but the display shows the logic gate that contains it), the complete higher level device is highlighted. You can use the Push View command in the Hierarchy menu to display the individual devices. The mismatched devices are highlighted and those that match are not.
Unmatched Parameters

For the selected window, LVS highlights all instances (devices) that appear in that network and whose parameters violated the requested comparison with instances in the alternate network.

LVS compares parameters only when the instances match in the two representations. Therefore, you can find the corresponding instance for any parameter mismatch in the alternate representation by using the Probe command.

Any instances that did not match and appear in the unmatched instances display have not had their parameters compared.

For devices that have been combined in series or parallel and whose parameters have been combined by the function supplied to the permuteDevice command, the parameter comparison is made on the resulting values. When there is a mismatch, LVS highlights all the devices involved.

Unmatched Terminals

The Unmatched terminals option in the LVS Error Display form under the LVS form highlights terminals that failed to match. There are two kinds of mismatched terminals in this display.

The first are terminals at the top level of the design hierarchy. These terminals correspond to pins used to make connections to the rest of the design. Terminal checking is done after all matching has completed. Those terminals that are in both designs are checked to make sure they match each other. This is very important in Place and Route applications where connections are made to the pins. If a nonglobal terminal exists in one netlist, but not in the other netlist, an error is flagged unless the terminal is not connected to any devices. If a global terminal’s name in one netlist does not match to another global terminal’s name in the other netlist, an error is flagged.

The second kind of terminal mismatch is internal to the design. Under certain special circumstances, LVS recognizes that a better match can be made by switching the terminal connection of an instance from one net to another. This is referred to as a rewired terminal. For this to occur, many of the instances and nets in the local region must have already matched. This kind of error occurs when two nets are cross-wired. The LVS program interchanges the connection and continues processing.

LVS does terminal rewiring only in the extracted database. The schematic database is taken as the master.

The error message produced by a rewired terminal is very explicit. For example, it might say, “Terminal ‘gate’ of /m25 should be connected to
/clock2 instead of /clock1." In this case, LVS connects /clock2 to the gate and continues running. This makes cross probing nets somewhat confusing. If you try to cross probe the gate of /m25, it selects net /clock1 and cross probes it. However, because LVS rewired the gate internally, it believes /clock2 is the gate net. Therefore, to cross probe the gate, first probe /clock2 by name to highlight it and then cross probe it.

Rewired terminals often occur in pairs. By inspecting the pairs, you can see which nets have been cross wired.

There is usually more than one way to repair a cross-wired situation. For example, even in the simple example of a crossed pair, you can correct the error at either end of the nets. LVS might not find the terminal switching that is most appropriate for the your design. You must decide on the best way to correct the design.

**Pruned Nets**

This is a display of all the nets that were removed from the LVS processing with the command pruneDevice.

**Pruned Instances**

This is a display of all devices that were removed from LVS processing with the pruneDevice command. This file also contains the devices removed by the removeDevice command. The removed devices are differentiated from the pruned devices by the associated message that displays when you query the device highlight.

**LVS Verification Rules Commands**
**compareDeviceProperty**

```javascript
compareDeviceProperty( device compareFunction )
```

**Description**

Defines the name of the SKILL function used to compare properties of devices matched in the layout and the schematic. It applies to the device type specified.

**Fields**

- **device**: Name of the device type to be compared. It consists of a character string enclosed in quotes.

- **compareFunction**: SKILL function name. This function compares the parameters of pairs of devices that are matched in the layout and schematic. Parameter values are stored on devices as properties.

The function can return the following values:

- The value `nil` indicates the parameters match.
- The value `t` indicates the parameters do not match.
- A text string indicates the parameters do not match. The string is used as a label for the mismatched flag.

**Note**: The string must not exceed 32,000 characters in length.

**Examples**

The following example calls a SKILL function that compares parameters of all `nfet` devices.

```javascript
compareDeviceProperty( "nfet" nfetCompare )
```

The following example calls a SKILL function that compares parameters of all `capacitor` devices.

```javascript
compareDeviceProperty( "capacitor" capacCompare )
```
**ignoreTerminal**

`ignoreTerminal( device terminal )`

**Description**

Specifies which terminal types on which device types to ignore during the comparison.

Some comparisons might need to totally ignore a device terminal. A good example of this is a back-gate in CMOS that is correctly connected in the schematic. In the layout, however, whether it is connected or not can depend upon the size of the cell being processed. A small cell might not contain the layer to which the back-gate should be connected. A larger layout might have the layer but not have it connected to the correct net. Only on the final composite is everything in place to correctly connect the back-gate. A layout to schematic comparison always fails in the first two cases unless the back-gate is ignored.

You must specify a separate `ignoreTerminal` command for each terminal you want to ignore.

**Fields**

- **device**
  - Name of the device type whose terminal is to be ignored. It consists of a character string enclosed in quotes.

- **terminal**
  - Name of the terminal type to be ignored. It consists of a character string enclosed in quotes.

**Examples**

The following example ignores all bulk terminals of `nfet` devices during comparison.

```plaintext
ignoreTerminal( "nfet" "B" )
```

The following example ignores all bulk terminals of `pfet` devices during comparison.

```plaintext
ignoreTerminal( "pfet" "B" )
```
parameterMatchType

parameterMatchType ( device Function )

Description

Defines the name of a SKILL function to be used to process parameters for a device when the program encounters ambiguous circuits. The program uses the result of the function to resolve matching of the ambiguous circuits.

Fields

device

The name of the device for which parameter matching is to be used during ambiguous circuit matching.

compareFunction

The name of a SKILL function. The SKILL function is passed two arguments: the name of a device and a disembodied property list containing the parameters on the device. The function must return a fixed point integer. The integer is used to match devices with the same parameters. To return an integer, use the fix() function to return an integer so that there is no ambiguity from floating-point round-off.

If you are trying to match on MOS Width, for instance, multiply width by 1,000 before converting to integer in order to use three decimal places of accuracy. For instance, if you had 0.0012 and 0.0017, multiply by each by 1,000 to give you 1.2 and 1.7. When you convert to integer, both numbers become one.

Do not multiply by a very large number since that effectively asks for infinite accuracy and will have exactly the same problem as a floating-point number does.

Examples

The following example shows the definition of the SKILL function followed by the use of that function in the parameterMatchType command.

```plaintext
procedure( fetWidthDiff( deviceName dpl )
    if( dpl->w then
        fix( 1000 * dpl->w )
```
else
  0
)
)
)
....
parameterMatchType( "nfet" fetWidthDiff )
parameterMatchType( "pfet" fetWidthDiff )
permuteDevice

permuteDevice( type "device" [term1] [term2] [combineFunction] )

permuteDevice( MOS "device" )

Description

This command specifies which form of circuit simplification to perform for a specific device type. The simplification is based on the permutability characteristic of the device as derived from the device definition in the netlist.

You can specify multiple options for a single device by defining multiple permuteDevice commands. When you specify multiple options, the system applies all options iteratively as reduction takes place.
Arguments

defines the type of simplification that can be performed. The options are as follows:

**type**

- **series**
  Merges two or three terminal devices connected in series into a single device. Applies to all two-terminal devices. If
  the terminals are permutable, the reduction occurs regardless of the order of terminal connections. If the
  terminals are not permutable, the device is considered polarized and unlike terminals must be connected for series
  reduction to be performed.

  Applies to 3 terminal devices only if the first two terminals are permutable and the third terminal of each device is
  connected to a common net.

- **seriesN**
  Merges multiple terminal devices connected in series into a single device. The terminals to be connected in series are
  defined by the terminal name parameters term1 and term2. For series reduction to take place, all other terminals on
  each device must be connected to the same terminal on the other device.

  If the specified terminals are permutable, the reduction occurs regardless of the order of terminal connections. If
  the terminals are not permutable, the device is considered polarized and unlike terminals must be connected for series
  reduction to be performed.

- **parallel**
  Merges devices connected in parallel into a single device. You can specify all kinds of devices.

- **MOS**
  Combines MOS multiple-transistor configurations into single gate-function devices with permutable inputs. You can
  specify only MOS devices.

- **device**
  Name of the device type to which the reduction is to apply.

- **combineFunction**
  SKILL function name. This function combines the parameters of pairs of devices when they are combined into single devices. Parameter values are stored on devices as properties.

- **term1, term2**
  Names of the terminals building the seriesN relationship.

You need different *permuteDevice* commands for each of the options: to apply parameter consolidation, provide a separate combine function name for each command having series
or parallel options. These combine function references can be for the same or for different functions.

This parameter-combining function does not apply for MOS grouping.

The function returns the values described below. If you do not provide a function or the devices do not have parameters, parameter consolidation cannot take place although the devices are still combined.

■ A disembodied property list containing the resultant parameters that is added to the resultant combined device.

■ A text string of the value doNotCombine, which tells the program to not combine the two devices in series or parallel.

■ A symbol of the value doNotCombine, which tells the program to not combine the two devices in series or parallel.

Examples

The following example reduces resistor devices in parallel before comparison.

    permuteDevice( parallel "resistor" )

The following example reduces capacitor devices in series before comparison.

    permuteDevice( series "capacitor" )

The following example reduces capacitor devices in parallel and calls a SKILL function to combine the device’s parameters before comparison.

    permuteDevice( parallel "capacitor" parCapMerge )

The following example reduces resistor devices in series and calls a SKILL function to combine the device’s parameters before comparison.

    permuteDevice( series "resistor" serResMerge )

The following example reduces nfet devices in AND or OR configurations and swaps input terminals during comparison.

    permuteDevice( MOS "nfet" )

The following example illustrates the series reduction of a multiterminal device.

    permuteDevice( seriesN "res4" "anode" "cathode" res4merge )
pruneDevice

pruneDevice( type device terminal )

Description

Ignores inactive devices in the circuit. It functions as if the devices were totally removed or pruned from the circuit.

In some designs or design methodologies, such as gate array, the layout contains unused devices that have no equivalent in the schematic. These unused devices can cause a mismatch between the layout and schematic, and possibly mask or confuse real errors. If these devices are electrically inactive, they can be safely ignored.

The pruning process is iterative and takes place during reduction. The removal of one inactive device can make another device inactive, which in turn is pruned.

Fields

type

Defines the type of pruning to be performed. The options are as follows:

general

Prunes devices connected to a single net.

MOS

Prunes MOS devices whose connectivity guarantees they cannot affect the function of the circuit. The conditions for this are that either their source or drain is not connected to any other device in the circuit, and the gate is not driven. A gate that is not driven can connect to other MOS gates but nowhere else.

Also prunes MOS devices with the source, drain, and gate all connected to the same net.

device

Name of the device type to which the pruning is to apply. The name consists of a character string enclosed in quotes or a number of separate strings, each enclosed in quotes with the list of strings enclosed in parentheses.
terminal

Name of the terminal type to be ignored during device pruning. The terminal name applies to all device types specified. If no device types have terminals with this name, the program continues and a warning message is issued.

If this terminal name is specified, the devices are pruned as if the terminal does not exist. The terminal is retained for all other processing.

Examples

In this example, during comparison all *nfet* devices that are configured in compliance with MOS mode are ignored.

```
pruneDevice( MOS "nfet" )
```

In this example, during comparison all *capacitor* devices that are configured in compliance with general mode are ignored.

```
pruneDevice( general "capacitor" )
```

In this example, during comparison all *nfet* and *pfet* devices that are configured in compliance with general mode are ignored.

```
pruneDevice( general ( "nfet" "pfet" ) )
```

In this example, terminal *B* for the three terminal resistors is ignored. For pruning, it is regarded as a two-terminal transistor.

```
pruneDevice ( general "res3" "B" )
```
removeDevice

removeDevice ( type [short(term-list) ...] [function])

Description

Removes specific devices from the circuit based on the device type and the function you can optionally specify, which operates on the device parameters.

With this command, you can specify whether the terminal nets are left open or shorted together.

Fields

<table>
<thead>
<tr>
<th>Field</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>type</td>
<td>Name of the device type, in double quotation marks, to be removed. Instances of this device are removed in both the layout and schematic (depending on the use of the function option).</td>
</tr>
<tr>
<td>short</td>
<td>An optional keyword that introduces a list of terminal names. All terminals in this list are shorted together when the device is removed. You can specify more than one short keyword and associated list. Terminals in each short list are shorted to other terminals in the same short list, but are not shorted to terminals in other short-lists. All remaining terminals of the device not in the short lists are left open when the device is removed. If you do not specify the short option, all terminals are left open when the device is removed.</td>
</tr>
<tr>
<td>function</td>
<td>An optional argument that defines the name of a SKILL function previously defined in the rules file. If you specify the function argument, this tool calls the function prior to removing the device. If the function returns t, the device is removed. If the function returns nil, the device is not removed. If you do not specify the function argument, all devices of specified type are removed. This product passes the list of parameters on the device to the function. The intent of the function is to analyze the list of</td>
</tr>
</tbody>
</table>
parameters and decide whether or not the device should be removed.

Examples

The following example is the simplest form of the command. The capacitors are removed and the connections are left open.

```plaintext
removeDevice( "cap" )
```

The following example removes the resistors and shorts their terminals together:

```plaintext
removeDevice( "resistor" short( "PLUS" "MINUS" ) )
```

The following example includes two separate terminal shorts and a function which operates on the device parameters. It removes the device only if the size parameter is equal to 1.

```plaintext
procedure( myFunct( plist )
  prog( ()
    if( (plist && plist->size == 1 )
      return( t )
    )
    return( nil )
  )
)
removeDevice( "xyz" short( "a" "b" ) short( "c" "d" ) myFunct )
```

Backannotation (lvsbx)

Description

This function performs backannotation of schematic instance and net names to the extracted view of the circuit generated by the Diva verification extraction.

Prerequisites

Before you specify this function, you must create an extracted version of the layout and run LVS between the extracted and schematic views.
Operation

Backannotation is performed by a stand-alone program. It reads instance and net names from the schematic and uses the files created by the LVS run to create a copy of the extracted view updated with the schematic names.

You can either create an annotated copy of the extracted view, or you can overwrite the original extracted view.

There is no difference in disk or virtual memory usage between creating a new extracted cellview and overwriting the original extracted cellview. Both operations create a new version of the cell in the library.

Execution

Use the following command in the UNIX environment to execute backannotation:

```
  lvsbx < run_dir > [ library cell view ]
```

You must run the `lvsbx` program in the directory you ran the graphics editor when you executed LVS.

You must specify the `run_dir` argument that defines the name of the run directory previously used for an LVS run. This verification product determines the schematic and extracted cellviews to be processed from the `si.env` file in the `run_dir` directory.

The `library`, `cell`, and `view` form a single optional argument that defines the name of the output cellview. You must specify all three of these options or the argument is not valid. If you specify this argument, the backannotated extracted cellview is written to the view specified. If you do not specify this argument, the original extracted cellview is overwritten.

Limitations

Only flat extraction is supported. This tool does not backannotate to a layout that has been extracted hierarchically.

Names of instances and nets that are hierarchical in the top level of the schematic are modified to create flat names in the extracted view. For example, this tool converts the schematic hierarchical name of `/I1/net2` to the flat name `|I1|net2` in the extracted view.

If you overwrite the original extracted cellview, the cross-probe capability from the original LVS run is lost because the LVS mapping files are now out-of-date. You can correct this only by re-executing the LVS run.
If you re-run LVS after backannotating schematic names, devices or nets with the same names might not match. In areas of ambiguity, it is still possible for devices or nets to be matched to their ambiguous counterpart rather than the intended device or net. The backannotated names are not used as part of the LVS matching criteria.
Verify Menu Commands

Introduction

This chapter describes each command on the Verify menu in alphabetical order. Each command description provides detailed information about the command and the forms associated with the command. Each command description also tells you how to use the command and lists the associated SKILL functions.

DRC Command

Checks a layout for design rule violations. DRC puts any errors found on the marker layer. You can highlight and display information about errors using the Markers commands.
DRC Form

**Checking Method** sets the type of circuit check you want to use.

- **flat** checks all shapes in the layout, regardless of the hierarchy.
- **hierarchical** checks shapes within each cell in the layout, using their hierarchical relationship and pattern recognition optimization.
- **hier w/o optimization** checks shapes within each cell of the layout in hierarchical relationship, but does not use pattern recognition optimization.

**Checking Limit** describes the part of the layout to be checked.

- **full** checks the entire layout.
- **incremental** checks the areas of the circuit that have changed since the last DRC check.
by area checks for errors within an area you define. After you select by area for the checking limit, you can type the coordinates in the Coordinate field or use the mouse to define a rectangular checking area.

Coordinate lists the coordinates of the area to be checked. After you select by area for the checking limit, you can type the coordinates in the Coordinate field or use the mouse to define a rectangular checking area.

When you type the coordinates in the Coordinate field, you enter a value for a single rectangle with two pairs of coordinates defining the bottom left and top right. You must type at least one space between each set of coordinates, for example

12599:98991 115682:194485

The checking area you define in the Coordinate field takes precedence over coordinates specified in a run-specific command file.

Sel by Cursor lets you select the area you want to check by clicking on it with the mouse.

Switch Names identifies the section of statements in the technology file that you want to use for a specific DRC or extraction to control the command stream. The technology file sections you enter in this field are addition to the sections normally used to control DRC or extraction. You must type one space between multiple switch names, for example

drc1 substrateck
Set Switches brings up a list box showing all the switches in the DRC/Extract rules. You can then select the needed switches and click OK in the list box. The switches you select are entered into the switch names field.

If you want a single switch, click on it. If you want multiple switches, hold the control key while clicking on the second and subsequent switches.

The default switches ?drc and ?extract are not shown. If you are running DRC, any switches contained in a ?extract branch of the rules file are not shown. If you are running extraction, any switches contained in a ?drc branch of the rules are not shown. The only exception to this is when the ?drc or ?extract switches are part of more complex switch expressions. All switches within both branches are then shown.

Run-Specific Command File specifies either a file in the current directory or a path to the file that contains verification commands relevant to a specific run of the program. For more information about the run-specific command file, see Chapter 4, “Writing Diva Verification Rules.”

Inclusion Limit is an integer for including or excluding cells in a DRC run. An inclusion limit tells DRC to check all cellviews with an ivIncludeValue property less than or equal to this number. The Diva verification tool assigns an inclusion property of zero to each cell unless you define ivIncludeValue as a cell property. The default value is 1,000. For more information about inclusion limits, see Chapter 2, “Including and Excluding Cells.”
Join Nets With Same Name causes each group of electrical nets having the same name to be considered as a single net. This option connects the nets in the extracted cellview only, not in the physical layout.

Echo Commands displays the technology file DRC rules in the CIW as they are processed. Echo is not active during hierarchical processing because the rules are executed multiple times.

Rules File specifies either the UNIX location of a rules file or, if you specify a rules library, the name of the rules file within the library.

Rules Library specifies the library from which the rules file is referenced. If you do not select this option, this verification product gets the rules from a UNIX location.

Machine specifies the machine on which to run the verification checks. This field appears only if you have a Diva remote verification license.

- local designates your local machine
- remote designates a remote machine

Remote Machine Name specifies the workstation or server that contains the Diva verification software and where the job is run. See the Chapter 2, “Remote Mode” for details on running Diva remote verification.

Using DRC

1. Select DRC from the Verify menu.
   The DRC form appears.
2. Select the checking method.
3. Select the checking limit.
4. If you select by area as the checking limit, you can type the coordinates for a check area, or you can draw a rectangle to define the area with the mouse.
   - To type in the coordinates, click in the Coordinate field and start typing.
To select the area by clicking on it with the mouse, click on *Sel by Cursor*, then click once in the bottom left corner and once in the upper right corner of the area you want to check.

The rectangle coordinates appear in the *Coordinates* field.

5. To process nets that have the same name as though they are connected, select *Join Nets With Same Name*.

6. To see the DRC rules in the CIW, select *Echo Commands*.

7. To control if-then-else statements in the command stream, either type the switch names (with a space between each name) or select the *set switches* button to bring up a list box from which you can select the required switches. Click on a switch from the list box to select it. To select multiple switches, hold the control key while clicking on additional switches.

8. To identify a file containing run-specific verification commands, type the name of the file for *Run-Specific Command File*.

9. To specify an inclusion limit, type an integer.

For the inclusion limit to take effect, you must have added an *ivIncludeValue* property to each cellview you want to exclude. For more information about inclusion limits, see Chapter 2, “Including and Excluding Cells.”

1. To specify the rules file, click in the *RulesFile* field and type the name of the file.

2. To read the rules from a library, select the *Rules Library* button and type the name of the library in the field.

3. If you have an Diva remote verification license and want to run DRC on a remote machine, follow these steps:
   - Click on *remote* for machine.
Type the name of the remote machine.

4. To run DRC on the local machine, click on local for machine.

5. Click on OK to start DRC.

You cannot perform any other functions while DRC is running.

If the Echo Commands option is turned on, you can check the progress of the job by viewing the DRC rules displayed in the CIW as they are executed.

To get information about DRC errors, use the Marker commands. See the Markers Command command descriptions for more information.

**Extract Command**

```
Verify ➞ Extract
```

Extracts devices and connectivity to prepare a layout for ERC and LVS verification checks. Extract puts any errors found on the marker layer. You can highlight and display information about errors using the Markers commands.

**Prerequisites**

Your technology file must contain extract rules.
Extractor Form

<table>
<thead>
<tr>
<th>Extract Method</th>
<th>☐ flat</th>
<th>☑ macro cell</th>
<th>☑ full hier</th>
<th>☑ incremental hier</th>
</tr>
</thead>
<tbody>
<tr>
<td>Join Nets With Same Name</td>
<td></td>
<td></td>
<td></td>
<td>☐ Yes</td>
</tr>
<tr>
<td>Switch Names</td>
<td></td>
<td></td>
<td></td>
<td>Set Switches</td>
</tr>
<tr>
<td>Run-Specific Command File</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Inclusion Limit</td>
<td>1000</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>View Names</td>
<td>Extracted</td>
<td></td>
<td>Excell</td>
<td></td>
</tr>
<tr>
<td>Rules File</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rules Library</td>
<td>☐</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Machine</td>
<td>☑ local</td>
<td>☑ remote</td>
<td>Machine</td>
<td></td>
</tr>
</tbody>
</table>

**Extract Method** sets the method for extracting connectivity.

- **flat** establishes an electrical network regardless of the nested cell hierarchy.
- **macro cell** establishes an electrical network from all layout cells except macro cells. The result is a single extracted cellview in which the macro cells appear as device instances.
- **full hier** extracts the complete circuit hierarchically.
- **incremental hier** extracts only those cells in the hierarchy that have changed since the last extraction.

**Join Nets With Same Name** causes each group of electrical nets having the same name to be considered as a single net. This option connects nets only in the extracted cellview, not in the physical layout.

**Echo Commands** displays the technology file extract rules in the CIW as they are processed. Echo is not active during hierarchical processing because the rules are executed multiple times.
Switch Names identifies the section of statements in the technology file that you want to use for a specific extraction run to control the command stream. The technology file sections you identify here are in addition to the sections normally used to control extraction.

Run-Specific Command File specifies either a file in the current directory or a path to the file that contains verification commands relevant to a specific design. For more information about the run-specific command file, see Chapter 4, “Writing Diva Verification Rules.”

Set Switches brings up a list box showing all the switches in the DRC/Extract rules. You can then select the needed switches and click OK in the listbox. The switches you select are entered into the switch names field.

If you want a single switch, click on it. If you want multiple switches, hold the control key while clicking on the second and subsequent switches.

The default switches ?drc and ?extract are not shown. If you are running DRC, any switches contained in a ?extract branch of the rules file are not shown. If you are running extraction, any switches contained in a ?drc branch of the rules are not shown. The only exception to this is when the ?drc or ?extract switches are part of more complex switch expressions. All switches within both branches are then shown.

Inclusion Limit is an integer for including or excluding cells in an extraction run. An inclusion limit tells the extractor to check all cellviews with an ivIncludeValue property less than or equal to this number. Diva verification assigns an inclusion property of zero to each cell unless you define an ivIncludeValue as a cell property. The default value is 1,000. For more information about inclusion limits, see Chapter 2, “Including and Excluding Cells.”
**View Names** lets you override the default extracted and excell cellview names:

- **Extracted** specifies the name of the extracted view. The default name is *extracted*.

- **Excell** specifies the name of the excell view. The default name is *excell*.

**Rules File** specifies either the UNIX location of a rules file or, if you specified a rules library, the name of the rules file within the library.

**Rules Library** specifies the library from which the rules file is referenced. If you do not select this option, Diva verification gets the rules from a UNIX location.

**Machine** specifies the machine on which to run the verification checks. This field appears only if you have a Diva remote verification license.

- **local** designates your local machine.

- **remote** designates a remote machine.

**Remote Machine Name** specifies the workstation or server that contains the Diva verification software and where the job is run. See Chapter 2, “Running Diva Verification” for details on running Diva remote verification.

### Using Extract

1. Select *Extract* from the Verify menu.

2. Select the method of extraction.

3. To process nets that have the same name as though they are connected, select *Join Nets With Same Name*.

4. To see the extraction rules in the CIW, select *Echo Commands*.

5. To control if-then-else statements in the command stream, either type the switch names (with a space between each name) or select the *set switches* button to display a list box from which you can select switches. Click on a switch from the list box to select it. To select multiple switches, hold down the control key while clicking on additional switches.

6. To identify a file containing run-specific verification commands, select the *Run-Specific Command File* option and type the name of the file in the text field. To inactivate the *Run-Specific Command File* option, deselect it.

7. To specify an inclusion limit, type an integer.
For the inclusion limit to take effect, you must add an `ivIncludeValue` property to each cellview you want to exclude. For more information about inclusion limits, see the Chapter 2, “Including and Excluding Cells.”

8. To specify the rules file, click in the `RulesFile` field and type the name of the file.

9. To read the rules from a library, select the `Rules Library` button and type the name of the library in the field.

10. If you have an Diva remote verification license and want to run Extract on a remote machine, follow these steps:
   - Click `remote` for machine.
   - Type the name of the remote machine.

11. To run Extract on the local machine, click `local` for machine.

12. Click OK to start the extraction.

**Note:** You cannot perform other functions while extraction is running.

If the Echo Commands option is turned on, you can check the progress of the job by viewing the extraction rules displayed in the CIW as they are executed.

This verification tool generates an extracted cellview from the layout and places it in the same cell as the layout cellview. This tool automatically saves the extracted cellview to disk when the cellview is not displayed in a window. Otherwise, this tool simply redisplays the new extracted cellview. If an extracted cellview already exists, the output of this command overwrites the previous version.

To highlight information about extraction errors, you use the Marker commands. See the *Markers Command* descriptions for more information.

### ERC Command

| Verify | ⇒ | ERC |

Checks an extracted cellview of a layout or schematic for electrical violations. ERC lets you highlight and display information about any errors found.
Prerequisites

You must have an extracted cellview. Use the Extract command to extract a layout and commands from the Schematics menu to extract a schematic.

ERC Form

Run Directory Name sets the full or relative path to your run directory. This verification tool creates the run directory you specify if the run directory does not exist.

Generate New Netlist creates a new netlist when the ERC program is running. A new netlist must be generated when you first run ERC or when you make changes to the cellview. When you set this option to on, you activate the Library Name, Cell Name, and View Name fields.

Library Name is the library where the cellview resides.

Cell Name is the input cell name.

View Name is the view name, either extracted or schematic.
Browse displays the Library Browser, which you can use to select the library, cell, and view you want. When you select a cellview using the Library Browser, this verification tool automatically enters the cellview information in the form.

Sel by Cursor lets you use the mouse to select the library, cell, and view by clicking in a cellview window.

Job Priority (0 to 20) lets you specify the precedence of your ERC job you are running with other background jobs. The highest priority is 0, and the lowest is 20.

Rules File specifies either the UNIX location of a rules file or, if you specified a rules library, the name of the rules file within the library.

Rules Library specifies the library from which the rules file is referenced. If you do not select this option, this verification product gets the rules from a UNIX location.

Machine specifies the machine on which to run the verification checks. This field appears only if you have an Diva remote verification license.

local designates your local machine.

remote designates a remote machine.

Remote Machine Name specifies the workstation or server that contains this verification software and where the job is run. See Chapter 2, “Remote Mode” for details on running Diva remote verification.

Run starts an ERC job that runs in the background. If you turned on the Generate New Netlist option, this verification tool generates a new netlist before running the ERC simulator.

Job Monitor displays the Analysis Job Monitor form, which lets you see information about the current status of verification jobs, including the start time, host, status, and priority. You can also suspend, remove, or change the priority of jobs in process. See the Analysis Job Monitor Form for information about this form.

Show Run Info displays the Show Run Information form, which lets you see the log file, output file, netlist file, and error report by using this form. See the “Show Run Information Form” section for information about this form.

Display Errors highlights the net and device errors that this tool identified in an ERC run on the current cellview. You must specify a run directory on the ERC form before you use this command.
Explain Error explains the ERC error you select with the mouse. You must have already displayed errors with the Display Errors command.

- on CIW displays the error information in the Command Interpreter Window.

- on text window displays the error information in a text window.

Remove All Errors unhighlights the net and device errors that this product identified in an ERC run on the current cellview.

Using ERC

When you set the ERC environment, the default cellview is the cellview from which you invoked ERC. You might want to select a cellview that is not on the screen. The following steps tell you how to select a cellview.

1. Select ERC from the Verify menu.
   The ERC form appears.

2. Type in the path to the run directory.

3. Click on Generate New Netlist.
   The name fields show the default values for the current cellview.

4. To select a different cellview, you can do one of the following:
   - If the cellview is already open, click on Sel by Cursor and then click on the cell.
   - If the cellview is not open, follow these steps:
     Click on Browse in the ERC form.
     Click on the library name and select Expand Cells from the Library Browser menu.
     Click on the cell name you want and select Expand Cellviews from the Library Browser menu.
     To find the most recent version of the cellview, click on the cellview and select Expand Versions.
     Click on the version you want and select Select from the Library Browser menu.

5. If you do not want to generate a new netlist for this cellview, turn off the Generate New Netlist option.
6. To set the processing priority, click and drag the number bar until the priority you want shows above the bar.

7. To specify the rules file, click in the *RulesFile* field and type the name of the file.

8. To read the rules from a library, select the *Rules Library* button and type the name of the library in the field.

9. If you have an Diva remote verification license and want to run ERC on a remote machine, follow these steps:
   - Click on *remote* for machine.
   - Type the name of the remote machine.

10. To run ERC on the local machine, click on *local* for machine.

11. Click *Run* in the ERC form to start the job.

   For more information about your job, select from the following options:
   - To display and control job execution status, click on *Job Monitor*. See the “Using Explain Error” section for detailed information.
   - To see the ERC run file information, click on *Show Run Info*. See the “Using Show Run Info” section for detailed information.
   - To display errors on the cellview, click on *Display Errors*.
   - To get more information about the error, click on *Explain Error*. See the “Using Show Run Info” section for detailed information.
   - To unhighlight the net and device errors, click on *Remove All Errors*.
   - To highlight and query devices or nets, use the *Probe* command from the Verify menu. See the *Probe* command description for more information.
Analysis Job Monitor Form

If you click on Job Monitor in the ERC form, the Analysis Job Monitor form appears.

Show Run Log displays the log file for the verification run you specify.

Set Priority lets you reset the processing priority for a job. The highest priority is 0, and the lowest is 20.

Kill terminates the selected job that is running.

Suspend temporarily discontinues a job that is running.

Continue restarts a suspended job that has finished running.

Remove Entry removes the selected job that has finished running.

Using Job Monitor

1. Select Job Monitor from the ERC form.

   ERC displays the Job Monitor form, which lists all active and completed jobs. You can keep this form visible to see changes in the status of completed or currently running jobs.

2. Click on the name of the job for which you want to take action.

3. To display the run log for the selected job, click on Show Run Log.

   This tool displays the log file.

4. To change the priority of the selected job, click on Set Priority.

   The Set Priority form appears.

   The highest job priority is 0, and the lowest is 20.

   To reset the processing priority, click and drag the number bar until the priority you want shows above the bar.
Click on OK in the Set Priority form. The Set Priority form appears.

5. To stop processing the selected job, click on Kill.
6. To temporarily suspend the processing of the selected job, click on Suspend.
7. To resume processing the selected suspended job, click on Continue.
8. To delete a job from the Job Monitor listing, click on Delete.

Show Run Information Form

If you click on Show Run Info in the ERC form, the Show Run Information form appears.

Show Run Information displays information about the ERC run directory.

- **Log** displays a running summary of the progress (the si.log) of the ERC run.
- **Output** displays the contents of the si.out file from the ERC run directory. The si.out file states the software version, tells whether the job has finished successfully, and lists warning messages. For example, the warning messages might notify you of missing power and ground terminals.
- **Netlist** displays the netlist from the ERC run directory. If you find warning messages about net names in the output file, you can use the Netlist option to show the netlist used for the job.
- **Errors** displays a text description of ERC errors.

Using Show Run Info

1. To see the run information for the ERC job, click Show Run Info on the ERC form.
2. Click Log, Output, Netlist, or Errors, then click Apply or OK to view the information file.
Using Explain Error

1. To see the reason for the error, select *Explain Error* from the Display Error form.

   This verification product prompts you to point at the ERC error you want to explain.

2. Click on the ERC error.

   This tool displays the object name, the object type, and the ERC errors associated with the error you selected.

LVS Command

Verify ➔ LVS

Compares two versions of a circuit. You can compare two layouts, two schematics, or a layout and a schematic. You can highlight and display information about any differences found.
### LVS Form

<table>
<thead>
<tr>
<th></th>
<th>Command</th>
<th>Help</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Run Directory</strong></td>
<td>LVS</td>
<td></td>
</tr>
<tr>
<td><strong>Create Netlist</strong></td>
<td>schematic</td>
<td>extracted</td>
</tr>
<tr>
<td>Library</td>
<td>master</td>
<td>tutorial</td>
</tr>
<tr>
<td>Cell</td>
<td>mux2</td>
<td>mux2</td>
</tr>
<tr>
<td>View</td>
<td>schematic</td>
<td>extracted</td>
</tr>
<tr>
<td></td>
<td>Browse</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Sel by Cursor</td>
<td>Browse</td>
</tr>
<tr>
<td><strong>Rules File</strong></td>
<td>divaLVS.rul</td>
<td></td>
</tr>
<tr>
<td><strong>Rules Library</strong></td>
<td>cellTechLib</td>
<td></td>
</tr>
<tr>
<td><strong>LVS Options</strong></td>
<td>Rewiring</td>
<td>Device Fixing</td>
</tr>
<tr>
<td></td>
<td>Create Cross Reference</td>
<td>Terminals</td>
</tr>
<tr>
<td><strong>Correspondence File</strong></td>
<td>lvs_corr_file</td>
<td>Create</td>
</tr>
<tr>
<td><strong>Priority</strong></td>
<td>20</td>
<td>Run</td>
</tr>
<tr>
<td>Run Local</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Run Output</td>
<td></td>
<td>Error Display</td>
</tr>
</tbody>
</table>
**Run Directory** defines the full or relative path to the run directory. If the directory you specify does not exist, LVS creates the directory. If the environment of the directory you select is different from that in the LVS run form, the following form is displayed.

![LVS Form Contents Different](image)

**Browse** selects the LVS Run Directory browser. It contains a *Selection* field that displays the current path. When *OK* is selected, the path in the selection field is displayed in the *Run Directory* field in the LVS run form. There is also a *Path* field that displays the current context.
A list box shows selectable directories and lowest level files. If necessary, you can manually enter a file specification into the Selection field, rather than traverse the hierarchy.

Create Netlist creates netlists for the schematic and extracted cellviews. The first time you run LVS, you must generate netlists for both cellviews. Any time you change cellviews, you must generate a new netlist for each cellview.

- schematic creates a netlist for the schematic cellview and enables or disables the schematic cellview column.

The schematic option defines a working schematic cellview. If you have a schematic cellview open, or if you use the defaults from the run directory, the library name, cell name, and view name are filled in automatically.

- Library is the library name for the schematic cellview.
- Cell is the cell name for the schematic cellview.
- View is the view name for the schematic cellview.
**Diva Reference**

*Verify Menu Commands*

**Browse** displays the Library Browser. When you select the library, cell name, and view name you want, this verification tool automatically enters the cellview information in the form.

**Sel by Cursor** fills in the rest of the fields with information from the extracted cellview window you select with the mouse.

- **Extracted** creates a netlist for the extracted cellview and enables or disables the *extracted* cellview column.

The *extracted* option defines a working extracted cellview. If you have an extracted cellview open, or if you use the defaults from the run directory, the library name, cell name, and view name fields are filled in automatically.

**Library** is the library name for the extracted cellview.

**Cell** is the cell name for the extracted cellview.

**View** is the view name for the extracted cellview.

**Browse** displays the Library Browser, which lets you select the library, cell name, and view name you want. When you select a cellview using the Library Browser, this verification tool automatically enters the cell view information in the form.

**Sel by Cursor** fills in the rest of the fields with information from the extracted cellview window you select with the mouse.

**Rules File** specifies either the UNIX location of a rules file or, if you specified a rules library, the name of the rules file within the library.

**Browse** selects the LVS Rules File browser. It contains a *Selection* field that displays the current path. When OK is selected, the path in the selection field is displayed in the *Rules Library* field in the LVS run form. There is also a *Path* field that displays the current context. A list box displays all selectable files. If necessary, you can manually enter a file specification into the *Selection* field, rather than traverse the hierarchy.
Note: When a file is selected, the Rules File browser simply checks for read access to the file.

Rules Library specifies the library from which the rules are referenced. If this option is not selected, this verification product gets the rules from a UNIX location.

LVS Options generates a cross reference text file, fix devices, and bypass errors.

- **Rewiring** changes the extracted and schematic network to bypass errors and continue processing. This verification product flags each change.

- **Device fixing** uses fix properties on device instances to limit device permutability.

- **Create Cross Reference** generates a text file containing a list of nets and devices found in both the layout and the schematic. The first number of a pair refers to the layout, and the second number refers to the schematic. Net numbers begin with an N and device numbers begin with an I.

- **Terminals** uses terminals at the top-level of the circuit as correspondence points.
Correspondence File uses the file displayed in the Correspondence File field.

Create displays the Correspondence Point form. This form lets you create a file containing the correspondence points for the LVS run. The default is the file name in the Correspondence File field. For information about this form, see the “Correspondence Points Form” section.

Priority (0 to 20) specifies the job priority of the LVS run. The highest priority is 0, and the lowest is 20.

Run specifies the machine on which to run the verification checks. This field appears only if you have an Diva remote verification license.

- local designates your local machine.
- remote designates a remote machine.

Run starts the LVS program.

Output displays the LVS output report. If none exists, this verification ool displays the Display Run Info form.

Error Display displays the LVS Error Display form. For a description of this form, see the Virtuoso Schematic Composer User Guide to learn about connectivity and naming conventions for inherited connections and how to add and edit net expressions in a schematic or symbol cellview.

Monitor displays the Analysis Job Monitor form, which displays information about the current status of verification jobs, including the start time, host, status, and priority. You can also suspend, remove, or change the priority of jobs in process. For information about this form, see the “Analysis Job Monitor Form” section.

Info shows the Display Run Information form, which displays the log file, output file, netlist file, and error report. For information about this form, see the “Display Run Information Form” section.

Note: If you use Artist, two buttons appear on the LVS form for extracting parasitics. If you use Design Framework II, you do not see these buttons.

Using LVS

1. Select LVS from the Verify menu.

   The LVS form appears.
If both extracted and schematic views are open, the LVS form is already filled out for both views. If one view is open, only part of the form is filled out.

2. Type in the run directory name.

   If this directory does not exist, this verification tool creates it.

   If this directory exists and the setup from an earlier run conflicts with the contents of the LVS form, this verification product displays a dialog box that lets you choose whether to use the run directory defaults or the LVS form information.

   To use the run directory defaults, click OK.

       This tool updates the LVS form to reflect the contents of the run directory.

   ❑ To use the LVS form information, click Cancel.

3. To choose a schematic if the schematic cellview column is grayed out, click on schematic in the Create Netlist field. If you do not want LVS to generate a new netlist for the schematic cellview, you can click schematic again.

You can define the schematic you want in one of the following ways:

   ❑ To point to the schematic you want, click Sel by Cursor.

       This tool prompts you to point at the window containing the schematic you want.

   ❑ To use the Library Browser to find the cellview you want, click Browse and select the library, cell, and view, you want.

       If you use either of these options, this tool automatically fills in the form with information about the cellview you selected.

   ❑ To enter the cellview information manually, type in the names.
4. To choose an extracted cellview, when the extracted cellview column is grayed out, click extracted in the Create Netlist field. If you do not want LVS to generate a new netlist for the extracted cellview, you can click extracted again.

You can define the extracted cellview you want in one of the following ways:

- To point to the extracted cellview, click Sel by Cursor.
- To use the Library Browser to find the cellview you want, click Browse.

If you use either of these two options, this tool automatically fills in the form with information about the cellview you selected.

- To enter the cellview information manually, type in the names.

5. To define the correspondence points file, first click the Correspondence File field option, then type a file name in the field. Click the Create button and the Correspondence Points form appears.

6. If you do not want to generate a new netlist for these cellviews, turn off the Create Netlist (schematic or extracted) option.

7. To set the processing priority, click and drag the number bar until the priority you want shows above the bar.

8. To specify the rules file, click in the RulesFile field and type the name of the file.

9. To have the rules read from a library, select the Rules Library button and type the name of the library in the field.

10. If you have an Diva remote verification license and want to run LVS on a remote machine, follow these steps:

- Select Remote for Run.
- Type the name of the remote machine.

11. Click Run in the LVS form to start an LVS job.

When the job is finished, a message is displayed in a dialog box.

For more information about your job, select from the following options:

- To display and control job execution status, click Monitor.
- To see the LVS run file information, click Info.
- To display errors on the cellview, click Error Display.
This tool updates the run directory to reflect the definition of the schematic and extracted cellviews and the options you selected.

To highlight and query devices or nets, use the *Probe* command from the Verify menu. See the *Probe* command description for more information.

If your system is running the Analog verification package, you can backannotate a schematic or simulate a schematic with parasitics after LVS has completed successfully.

**Conflicting Paths**

If the library search path set in the run directory and the system library search path set in Design Manager do not match, the Initialize Environment form appears. You use this form to choose a path.

**Initialize Environment Form**

<table>
<thead>
<tr>
<th>Current Library Path</th>
<th>Library Path from si.env</th>
<th>Use Library Path from si.env?</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>./cds/Main/red/cdaslib</td>
<td></td>
</tr>
</tbody>
</table>

**Current Library Path** shows the search path set in the Design Manager.

**Library Path from si.env** is the search path set in the run directory.

**Use Library Path from si.env?** selects which path to use.

- *uses the library path in the run directory.*
- *uses the current library path.*
Correspondence Points Form

If you click Create in the LVS form, the Correspondence Points form appears.

Working File identifies the file to use for correspondence points. This file contains the net and device correspondence points. If the file does not already exist, this tool creates it when you select correspondence points to add.

Add displays the Add Correspondence Points form.

Add to adds a net or device correspondence point to the working file.

- net adds a net correspondence point.
- device adds a device correspondence point.

Clear deletes the entries in the Name in Extracted View field or in the Name in Schematic View field.

Name in Extracted View is the name of the net or device in the extracted cellview. You can type in the name or use the mouse to select a net or device from the cellview.
Sel by Cursor fills in the Name in Extracted View field with the net or device name when you select the net or device with the mouse.

Name in Schematic View is the name of the net or device in the schematic cellview. You can type in the name or use the mouse to select a net or device from the cellview.

Sel by Cursor fills in the Name in Schematic View field with the net or device name when you select the net or device with the mouse.

Remove displays the Remove Correspondence Points form.

<table>
<thead>
<tr>
<th>Remove Correspondence Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>OK</td>
</tr>
</tbody>
</table>

Remove from removes the correspondence points from the working file.

- net removes a net correspondence point.
- device removes a device correspondence point.

Identify in View specifies the cellviews from which you want to delete correspondence points.

- extracted specifies the extracted cellview.
- schematic specifies the schematic cellview.

Name of Point is the name of the net or device to remove. You can type in the name or use the mouse to select a net or device.

Sel by Cursor fills in the Name of Point field with the net or device name when you select the net or device with the mouse.
**Display** displays the Display Specific Correspondence Points form.

![Display Specific Correspondence Points form](image)

**Display from** displays the corresponding net or device of a point in the extract or schematic view.

- **net** displays the corresponding net of a point.
- **device** displays the corresponding device of a point.

**Identify in View** displays correspondence points.

- **extracted** displays the correspondence points in the extracted cellview window.
- **schematic** displays the correspondence points in the schematic cellview window.

**Name of Point** is the name of the net or device to display. You can type in the name or use the mouse to select a net or device from the cellview.

**Sel by Cursor** fills in the **Name of Point** field with the net or device name when you select the net or device with the mouse.

**Display All Pts** displays all the correspondence points in the working file.

**Clear All Pts** clears all highlighted correspondence points.
LVS Error Display Form

If you click *Error Display* in the LVS form, the LVS Error Display form appears.

![LVS Error Display Form](image)

**Explain** explains the error you select. You must have already displayed the error with one of the Display buttons below.

**Clear Display** unhighlights the net and device errors that this tool identified in the LVS run on the current cellview.

**Probe Form brings up the Probing form.**

**Display** allows you to examine individual net and device errors that this tool identified in the LVS run on the current cellview.

- **First** displays the first error.
- **Next** goes to the next error.
- **Prev** goes back to the previous error.
- **Last** displays the last error.
- **All** shows you all net and device errors.

**Note:** To skip directly to a specific error, enter the number in the field to the right of the display buttons.

**Error Color** allows you to choose the error highlighting color.

- **Cycle Colors** sets the errors to alternate highlight colors.
Diva Reference
Verify Menu Commands

Auto-Zoom zooms the current window into the area surrounding the highlighted error.

Note: Auto-Zoom might not be able to handle netlisting configuration.

All activates all Display options.

Note: If you select All, the window is zoomed out to a fit position.

None deactivates all Display options.

Unmatched highlights the unmatched nets, instances, parameters, or terminals in the source window.

- nets highlights the unmatched nets.
- instances highlights the unmatched instances.
- parameters highlights the instances that match in the schematic and the layout, but whose parameters do not pass your parameter check.
- terminals highlights the unmatched terminals.

Pruned highlights inactive instances or nets, that are ignored by LVS.

- nets highlights the inactive nets.
- instances highlights the inactive instances.

Merged highlights nets that LVS merges because the nets have the same net name.

- nets highlights not only the net with the error but also the net with which it is merged when you use the First, Next, Prev, and Last buttons. The All button highlights only the error nets.

Note: Auto-zoom zooms only the error net.
Net Display Limit limits the displayed errors by the number of connections the net makes. This helps viewing by eliminating large shorted nets, making misconnections more visible.

Note: Some violations may not be visible while this feature is in use. See Affirma Analog Artist Simulation Reference Manual.

Analysis Job Monitor Form

If you click Monitor in the LVS form, the Analysis Job Monitor form appears.

Command displays a list of the following options:

Show Run Log displays the log file for the run you specify.

Set Priority resets the processing priority for a job. The highest priority is 0, and the lowest is 20.

Kill terminates the selected job that is running.

Suspend temporarily discontinues a job that is running.

Continue restarts a suspended job that has finished running.

Delete removes the selected job that has finished running.
Display Run Information Form

If you click Info in the LVS form, the Display Run Information form appears.

<table>
<thead>
<tr>
<th>Display Run Information</th>
</tr>
</thead>
<tbody>
<tr>
<td>OK</td>
</tr>
<tr>
<td>Run Info</td>
</tr>
<tr>
<td>Schematic</td>
</tr>
<tr>
<td>Audit</td>
</tr>
<tr>
<td>Extracted</td>
</tr>
<tr>
<td>Audit</td>
</tr>
</tbody>
</table>

Log File displays a running summary of progress (the si.log) from the run.

Output displays the contents of the si.out file from the run. The si.out file states the software version, tells whether the job has finished successfully, and lists warning messages.

Netlist displays the netlist of the Schematic and Extracted cellview.

To display an internal report file you can click on any of the following buttons in the Schematic field or the Extracted field: Bad Devices, Bad Nets, Bad Terminals, Audit, Merged Nets, Pruned Nets, Pruned Devices.

Note: Formats may differ from one version to the next.

Inherited Connections for LVS in Diva Verification

Inherited connections is an extension to the connectivity model that allows you to create global signals and override their names for selected branches of the design hierarchy. This flexibility allows you to use

- Multiple power supplies in a design
- Overridable substrate connections
- Parameterized power and ground symbols

To learn about using inherited connections and net expressions with various Cadence® tools in the design flow, refer to the Inherited Connections Flow Guide.
To learn about connectivity and naming conventions for inherited connections and how to add and edit net expressions in a schematic or symbol cellview, refer to the *Virtuoso Schematic Composer User Guide*.

**Probe Command**

![Verify Menu Commands](Verify Menu Commands)

Let's you highlight and display information about devices and nets in a layout following an ERC or LVS run. Net and device information is supplied by the extracted versions of cellviews you generate. When you probe a net or device, the object is highlighted and you can request information about it. Single-probing lets you highlight a net or device in a single cellview. Cross-probing lets you highlight a net or device in two different cellviews that have been compared using LVS.

**Prerequisites**

You must have run circuit extraction on the cellview you want to probe. For cross-probing, you must also have run LVS to match a selected net or device between extracted and schematic cellviews.

**Probing Form**

![Probing Form](Probing Form)
Probing Method determines whether you want to probe in a single cellview or cross-probe correlated cellviews.

- **single probe** highlights the net or device in one cellview.
- **cross probe** highlights the net or device in the extracted and schematic cellviews at the same time.

Probing Scope indicates the way you want to match nets for a device or devices for a net.

- **matched** highlights matched nets for a device or devices for a net.
- **unmatched** highlights unmatched nets for a device or devices for a net.
- **all** highlights both matched and unmatched nets for a device or devices for a net.

Probe Type indicates the type of net or device you want to probe.

- **device or net** highlights a device or net.
- **device** only highlights only the device.
- **net** only highlights only the net.

Add Device or Net highlights a single net or device.

Add Nets for Device highlights all nets connected to a device.

Add Devices for Net highlights all devices connected to a net.

Show Probe Info displays information about all nets and devices currently highlighted.

Remove Device or Net eliminates a specific probe from a cellview.

Remove Nets for Device eliminates a specific net probe for a device.

Remove Devices for Net eliminates a specific device probe for a net.

Remove All removes probes from the current cellview and the cross-probes, if any, from the alternate cellview.
Explain explains the probe you select with the mouse.

- **on CIW** explains the probe in the Command Interpreter Window.
- **on text window** explains the probe in a text window.

**Run Dir** specifies the name of the run directory used by the probing. You can type the name of the run directory used for ERC or LVS in this field. For both ERC and LVS, the probing program uses this directory to get information on the netlisting switching and stopping lists so the hierarchy is processed correctly. For LVS, the probing program also uses this directory to get data for cross probing and highlighting of matched and unmatched devices and instances.

**Using Add Device or Net**

1. Select *Add Device or Net* from the Probing form.

   The CIW prompts you to point to the device or net or enter the name.

2. Do one of the following:

   - Click on the device or net you want to add.

   **Note:** When selecting a net with the mouse, you can click on a location where more than one net exists. A list window opens containing the names of the nets at that location. Click on the one you want and click *OK* to select it.

   - In the CIW, type the name in string format (for example, "/trinv1"), then press the Return key.

3. Select the probing method.

4. Select the type of net or device you want highlighted.

**Using Add Nets for Device**

1. Select *Add Nets for Device* from the Probing form.

   The CIW prompts you to point to a device or enter the name.

2. Do one of the following:

   - Click on the device you want to add.

   - In the CIW, type the name in string format (for example, "/trinv1"), then press the Return key.
3. Select the probing method.

4. Select the probing scope.

   Note: If you select cross probe, you cannot also select unmatched.

Using Add Devices for Net

1. Select Add Devices for Net from the Probing form.

   The CIW prompts you to point to the net or enter the name.

2. Do one of the following:

   - Click on the net you want to add.

     Note: When selecting a net with the mouse, you can click on a location where more than one net exists. A list window opens containing the names of the nets at that location. Click on the one you want and click OK to select it.

   - In the CIW, type the name in string format (for example, “/gnd!”), then press the Return key.

3. Select the probing method.

   - To highlight the devices for a selected net in one cellview, select the probing scope.

     Note: If you select cross probe, you cannot also select unmatched.

Using Show Probe Info

1. Select Show Probe Information from the Probing form.

   The object type, object name, and the ERC or LVS errors (if any) associated with all probes in the cellview are displayed.

2. To close the view window

   - Click on File at the top of the view window.

   - Select Close Window from the File menu.

Using Remove Device or Net

1. Select Remove Device or Net from the Probing form.

   The CIW prompts you to point to the device or net or enter the name.
2. Do one of the following:
   - Click on the device or net you want to remove the probes from.
   - In the CIW, type the name in string format (for example, “/gnd!”), then press the Return key.

Using Remove Nets for Device
1. Select Remove Nets for Device from the Probing form.
   The CIW prompts you to point to the device or net or enter the name.
2. Do one of the following:
   - Click on the device or net you want to remove the probes from.
   - In the CIW, type the name in string format (for example, “/trinv1!”), then press the Return key.

Using Remove Devices for Net
1. Select Remove Devices for Net from the Probing form.
   The CIW prompts you to point to the net or enter the name.
2. Do one of the following:
   - Click on the net you want to remove the probes from.
   - In the CIW, type the name in string format (for example, “/gnd!”), then press the Return key.

Using Explain
You must highlight the net or device with Probing “add” options before you can request information about it.
1. Select Explain from the Probing form.
2. Select the net or device for which you want information.
   The object type, object name, and the ERC or LVS errors (if any) associated with the probe are displayed.
Remove All

Select Remove All from the Probing form.

All probes and cross-probes of devices and nets from the cellview are removed.

Markers Command

When the results of a verification command produce errors or warnings, the Virtuoso layout editor provides information about the errors through the Marker commands. These commands help you find errors and get information about them. These commands appear under the Verify menu.

- Verify – Marker – Explain displays the reason for the error or warning marker in a text window. For more information, see the “Explaining Markers” section in the Virtuoso Layout Editor User Guide.
Diva Reference
Verify Menu Commands

- **Verify – Marker – Find** searches for and highlights each error or warning marker. For more information, see the “Finding Markers” section in the *Virtuoso Layout Editor User Guide*.

After you get error information you need, you can delete the marker.

- **Verify – Marker – Delete** removes a specific marker. For more information, see the “Deleting Individual or Multiple Markers” section in the *Virtuoso Layout Editor User Guide*.

- **Verify – Marker – Delete All** removes all markers. For more information, see the “Deleting All Markers” section in the *Virtuoso Layout Editor User Guide*. 
Short Locator

Shorts

Short locator lets you locate shorts in a layout following an Extract run. You can save and restore runs part of the way through to spread the work over several sessions. You can also add or remove labels and zoom in to identify the short location.

To start short locator, select Verify – Shorts or type ivHiShorts in the CIW.

Prerequisites

Before you can locate a short, the following must be true:

- You must know the net in which the short exists. The short location program only handles one shorted net at a time, although that actual net may be the result of many intended nets shorted together.
- You must have an extracted view and the layout view it was generated from.

All connect layers must be available, with one exception. If a layer is only attached to a net at one point, as in the case of a tub or well, the absence of the layer does not affect the connectivity of the rest of the net; all shorts can still be isolated.

How Short Location Works

Short location uses the interconnect polygons and contacts that form the net. A shorted net results from two nets placed so close together that they touch or from a mistake in labeling the nets. Short Locator analyzes the shorted net using all labels associated with the net. It assumes that the section of the shorted net under the label is part of the net defined by that label. Using these labels, the program determines which parts of the shorted net belong to one net, and which belong to another. Any part of the net that could belong to more than one label contains a short.
Short Locator attempts to isolate the short based on the labels available in the graphics data. It knows which labels apply to which layers (from the `geomConnect` rule). If these labels do not enable the program to isolate the short to a small enough section of the net, you can add more labels or delete misleading labels. When you do this, you do not affect the original graphics data; you only affect the Short Locator database.

**Placing Labels**

The first consideration in placing a label is knowing which part of the shorted net is really one of the original nets. Pads are obvious places to put labels. Also, if you know the circuit well, you can label the part of a net with long runs.

The second consideration is separation of labels. The farther apart the labels, the more information they supply. This is not a measurement of physical distance across a circuit, but of logical distance along the net. You can place labels physically next to each other providing they are at opposite ends of the shorted net.

You do not have to place labels on the part of the net that has been isolated as containing the short. You can place them anywhere on the net. In some cases, it is better to place the labels on a part of the net already discarded as not part of the short.

Consider the following example.

```
A    B
|    |
A    B
```

The labels A and B on the two shorted nets allow the program to isolate the short to the path between A and B as follows.

```
A    B
|    |
```

The labels A and B on the two shorted nets allow the program to isolate the short to the path between A and B as follows.
Further labels on the path only help if they are very close to the short. If, however, you put more labels on the already eliminated part of the net, the isolation becomes precise.

Defining Active Layers

You can define which of the layers forming your interconnections are to be active. The difference between active and inactive layers is as follows:

- An active layer accepts the addition of a text label. If a layer is not active, you cannot add a label to it. By setting global layers such as tub and substrate to inactive status, you can add labels to shapes on layers over them without chance of confusion.

- Active and inactive layers are displayed in different colors and fill patterns. This lets you set the color and fill patterns of layers such as tub and substrate to a lower priority and more muted color than other interconnect layers. If you do not do this, tub and substrate hide all other parts of the net.

Displaying Information

Information about the shorted net is displayed on top of any existing display in all windows containing the layout view of the cell you are processing.

The short location program uses five different display colors and fill patterns. It clones these colors and fill patterns from predefined system layers. You can change the color and fill patterns by changing the system layers using the Layer menu in the CIW. The program clones the new colors and fill patterns after you have changed them.

The five layers and their uses are as follows:

hilite drawing Displays the text labels.

hilite drawing1 For active layers, displays the part of the net that has been determined to contain the short.
hilite drawing2 For active layers, displays the original net. All shapes forming the net, regardless of the layer from which they came, are displayed in the same color.

hilite drawing3 For inactive layers, displays the part of the net that has been determined to contain the short.

hilite drawing4 For inactive layers, displays the original net. All shapes forming the net, regardless of the layer from which they came, are displayed in the same color.

Usually, you should turn off the inactive layers or set them to more muted colors and lighter fill patterns than the active layers. Set the shorted part of the net to a higher intensity color and a more solid fill pattern than the full net. You should also set the text label color to show above everything else.

**Saving and Restoring**

Since the isolation of a short is an iterative process, you may want to stop the process before you are satisfied with the result, and come back another time to complete it. For this purpose, you can save and restore a run.

When you save the run, you define the file name and its location. When you restore a run, you can define this file name or any other file name you have saved. The contents of the file are in text format that you can edit using a conventional system text editor. However, the integrity of the restored run cannot be guaranteed if you edit the file.

**Note:** The saved information is not the entire status of the run, but gives enough information to return to that status with one iteration of the Short Locator program. That iteration is performed automatically when you restore the run.
Short Locator Form

Label Height defines the height of the characters for the label text you enter. You can either choose a default height or type in a height in user units.

match existing text uses the text height from the first label Short Locator reads from the circuit where the layer is defined in a geomConnect command label definition. If there is no text, Short Locator uses a default value of two user units.

user-defined lets you type a height in user units for labels shown in the cellview.

Net Name specifies the name of the net you want to analyze. You can type in the name or use the mouse to select it.

Sel by Cursor lets you point to the net with the mouse. If more than one net is under the cursor, a list box appears listing all nets. Short Locator enters the name of the net you select in the name field.

Run mode defines which mode to use when running Short Locator.

new run initiates a new run of short locations.
**Diva Reference**  
**Short Locator**

*rerun* starts a run after a new run has been completed and you have added or removed labels.

**Run** starts the Short Locator, which runs in one of the three run modes you selected.

**Add Label** adds a label to the net being analyzed.

**Remove Label** removes an existing label from the net being analyzed.

**Fit Short** zooms in on the short area.

**Restore** initiates a local run that uses information derived from a previous stored run.

**Save** saves the current run information for a later restart.

**Finish** ends the Short Locator run and removes net or text highlights.

**Active Layers** lists circuit layers that are part of the interconnect. You can choose whether layers are active or inactive. Making a layer inactive turns off the highlighting of that layer, which is useful for layers such as a substrate, whose display highlights the complete circuit and interferes with selecting a net with a mouse. You cannot add text to an inactive layer. However, inactive layers are included in the short analysis.

**Using Shorts**

1. Select *Shorts* from the Verify menu, or type *ivHiShorts* in the CIW.
   
   If the extracted cellview of the circuit to be processed contains an *ivConnectErrors* property, the following dialog box appears:

   ![Image of dialog box]

   Missing interconnect layers can cause incorrect short location results.

2. To select the label height, click on *user-defined* and type in the height you want.

3. To make layers active or inactive, click on the layers to turn them on or off. You can set all layers on or off by selecting *all or none*.
4. Specify the net name in one of the following ways:
   - Type the name in the field.
   - Click on *Sel by Cursor* and then select a net.

   The CIW prompts you to point at the net you want.

   If you select a net with the mouse, you might pick up the substrate, the well, or some other global layer if those layers are left active.

   If more than one net appears under the cursor, a list box appears listing all the nets. Click on a net and select *OK*. The name of the net you selected is entered into the name field.

5. Set the run mode to *new run* and click on *Run*.

   Short Locator uses any text that was used to label the nets during extraction. The initial net, text points, and shorted section of the net are then displayed in any window with an extracted or layout view of the cell.

   **Note:** If you click on the new run mode, specify a new net name, and then click *Run*, the Diva verification tool displays a dialog box warning that you must end one run (using *Finish*) before starting another.

6. To add or remove a label, click on *Add Label* or *Remove Label*.

7. Set the run mode to *rerun* and click on *Run*.

   You cannot start a new run before finishing the current run.

8. To zoom into the short area, click on *Fit Short*, or to save the current run status, click on *Save*.

9. To restore a previous run status, click on *Restore*.

10. When you’re finished analyzing a short, click on *Finish* to remove all net and text highlights from the cellview window.

**Connect Errors Property**

This tool displays the connect errors dialog box if the extracted cellview of the circuit has an *ivConnectErrors* property. This property is created if, during layout extraction, some interconnect layers are not saved in the extracted cellview. Missing layers cannot be used during the short analysis. If a missing layer contributes to the short or is essential to the correct analysis of the short, the short analysis program cannot do its job.
Note: The exception to this rule is if a layer is only attached to a net at one point, as in the case of a tub or well; the absence of the layer does not affect the connectivity of the rest of the net.
Translating PDV Files to Diva Verification Files

The pdvtodiva Command

To translate a PDV file to a Diva verification file, you must use the pdvtodiva command. The command invokes the pdvtodiva program that translates a PDV rules file into a Diva verification rules file.
pdvtodiva

pdvtodiva key [-l libName] [-r rulesFile] [-o outputFile] [-t techFile]

Description

Translates a PDV rules file into this verification tool's rules file. During the translation, the program checks the syntax of the PDV rules file. The program, as it runs in the UNIX environment, warns you if errors or ambiguities are found in the PDV rules file.

The resulting verification product rules file is correctly formatted for Diva verification. The pdvtodiva program preserves comments and nested commands from the PDV file.

Arguments

key

A keyword from this list identifies the program for rules conversion.

drc design rule checking
extract device, parameter, and parasitic extraction
erc electrical rule checking
lvs layout versus schematic comparison
abgen abstract generation

-l libName

The name of the library that contains the technology file that provides layer information required for translation.

You must specify the -l lib_name argument when the rules you translate are for DRC, Extract, or Abgen. You do not need to specify -l lib_name when you translate ERC or LVS rules.

-r rulesFile

An optional name for the PDV rules file to be translated. The default is myrules.

-o outputFile

An optional name for the output file of the translated data. If you do not specify an output file name, the program creates a name by appending .diva to the rules file name.
This name can be a file name in the current directory, a path name relative to the current directory, or an absolute path name. To specify an output file inside a library, you must specify the absolute path name.

-t techFile

The name of the technology file inside the library specified by the -l option. If no technology file is specified, techfile.cds is used as the default.

The technology file can be a file name or a path name relative to the library directory referencing the cell or view levels in the directory.

Examples

In the following example, the program translates the file drcRules and create an output file called drcRules.diva in the current directory. The translation uses the layers from the techfile.cds file in the mylib directory.

```bash
pdvtodiva drc -r drcRules -l mylib
```

In the following example, an LVS rules file is translated using the default input file name of myrules:

```bash
pdvtodiva lvs -o myoutfile
```

In the following example, the program gets the layers information from the file mytech inside the library mylib:

```bash
pdvtodiva extract -l mylib -r pdvrules -t mytech
```
Translating Dracula Files

Dracula To Diva Verification Translator: DraculaToDiva

The DraculaToDiva program translates a Dracula rules file into an Diva verification rules file.

Before you translate your files

- Make sure that the directory containing DraculaToDiva is in your search path.
- Make sure that you have a Dracula rules file.

Running DraculaToDiva

You can execute DraculaToDiva from any UNIX prompt.

When you invoke DraculaToDiva, the Dracula program heading appears, along with a colon (:) prompt.

```
****************************************************************************
*/N*            Dracula To Diva (REV. 1.0/ SUN-3 (4.0) /GENDATE: 01-May-89 )
*** ( COPYRIGHT 1988 ) ***
*/N*            EXEC TIME =08:25:41       DATE = 5-FEB-90
******************************************************************************
```

You can use these PDRACULA commands with DraculaToDiva.

```
/get

Specifies the rules file to be translated.
syntax: get [filename][nolist]
or
/get

The nolist option causes the rules file not to be displayed on the screen.
```
Use the *nolist* option to display all Dracula errors in a group, rather than interspersed between the scrolling rules-file text.

/finish  
Tells the translator that input is complete and generates the new Diva verification rules file.

/abort  
Terminates translator operation.

Other PDRACULA commands can be used with DraculaToDiva. Refer to the Dracula Reference Manual for more information.

**What Happens During Translation?**

When you type /get and specify the rules file, DraculaToDiva scans the file and displays errors on the screen.

When you type /finish, DraculaToDiva generates an Diva verification rules file.

**Dracula To Diva Verification Command Correspondence**

These figures list Dracula commands and their corresponding Diva verification commands. For more information about Dracula or Diva verification commands, see the Dracula Reference Manual.

**Dracula To Diva Command Correspondence**

<table>
<thead>
<tr>
<th>Dracula command</th>
<th>Diva Verification command</th>
</tr>
</thead>
<tbody>
<tr>
<td>LAYER Processing – Logical</td>
<td></td>
</tr>
<tr>
<td>OR layerA layerB layerC</td>
<td>layerC = geomOr( layerA layerB )</td>
</tr>
<tr>
<td>AND layerA layerB layerC</td>
<td>layerC = geomAnd( layerA layerB )</td>
</tr>
<tr>
<td>NOT layerA layerB layerC</td>
<td>layerC = geomAndNot( layerA layerB)</td>
</tr>
<tr>
<td>XOR layerA layerB layerC</td>
<td>layerC = geomXor( layerA layerB )</td>
</tr>
<tr>
<td>FLATTEN layerA layerB</td>
<td>layerB = geomCat( layerA )</td>
</tr>
<tr>
<td>HIERARCHEN layerA layerB</td>
<td>No corresponding Diva verification command.</td>
</tr>
<tr>
<td>SELECT layerA INSIDE layerB layerC</td>
<td>layerC = geomInside( layerA layerB )</td>
</tr>
<tr>
<td>SELECT layerA OUTSIDE layerB layerC</td>
<td>layerC = geomOutside( layerA layerB )</td>
</tr>
</tbody>
</table>
### Diva Reference

#### Translating Dracula Files

<table>
<thead>
<tr>
<th>Dracula command</th>
<th>Diva Verification command</th>
</tr>
</thead>
<tbody>
<tr>
<td>SELECT layerA CUT layerB layerC</td>
<td>layerC = geomStraddle( layerA layerB )</td>
</tr>
<tr>
<td>SELECT layerA ENCLOSE layerB layerC</td>
<td>layerC = geomEnclose( layerA layerB )</td>
</tr>
<tr>
<td>SELECT layerA VERTEX[n1:n2] layerB</td>
<td>layerB = geomGetVertex( layerA n1&lt;= keep &lt;= n2 )</td>
</tr>
<tr>
<td>SELECT layerA LABEL labelname layerB</td>
<td>layerB = geomGetNet( layerA &quot;labelname&quot; )</td>
</tr>
<tr>
<td>SELECT layerA HOLE layerB layerC</td>
<td>layerC = geomEnclose( geomInside geomHoles (layerA) layerB )</td>
</tr>
<tr>
<td>SELECT layerA TOUCH[n1:n2]layerB</td>
<td>layerC = geomButting( layerA layerB n1 &lt;= keep &lt;= n2 )</td>
</tr>
<tr>
<td>SIZE layerA by n layerB</td>
<td>layerB = geomSize( layerA n )</td>
</tr>
<tr>
<td>OUTPUT ERR 10</td>
<td>saveDerived( ERR10 )</td>
</tr>
<tr>
<td>RELOCATE</td>
<td>No corresponding Diva verification command.</td>
</tr>
<tr>
<td>CUT</td>
<td>No corresponding Diva verification command.</td>
</tr>
</tbody>
</table>

**Note:** The Diva verification drc command automatically stores the result in the database, so it is not necessary to use the Diva verification saveDerived command.

**Layer Connectivity**
Dracula To Diva Command Correspondence

<table>
<thead>
<tr>
<th>Dracula command</th>
<th>Diva Verification command</th>
</tr>
</thead>
<tbody>
<tr>
<td>CONNECT</td>
<td>geomConnect(</td>
</tr>
<tr>
<td>*input-layer</td>
<td>via( via L2 L1 )</td>
</tr>
<tr>
<td>L1 = 1</td>
<td>via( contact L3 L2 L1 )</td>
</tr>
<tr>
<td>L2 = 2 text = 20 attach L4</td>
<td>label( “text” L4 )</td>
</tr>
<tr>
<td>L3 = 3 ctext = 21</td>
<td>label( “ctext” L1 L2 L3 )</td>
</tr>
<tr>
<td>connect-layer = L1 L2 L3</td>
<td></td>
</tr>
<tr>
<td>*end</td>
<td></td>
</tr>
<tr>
<td>*operation</td>
<td></td>
</tr>
<tr>
<td>connect L2 L1 by via</td>
<td></td>
</tr>
<tr>
<td>connect L3 L2 by contact</td>
<td></td>
</tr>
<tr>
<td>connect L3 L1 by contact</td>
<td></td>
</tr>
<tr>
<td>*end</td>
<td></td>
</tr>
</tbody>
</table>

Note: You need to change the label names inside the label function to the names you use in the Diva verification database.

STAMP layerA by layerB
layerA = geomStamp( layerA layerB )

LINK
No corresponding Diva verification command.

RCONNECT
RCONNECT is used to form parasitic resistance.

Note: Since there is no straightforward way to transform Dracula parasitic-resistance checking to Diva verification commands, all RCONNECT statements are commented out. Messages are displayed on the screen or in the file.

Design Rule Checks

WIDTH layerA op n1 n2 layerB
layerB = drc( layerA n1 <= width <= n2 )

ENC layerA layerB op n1 n2 layerC
layerC = drc( layerB layerA n1 <= enc <= n2 )

The Dracula ENC layerA layerB checks for layerB enclosing layerA. In Diva verification, the order is reversed, and layerB is specified as the first layer.
## Dracula To Diva Command Correspondence

<table>
<thead>
<tr>
<th>Dracula command</th>
<th>Diva Verification command</th>
</tr>
</thead>
<tbody>
<tr>
<td>EXT layerA layerB op n1 n2 layerC</td>
<td>layerC = drc( layerA layerB n1 &lt;= sep &lt;= n2 )</td>
</tr>
<tr>
<td>INT layerA layerB op layerC</td>
<td>layerC = drc( layerA layerB n1 &lt;= ovlp &lt;= n2 )</td>
</tr>
<tr>
<td>LENGTH layerA op n1 layerC</td>
<td>Temp-File1 = geomGetLength( Temp-file2 length op n1 )</td>
</tr>
<tr>
<td>length L3 Le .002 &amp;</td>
<td>LENGTH can only be used in a spacing conjunction rule and does not need the edge modifier. See the example.</td>
</tr>
<tr>
<td>enc L3 L1 Lt .001 &amp;</td>
<td>TEMP:?1 = drc( L1 L3 enc &lt; .001 edge )</td>
</tr>
<tr>
<td>enc L3 L2 Lt .003 out err 11</td>
<td>?L3 = geomAnd( TEMP:?1 L3 )</td>
</tr>
<tr>
<td>AREA layerA RANGE n1 n2 layerB</td>
<td>layerB = drc( layerA n1 &lt;= area &lt;= n2 )</td>
</tr>
<tr>
<td>OVERLAP layerA layerB</td>
<td>layerB = geomAnd( layerA )</td>
</tr>
<tr>
<td>PLENGTH layerA RANGE n1 n2 layerB</td>
<td>layerB = geomGetLength( layerA n1 &lt;= length &lt;= n2 contiguous )</td>
</tr>
</tbody>
</table>

### Dracula DRC Options

When a Dracula DRC command has one or more options, DraculaToDiva translates the option(s) to an Diva verification `drc` command with modifiers. Where needed, additional commands are added by DraculaToDiva for specific options.

- **[C]** Uses Diva verification `parallel` modifier.
- **[C']** Uses Diva verification `notParallel` modifier.
- **[E]** Combines the commands.
- **ENC[E]layerA layerB op n1 n2 layerC**
  ```
  layerC = drc( layerB layerA n1 <= op <= n2 )
  saveDerived( geomOutside(layerA layerB ) )
  ```
## Diva Reference
### Translating Dracula Files

## Dracula To Diva Command Correspondence

<table>
<thead>
<tr>
<th>Dracula command</th>
<th>Diva Verification command</th>
</tr>
</thead>
</table>
| EXT[E]layerA layerB op n1 n2 layerC | layerC = drc( layerA layerB n1 <= sep <= n2 )  
 |                           | saveDerived( geomInside( layerA layerB ) )                                      |
| [EO]                     | Combines the commands.                                                         |
| EXT[EO]layerA layerB op n1 n2 layerC | layerC = drc( layerA layerB n1 <= sep <= n2 )  
 |                           | saveDerived( geomOverlap( layerA layerB ) )                                    |
| [G] or [O]               | Combines the commands.                                                         |
| EXT[G]layerA layerB op n1 n2 layerC | layerC = drc( layerA layerB n1 <= sep <= n2 )  
 |                           | saveDerived( geomStraddle( layerA layerB ) )                                   |
| [H]                      | Combines the commands.                                                         |
| EXT[H] layerA op n1 n2 layerC | layerC = drc( layerA n1 <= sep <= n2 )  
 |                           | drc( layerA n1 <= notch <= n2 )                                                |
| [N]                      | Uses Diva verification $diffNet$ modifier.                                     |
| [N']                     | Uses Diva verification $sameNet$ modifier.                                     |
| [P]                      | Uses Diva verification opposite modifier.                                      |
| [P']                     | Uses Diva verification app<0 modifier.                                         |
| [R] [R']                 | Default Diva verification operation generates shapes. Default Dracula operation generates edges. When you do not specify the [R] or [R'] option in a Dracula command file, DraculaToDiva automatically adds an edge modifier. When you specify an [R] modifier in the Dracula command file, also specify the Diva verification opposite modifier. |
| SELLE or SELLT           | Uses Diva verification $fig$ modifier.                                          |
| [T]                      | Default Diva verification operation has touching on. Whenever there is no [T] specified in the Dracula DRC, and there is no lower bound, DraculaToDiva adds a lower bound of 0 without the equal sign. |
| EXT layerA layerB LT 0.5 layerC | layerC = drc( layerA layerB 0 < sep < 0.5 )                                   |
| EXT[T]layerA layerB LT 0.5 layerC | layerC = drc( layerA layerB sep < 0.5 )                                       |
| [U]                      | Ignored.                                                                       |
## Dracula To Diva Command Correspondence

<table>
<thead>
<tr>
<th>Dracula command</th>
<th>Diva Verification command</th>
</tr>
</thead>
<tbody>
<tr>
<td>[U′]</td>
<td>Ignored.</td>
</tr>
<tr>
<td>[V]</td>
<td>Ignored.</td>
</tr>
</tbody>
</table>
### Dracula To Diva Command Correspondence

<table>
<thead>
<tr>
<th>Dracula command</th>
<th>Diva Verification command</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Layout Extraction</strong></td>
<td></td>
</tr>
<tr>
<td><code>ELEMENT MOS[type] layerA layerB layerC layerD</code></td>
<td><code>extractDevice( layerA layerB(&quot;G&quot;) layerC (&quot;S&quot; &quot;D&quot;) layerD(&quot;B&quot;) model )</code></td>
</tr>
<tr>
<td>-</td>
<td>For example, the model can be “pfet” or “nfet,” depending on whether the MOS is p-type or n-type.</td>
</tr>
<tr>
<td><code>ELEMENT RES[type] layerA layerB</code></td>
<td><code>extractDevice( layerA layerB(&quot;PLUS&quot; &quot;MINUS&quot;) &quot;resistor&quot; )</code></td>
</tr>
<tr>
<td><code>ELEMENT CAP[type] layerA layerB layerC</code></td>
<td><code>extractDevice( layerA layerB(&quot;PLUS&quot;) layerC(&quot;MINUS&quot;) &quot;capacitor&quot; )</code></td>
</tr>
<tr>
<td><code>ELEMENT DIO[type] layerA layerB layerC</code></td>
<td><code>extractDevice( layerA layerB(&quot;PLUS&quot;) layerC(&quot;MINUS&quot;) &quot;diode&quot; )</code></td>
</tr>
<tr>
<td><code>ELEMENT BJT[type] layerA layerB layerC layerD</code></td>
<td><code>extractDevice( layerA layerB(&quot;collector&quot;) layerC(&quot;base&quot;) layerD(&quot;emitter&quot;) &quot;bjt&quot; )</code></td>
</tr>
<tr>
<td><code>ELEMENT PAD layerA layerB</code></td>
<td>Not implemented.</td>
</tr>
<tr>
<td><code>ELEMENT BOX layerA layerB layerC layerD layerE</code></td>
<td><code>extractDevice( layerA layerB(&quot;pos1&quot;) layerC(&quot;pos2&quot;) layerD(&quot;pos3&quot;) layerE(&quot;pos4&quot;) &quot;BOX&quot; )</code></td>
</tr>
</tbody>
</table>

*BOX* is the user-defined device name.

| **PARAMETER CAP** | |
| `ELEMENT CAP[type] layerA layerB layerC` | `area = measureParameter( area (layerA) n1 )` |
| - | `saveParameter( area “ca” )` |
| `PARAMETER CAP[type] n1 n2` | `peri = measureParameter( perimeter savingParameter(peri “cp”)(layerA) n2 )` |

| **PARAMETER RES** | |
| `ELEMENT RES[type] layerA layerB` | `area = measureParameter( area (layerA) ) width = measureParameter( length ( layerA butting layerB) 0.5 ) resist = calculateParameter(area / ( width * width)* n1 )` |
| `PARAMETER RES[type] n1` | `saveParameter(resist “r”)` |
## Diva Reference

### Translating Dracula Files

### Dracula To Diva Command Correspondence

<table>
<thead>
<tr>
<th>Dracula command</th>
<th>Diva Verification command</th>
</tr>
</thead>
<tbody>
<tr>
<td>PARASITIC CAP</td>
<td>See ATTRIBUTE CAP.</td>
</tr>
<tr>
<td>ATTRIBUTE CAP</td>
<td></td>
</tr>
<tr>
<td>PARASITIC CAP[type] layerA layerB layerC</td>
<td>area = measureParasitic( area ( layerA over layerC ) two_net )</td>
</tr>
<tr>
<td>ATTRIBUTE CAP[type] n1 n2</td>
<td>peri = measureParasitic( perimeter ( layerA over layerC ) two_net )</td>
</tr>
<tr>
<td></td>
<td>capacitor = calculateParasitic( (area<em>n1)+(peri</em>n2) )</td>
</tr>
<tr>
<td></td>
<td>saveParasitic( capacitor “PLUS” “MINUS” “c” “capacitor” )</td>
</tr>
<tr>
<td>fringe-cap example:</td>
<td></td>
</tr>
<tr>
<td>PARASITIC CAP[type] layerA layerA</td>
<td>fringe = measureFringe( layerA</td>
</tr>
<tr>
<td>ATTRIBUTE CAP[type] n1 n2</td>
<td>calculate((l*n2)/s)</td>
</tr>
<tr>
<td>PARASITIC RES</td>
<td>See ATTRIBUTE RES.</td>
</tr>
<tr>
<td>ATTRIBUTE RES</td>
<td></td>
</tr>
<tr>
<td>PARASITIC RES[type] layerA layerB</td>
<td>area = measureParasitic( area( layerA ) two_net )</td>
</tr>
<tr>
<td>ATTRIBUTE RES[type] n1</td>
<td>width = measureParasitic( length ( layerA butting layerB ) 0.5 two_net )</td>
</tr>
<tr>
<td></td>
<td>resist = calculateParasitic( area/(width * width ) * n1 )</td>
</tr>
<tr>
<td></td>
<td>saveParasitic( resist “PLUS” “MINUS” “r” “resist” )</td>
</tr>
<tr>
<td>PARASITIC DIO</td>
<td>In PARASITIC DIO, layerD and layerE are optional; if they do not appear, you do not need to calculate a2, p2, a3, and p3.</td>
</tr>
<tr>
<td>PARASITIC DIO[type] layerA layerB layerC layerD layerE</td>
<td>a1 = measureParasitic( area (layerA over layerB not_over layerD not_over layerE) two_net )</td>
</tr>
</tbody>
</table>
Dracula To Diva Command Correspondence

Dracula command | Diva Verification command
--- | ---
If P-N diode, then layerA over layerC | p1 = measureParasitic( perimeter (layerA over layerB outside layerD outside layerE) two_net )
If N-P diode, then layerA over layerB | a2 = measureParasitic( area (layerA over layerD not_over layerE) two_net )
p2 = calculateParasitic( measureParasitic( perimeter (layerA over layerD outside layerE) two_net) + measureParasitic( perimeter (layerA butting layerD outside layerE) two_net) )
a3 = measureParasitic( area (layerA over layerE) two_net )
p3 = calculateParasitic( measureParasitic( perimeter (layerA over layerE) two_net) + measureParasitic( length (layerA butting layerE) two_net) )
saveParasitic( (a1 p1) “PLUS” “MINUS” (“a1” “p1”) “diode” )
saveParasitic( (a2 p2) “PLUS” “MINUS” (“a2” “p2”) “diode” )
saveParasitic( (a3 p3) “PLUS” “MINUS” (“a3” “p3”) “diode” )
DEVTAG | DEVTAG is used to attach a device number to a certain layer, enabling you to extract the parameter from that layer via LEXTRACT. In Diva verification, the approach is reversed. You first calculate the parameters of a specific layer, which is not a recognition layer, and then you associate that layer to the recognition layer. See the example under LEXTRACT.
LEXTRACT | extractDevice( layerA layerA(“collector”) layerB(“base”) layerC(“emitter”) “bjt”) area = measureParasitic( area (layerE over layerC) figure )
### Dracula To Diva Command Correspondence

<table>
<thead>
<tr>
<th>Dracula command</th>
<th>Diva Verification command</th>
</tr>
</thead>
</table>
| LEXTRACT parset layerA by BJT[type] | attachParasitic(area “prop_1” layerA layerC)  
peri = measureParasitic(perimeter(layerE over layerC) figure)  
attachParasitic(peri “prop_2” layerA layerC)  
area = measureParameter(area (layerA))  
saveParameter(area “a”)  
peri = measureParameter(perimeter (layerA))  
saveParameter(peri “l”) |
| EQUATION | EQUATION is used for flexible parameter extraction with the LEXTRACT command. You calculate the primitive parameters for each different type of device and then use calculateParameter for the EQUATION. |
| ELEMENT BJT[type] layerC layerA layerB layerC | area = measureParameter(area (layerC))  
saveParameter(area “a”)  
peri = measureParameter(perimeter (layerC)) |
| LEXTRACT parset layerC by BJT[type] outfile & | saveParameter(peri “l”)  
c1 = calculateParameter(1.2 * area + 1.3 * peri) |
| EQUATION c1 = 1.2 * area + 1.3 * peri | saveParameter(c1 “c1”) |
| LPESELECT | Ignored. |
| LPECHECK | LPECHECK compares the layout and schematics and cannot be translated. |
| NDCOUNT | If layerC is the source-drain layer, DivaToDiva assigns “S” and “D” as the layer’s terminal types. In this example, an MOS element is defined as |
| NDCOUNT MOS[P] layerC GT 2 | checkDeviceNetCount("pfet" ("S" "D") keep > 2) |
Dracula To Diva Command Correspondence

<table>
<thead>
<tr>
<th>Dracula command</th>
<th>Diva Verification command</th>
</tr>
</thead>
<tbody>
<tr>
<td>ECONNECT</td>
<td>If layerA is the source-drain layer, <em>DraculaToDiva</em> assigns “S” and “D” as its terminal types. The conjunction rule cannot be translated. If using DISC, <em>DraculaToDiva</em> uses checkNotConnected.</td>
</tr>
<tr>
<td>ECONNECT MOS[P] layerA CONN VDD</td>
<td>checkConnected( &quot;pfet&quot; (&quot;S&quot; &quot;D&quot;) (&quot;VDD&quot;) )</td>
</tr>
<tr>
<td>LCONNECT</td>
<td>The connect-layers in a <em>geomCat</em> are a list of all the connect layers that are defined by the connect-layer command in the input-layer block of a Dracula rules file. The <em>geomGetNet</em> command is placed in the <em>drcExtractRules</em> section of the corresponding Diva verification rules file.</td>
</tr>
<tr>
<td>LCONNECT layerA CONN VDD</td>
<td>geomGetNet( geomCat( connect-layers) &quot;VDD&quot;)</td>
</tr>
<tr>
<td>LCONNECT layerB DISC VDD</td>
<td>temp = geomGetNet( geomCat(connect-layers) “VDD”) geomAndNot( layerB temp )</td>
</tr>
<tr>
<td>ELCOUNT</td>
<td>If layerA is the source-drain layer, <em>DraculaToDiva</em> assigns “S” and “D” as its terminal types. The conjunction rule cannot be translated.</td>
</tr>
<tr>
<td>ELCOUNT MOS[P] layerA GT 1</td>
<td>checkFanOut( &quot;pfet&quot; (&quot;S&quot; &quot;D&quot;) keep &gt; 1 )</td>
</tr>
<tr>
<td>PATHCHK</td>
<td>If you use nonstandard labels for power or ground in your Dracula erc rules file, you need to modify the labels in the translated Diva verification file by using <em>setPower</em> or <em>setGround</em>.</td>
</tr>
<tr>
<td>PATHCHK LEVEL 1 OUTPUT ERR 10</td>
<td>setPower( “VDD” “VCC” “PWR” ) setGround( “VSS” “GND” ) checkPullUp( ) checkPullDown( )</td>
</tr>
<tr>
<td>PROBE</td>
<td>The translated rule appears in the <em>drcExtractRules</em> part. The conjunction rule cannot be translated.</td>
</tr>
<tr>
<td>PROBE VDD OUTPUT VDDNET 10</td>
<td>tempt = geomGetNet( geomCat(connect-layers) “VDD”) saveDerived( tempt(&quot;y0&quot; “drawing”) ext_view) )</td>
</tr>
</tbody>
</table>
### Diva Reference
Translating Dracula Files

#### Dracula To Diva Command Correspondence

<table>
<thead>
<tr>
<th>Dracula command</th>
<th>Diva Verification command</th>
</tr>
</thead>
<tbody>
<tr>
<td>MULTILAB</td>
<td>Diva verification automatically executes MULTILAB.</td>
</tr>
<tr>
<td>SAMELAB</td>
<td>Diva verification automatically executes SAMELAB.</td>
</tr>
</tbody>
</table>

#### Layout Vs. Schematic

<table>
<thead>
<tr>
<th>Command</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>LVSCHK</td>
<td>No corresponding Diva verification command. Whenever LVSCHK or LPECHK appear in the rules file, DraculaToDiva inserts an lvsRules block and a comment asking you to run the LVS interactively, using Diva verification iLVS.</td>
</tr>
<tr>
<td>LVSPlot</td>
<td>No corresponding Diva verification command.</td>
</tr>
</tbody>
</table>
SQUARE Check

Dracula has a check similar to the Diva verification check for alignment and registration checking, called the SQUARE check. The philosophy of the SQUARE check is different. This section includes a brief description of this check and the methodology for translating rules from DraculaToDiva checking.

Checking Philosophy

The SQUARE check has normal spacing dimensions parallel to edges, but has special checks at 90 degree corners. There are two corner dimensions involved. The first dimension is for the source corner to a destination edge. The second dimension is for the source corner to a destination corner. The way in which these dimensions are applied is illustrated by these examples.

This example shows a corner to edge spacing. The normal side spacing is “S” but the spacing at 45 degrees on the corners is X root 2.
The basic intent behind this rule is to simulate the case of sizing the original shape by one value and then checking the regular spacing by a second value. The same check with the Diva verification rules is shown.

The original edge face spacing value remains the same. In the Diva verification example, the corner spacing is calculated

\[ g + g + (S-2g) \]

This is equivalent to the Dracula corner to edge spacing of

\[ \sqrt[2]{g} \]

Putting the two together we get the conversion calculation of

\[ x = ((-1)g+s) / \]

The same check, except with corner to corner, is shown.

For exactly 45 degree separation, the Diva verification corner spacing is calculated as

\[ g + g + (s-2g) \]

This results in this calculation of the Dracula X value.

\[ x = \frac{2 \cdot (-1)g + s}{l} \]

**Translation Pitfalls**

The Dracula check has two independent dimensions which need not be consistent with each other or consistent with an overall measurement philosophy. It is anticipated therefore that the dimensions derived for the Diva verification checks from translation of corner to corner and corner to edge will be different from each other.

The translation cannot work if in the Dracula check, the dimension X is greater than the dimension S. Also, the values cannot be computed if \( x \cdot \sqrt{2} \) (the corner check dimension) is less than S. This would violate the basic spacing requirement that nothing comes closer than S at any point.
Even if the numbers map from the Dracula check to the Diva verification check, the results need not be the same. The diagram illustrates this.

This diagram superimposes the checking methodologies of Dracula and Diva verification for corner to edge separation. The dark area is one example of an area which if it was part of the original polygon would not be treated as an error by Dracula but, would be an error in Diva verification.
Improving Hierarchical DRC Performance

This appendix describes two methods for improving hierarchical DRC performance.

- An extended methodology is available for improving performance when you have nonrectangular cells whose bounding boxes overlap significantly but whose data overlaps by a small amount.

- Switches are available that let IDRC address a small but difficult class of design rule checks. IDRC has switch names that you can test during certain phases of program execution.

Assigning the \textit{hdrc boundary} Layer

To improve performance when you have nonrectangular cells whose bounding boxes overlap significantly but whose data overlaps by a small amount, you need to create a layer in your technology file \textit{lkcalled hdrc boundary}. You can assign any number between 0 and 127 to the \textit{hdrc} layer. Only the name and purpose are essential for identifying the layer.

IDRC checks for the existence of layers only when the rules are compiled. IDRC is not aware of the \textit{hdrc} layer if you add it during an interactive session unless you update the library by reloading the rules or by closing and reopening the library.

\textbf{Note:} Do not digitize data on the \textit{hdrc} layer. The layer is erased and created automatically by IDRC. See the “Rules File Example” section for an example.

How the \textit{hdrc boundary} Layer Works

With the extended methodology, a polygonal boundary shape is created for each cell on the \textit{hdrc boundary} layer. The shape is always manhattan, containing no slanted edges. The boundary is formed by merging all shapes in the cell with all boundary shapes of lower level cells. The shapes are converted into trapezoids in the same manner as the tile option for \textit{saveDerived} functions. Each trapezoid is converted into a rectangle by using the bounding box of the trapezoid. This shape is then “grown” by the halo distance of the rules. The grow
process removes small notches and holes. Lastly, the shape is shrunk by the halo distance and stored in the database on the *hdrc* layer with a *boundary* purpose.

The hierarchical DRC program uses these boundary shapes to determine interactions between cells or between instances and polygons in the cell. When cells are not rectangular, the area that is rechecked is greatly reduced. This improves performance significantly.

Note the following:

- There might be multiple boundary shapes and the shapes might have holes. The boundary shapes are available for use during DRC execution by using the `geomGetPurpose` command.

- Small cells whose size is less than twice the halo distance are treated as polygons and not as real cells, meaning they are processed as flat data. Small cells do not have boundary shapes placed in them.

### Hierarchical DRC Switch Names

IDRC has three switches you can set during certain phases of program execution. These switches let IDRC address a very specific class of design rule checks.

- **hier?**
  
  This switch lets you use a different set of rules in hierarchical mode than in flat mode. In hierarchical mode, the `hier?` switch is automatically on.

- **currentCell?**
  
  This feature lets you write rules that are cell based and not truly hierarchical. The `currentCell?` switch is toggled on and off as the design is checked. In hierarchical mode, checking occurs in two passes. The first pass uses only the polygons in the current cell. The second pass runs in flat mode, from the current cell down, rechecking small areas where two instances overlap or where an instance and a polygon overlap in the current cell. The `currentCell?` switch is set on in the first pass and set off in the second pass.

- **topCell?**
  
  This switch, when present, instructs Diva verification to perform a flat DRC analysis on the top cell after all hierarchical checking is done. You can use this feature to check sparse data, for example pads, to improve execution speed.

In the final pass, you can run a rules check in flat mode on data that is best checked in flat mode. For example, pad checks have limited data and do not check efficiently in hierarchical mode.
Rules File Example

To understand how the switches can be used, consider four simple checks on a design.

1. Flag any metal spacing less than 0.5 microns.

2. Flag any pad spacing that is less than 10 microns. The pad layer is a sparse layer and is better checked in flat mode. This check can be done in hierarchical mode, but it increases the size of the halo, which makes the program run slow even when there is no pad layer in the vicinity.

3. Check that any cell containing layer Q also contains layer R and that layer R is exactly 0.4 micron larger than layer Q. This is where the currentCell? switch comes into play.

4. Check that no part of the design is more than 1 micron from a piece of metal. This check is easily done in flat mode.

Note: This check is the most difficult of the four to implement.

\[
\text{m1g = geomSize( "m1" 0.5 )}
\]
\[
\text{err = geomAndNot( geomBkgnd() m1g )}
\]
\[
\text{saveDerived( err "m1 coverage error" )}
\]

The problem in hierarchical mode is that instances form an area where no polygonal data exists and a large error is generated. It is necessary to find errors that are not over instances, yet errors must also be found that are between instance polygons and cell polygons.

This problem is solved by introducing a new layer in the data base, \textit{m1 boundary}. As the design is processed, a boundary shape for \textit{m1} is created. The check is done by looking at shapes at the current level and \textit{m1} boundaries from instances that have already been processed.

The rules for the check follow. The rules apply only to hierarchical mode.

```plaintext
drcExtractRules(
    ivIf( switch( "drc?" ) then
        ivIf( switch( "hier?" ) then
            ivIf( switch( "topCell?" ) then
                ; check 2
                pad = geomOr( "pad" )
                drc( pad sep < 10 "pad spacing less than 10" )
            else
                m1 = geomOr( "m1" )
                ; check 1
                drc( m1 sep < 0.5 "m1 spacing less than 0.5" )
            ivIf( switch( "currentCell?" ) then
```
; check 3
check3 = geomXor( geomSize("Q" 0.4) "R"
saveDerived( check3 "R and Q check"

; check 4
; Remove any m1 boundary shapes because they are
going to be rederived
geomErase("m1" "boundary"

; Collect m1 in the current cell with m1 boundary
polygons from one level down in the hierarchy.
mlb = geomOr("m1" geomGetPurpose("m1" "boundary" 1)

; Grow the m1 shapes by 0.5 microns to find areas
that are not covered.
mlbg = geomSize( mlb 0.5

; Get the cell boundary to
; (1) determine where a coverage error should be flagged.
; This gets rid of errors on the boundary of the cells.
; (2) build the m1 boundary for the current cell.
;
; The hdrc boundary shapes were calculated just prior
to rules execution in the current cell.
cell_bnd = geomGetPurpose( "hdrc" "boundary" 0 0 )
cell_bnd_shrunk = geomSize( cell_bnd -0.5
check4 = geomAndNot( cell_bnd_shrunk mlbg
saveDerived( check4 "m1 coverage error"

; Derive the m1 boundary for the next level of hierarchy.
ml_bnd_next = geomOr( "m1" cell_bnd_shrunk )
saveDerived( ml_bnd_next ( "m1" "boundary" )

else
*/
*  Rules to run in flat mode.
*/
ml = geomOr("m1"
drc( ml sep < 0.5
*/
*  ... other flat checks ...
*/
.....

)
Simulation and Environment Control

The ERC and LVS programs both run under the control of the system simulation environment (se). When you start a job, the se takes control and performs a number of operations. These include running the system netlister if required, running the LVS or ERC program, and translating the output for probing and cross probing.

The netlister produces a text format netlist that provides the input to LVS and ERC. To run the netlister correctly, you must set up the netlist control environment. To understand this environment, read the Simulation Environment Help.

The LVS and ERC system defaults are set up to run the programs without needing to know about the operations of se.

Directories and Files

To enable the se to work automatically, you must provide a directory structure. For LVS, you do this by naming the directory in the LVS form or in the ivLVS command. For ERC, you do this by naming the directory in the ERC form or in the ivERC command. These programs then set up the directory and the necessary files and subdirectories.

The run directory for LVS contains two subdirectories, one for the extracted layout circuit data and one for the schematic circuit data. All files specific to a circuit are stored in its subdirectory. All other files are stored in the main run directory.

The LVS run directory contains the following files:

- **si.env**: Contains the se variables with their states as set up by the system or the Setup Environment form.
- **si.log**: Contains the simulation environment run log.
- **si.out**: Contains the simulation environment output.
- **layout**: Subdirectory for extracted layout data.
Subdirectory for schematic data.

The layout and schematic subdirectories contains the following files:

- **map**: Directory for both extracted and schematic cellviews containing the net and device mapping between the original hierarchical circuit and the flat netlist. The mapping files are in binary format.
- **netlist**: Text file for both extracted and schematic cellviews containing the netlist.
- **netxref.out**: LVS net cross-references.
- **devxref.out**: LVS device cross-references.
- **netbad.out**: LVS unmatched nets for both extracted and schematic cellviews.
- **devbad.out**: LVS unmatched devices for both extracted and schematic cellviews.
- **prune.out**: LVS nets and devices pruned from both extracted and schematic cellviews.
- **audit.out**: LVS nets and devices failing the parameter matching for both extracted and schematic cellviews.
- **ovfldev.out**: Overflow file used with devxref.out for cross probing parallel device groups.
- **termbad.out**: LVS unmatched terminals and matched terminals that have different I/O directions in the extracted and schematic cellviews.

The ERC run directory contains the following files:

- **si.env**: SE variables with their states as set up by the system or the ERC form.
- **si.log**: Simulation environment run log.
- **si.out**: Simulation environment output.
- **map**: Directory for both extracted and schematic cellviews containing the net and device mapping between the original hierarchical circuit and the flat netlist. The mapping files are in binary format.
### Variables

The `se` uses many variables to control LVS and ERC. You can set these variables from the menu forms or automatically through the system.

Although you do not normally change these variables, they are listed for reference. They are shown with their true default values or with dummy values where there is no specific default.

**Note:** In the following variable lists, the apostrophe (`'`) in `t` and `nil` is optional.

The following variables are contained in the `si.env` file:

```plaintext
lvsLayoutLibName = "lib"
lvsLayoutCellName = "fflatch"
lvsLayoutViewName = "extracted"
lvsSchematicLibName = "lib"
lvsSchematicCellName = "fflatch"
lvsSchematicViewName = "schematic"
lvsNetlistSchematic = 't
lvsNetlistLayout = 't
applyDeviceFixing = 'nil
correspondenceFile = 'nil
useFileCorrespondence = 'nil
useTerminalCorrespondence = 't
createXref = 'nil
simSimulator = "LVS"
simLibName = "lib"
simCellName = "fflatch"
simViewName = "schematic"
tmp/ff"
```

The following variables can be added to the `.simrc` file to override the defaults:

```plaintext
lvsRulesLibName = lvsSchematicLibName
simNlpGlobalLibName = "basic"
simNlpGlobalCellName = "nlpglobals"
disableLikeMatching = 'nil
disableReWire = 'nil
```
Control Input

The se uses several files. Although you never need to access most of these files, they are listed for reference. If you do need to make changes, see your system administrator.

- **.simrc**: Overrides the se variable default values. Resides in your home or execution directory.

- **caplib**: Contains files for controlling the se operation. Resides under /etc/skill/si in the cds system path. The file for ERC is ERC and the file for LVS is LVS.

- **sed.lvs**: File used by the program to translate the output files to a form usable by the system probing capabilities. Resides in /etc/si in the cds system path.

Setting Up the Netlister

Both LVS and ERC perform their main processing from netlists of the circuit. These netlists are produced by the system netlister program under the control of the se.

For netlisting of low-level devices such as transistors and capacitors, a special cellview of these devices called /vs is available in the system device library. This has the properties preset for the netlisting for both LVS and ERC.

Cellview Switching

Both the schematic and extracted cellviews of the circuit being compared use the same organization of cells and views. The schematic cellview organization is usually a multilevel hierarchy, where each level of cells refers to lower level cells, down to the lowest device level. The extracted cellview, even though it can be described as “flat,” has the same hierarchy, because the network cell refers to device cells placed within it.

Depending upon whether you are working on a schematic or a layout, you might want to reference different views of cells in the hierarchy. For graphic display, you can reference a symbol containing drawing information. For simulation, you need to reference a view containing the required simulation information. The system facilitates such references by having different views available for each cell. First, you must be able to tell the system how to find the correct cell within the computer directory system, and then you must tell the system how to determine which view to use.
In addition to knowing which view to use at each level of hierarchy, the netlister needs to know at which level of hierarchy to stop processing. A netlist for logic simulation needs to process the hierarchy down to the logic gates. A netlist for a circuit simulator, such as Cadence SPICE, needs to process the hierarchy down to the lowest device level, such as transistors.

Both the schematic and layout netlisting must stop at the same device definition. If you stop at an *n*\textit{f}et in the schematic, you must stop at an *n*\textit{f}et in the layout. Also, these two devices must use the same model with the same terminals in the same order.

Similarly, the netlist for the LVS and ERC programs must be processed to the level being analyzed, which might be the lowest device level, the macro cell level, or a combination of both. The view for the device does not need to contain graphic or symbolic information, but must contain formatting information to tell the netlister which form and information to generate. You can find further details in the “Netlist Formatting” section in this chapter.

To determine where to find specific cells, the program uses the system library path. To determine which views to use, the program uses “switching lists” and “stopping lists.” These variables are initialized to default values.

The switching and stopping lists are used as follows:

- When the master cell is found, the program searches the switching list and tries to match a view name in the list with the views available for that cell. The list is searched left to right, and the first matching view is used.

- The program then searches for this view in the stopping list. If the view is found, the expansion process stops for that cell. If it is not found, the program switches into the view, determines the instances within the view, and repeats the process.

The switching and stopping list variables and their default values are as follows:

**LVS**

```plaintext
lvsSchematicViewList = '( "lvs" "schematic" gate.sch "cmos.sch" )
lvsLayoutViewList = '( "lvs" "extracted" "schematic" "gate.sch" "cmos.sch" )
lvsSchematicStopList = '( "lvs" )
lvsLayoutStopList = '( "lvs" )
```

**ERC**

```plaintext
ercViewList = '( "lvs" "schematic" gate.sch" )
ercStopList = '( "lvs" )
```
To override these default values, you can provide a file in your home directory called `.simrc`. This file can already exist for specifying other information, such as the switching and stopping lists for SPICE netlisting. If the file does not exist, you can create it.

If having a single simulator control file in your home directory leads to conflicts, the same data can be put into a file called `.simrc` in the directory from which you started Design Framework II.

**Netlist Formatting**

The output of the netlister is controlled by format strings stored in cellviews as `nlpExpr` properties. The netlisting process is triggered by specific property names in the cellviews defined by the netlisting stopping list. In the case of LVS and ERC, these cellviews are the `lvs` cellviews.

The properties consist of netlisting instructions. Each instruction can print something in the netlist, go to another instruction property, or both. Each cell that is encountered in the circuit prints out the correct information in the netlist by using a sequence of instruction properties.

One source of the netlisting instruction properties is the cellview itself. Another source of netlisting instruction properties is a cell called `nlpglobals` in the system device library. Any netlisting instructions that are common to more than one device are stored in the `nlpglobals` cell.

The device library supplied by Cadence contains device simulation and global cells for all currently supported simulators. The LVS and ERC programs have their own `nlpglobals` cell called `lvs` and their own cellviews, also called `lvs`, for each of the lowest level devices such as transistors, capacitors, and so forth. If an `lvs` cellview does not exist for a specific device you need for LVS or ERC, you can either create your own `lvs` cellview or create an equivalent cellview with any other name. Use that name in the switching and stopping lists.

For LVS and ERC, the predefined `lvs` cellviews generate one line in the netlist for each device type and two lines for each instance of the device: one for comments and one for device usage. Additional lines are automatically generated for terminals and net names. The next sections detail the process of generating the device type and device instance lines.

**Device Type Definition**

Each different device type in the netlist must have a single-line definition of its structure. The `nlpglobals` cell contains a property called `NLPcreateModelString`. This property starts the sequence of operations that generates the device type definition. In LVS and ERC, this property references a property in the master cell of the device called `NLPModelPreamble`.
The master cell of the device is defined by the stopping list as previously described, for example, *nfet lvs*.

The *NLPModelPreamble* property, which determines the specific netlist format for a device, appears in all netlisted devices. Instead of putting the format inside the device cell, the *NLPModelPreamble* property references a model property (usually found in the *nlpglobals* cell). This allows multiple devices to have the same netlist format without duplicating information. The model property in *nlpglobals* has a name, such as *NLPMosfetModelCard*. This property provides the string to be written to the netlist. It contains a mixture of characters, written directly to the netlist, and substitutions, in which keywords are replaced by other information. There is a different model property for each device type having different netlisting information.

The substitutions used are as follows:

- **BlockName**. The system automatically places the cell name of the device in the netlist at this point.

- **permuteRule**. As shown in the example, if the property of *permuteRule* exists in the device cell, or in the *nlpglobals* cell, *permuteRule* is placed in the netlist at this point. If *permuteRule* does not exist, the default permute rule, following it in the property, is placed in the netlist.

The following is an example of device type formatting:

**Properties (in sequence)**

```plaintext
NLPcreateModelString = "[@NLPModelPreamble:%n]"
NLPModelPreamble = "[@NLPMosfetModelCard]"
NLPMosfetModelCard = "d [@BlockName] D G S B
[@permuteRule:%:(p S D)]"
permuteRule = "(p S D)"
```

Result in netlist

```plaintext
d nfet D G S B (p S D)
```

In this example, the *permuteRule* property on the device is the same as the default rule in the *nlpglobals* model property.

**Device Instance Definition**

Each different device instance appearing in the netlist must have at least one single line specifying its connections. It can also have an additional comment line.
The `nlpglobals` cell contains a property called `NLPcompleteElementString`. This property begins the sequence of operations that generates the device instance line in the netlist. In the case of LVS and ERC, it references a property called `NLPElementPostamble` in the device master cell. The master cell is defined by the stopping list as described previously (for example, `nfet lvs`).

The `NLPElementPostamble` property appears in all netlisted devices and determines the specific netlist format for that device instance. Instead of putting the format inside the device cell, the `NLPElementPostamble` references an element property (usually found in the `nlpglobals` cell). This allows multiple devices to have the same netlist format without duplicating information. The element property in the `nlpglobals` has a name, such as `NLPMosfetElementCard`. This property provides the string to be written to the netlist. It contains a mixture of characters, written directly to the netlist, and substitutions, in which keywords are replaced by other information. (There is a different element property for each device type having different netlisting information.)

The substitutions used are as follows:

- **ElementNumber** increments the index. Each device instance written to the netlist has a unique number.
- **BlockName** places the cell name of the device in the netlist at this point.
- Device terminals provide the net numbers connecting to the netlist.
- **LVSmosfetProp**. If this property exists in the device cell, it is placed in the netlist at this point. If it is not in the device cell or anywhere higher in the hierarchy, this property is taken from the `nlpglobals`. The `LVSmosfetProp`, in turn, contains its own substitutions which are a combination of property names and contents. In this case, the parameters of width (w) and length (l) as contained in the device instance are added to the netlist.

The following is an example of device instance formatting:

Properties (in sequence)
- `NLPcompleteElementString = "[@NLPElementPostamble:%\n]"
- `NLPElementPostamble = "[@NLPMosfetElementCard]"
- `NLPMosfetElementCard = i [@ElementNumber] [@BlockName] [D:%] [G:%] [S:%] [B:%] [LVSmosfetProp]"
- `LVSmosfetProp = " \" [@l:l %] [@w:w %] \"
- `w = 3`
- `l = 6`

Result in netlist
  
  i 26 nfet 26 57 17 87 " l 6 w 3"
Parasitic Devices

Normally, the schematic representation of a circuit does not contain parasitic devices. Any device referenced is nonparasitic (referred to as a “schematic” device). The extracted representation, on the other hand, can contain parasitic devices. These have usually been created from measured properties using a `saveParasitic` command and do not appear on the schematic.

If you try to make an LVS run match the extracted parasitics to the schematic, it fails. For LVS to be successful, either the extracted layout must not contain parasitic devices, or these devices must be ignored.

To overcome this problem, you can include special devices as parasitics in the extracted layout. These devices behave exactly the same as normal devices for SPICE simulation, but are ignored by the netlister for LVS (or ERC if so desired). This lets the program process only the schematic devices.

The `saveParasitic` statement (or `extractDevice` statement) used during circuit extraction must reference one of these special devices instead of a normal device, whenever that device does not appear in the schematic. The following special devices are available for this purpose.

- `pcapacitor` Equivalent to the normal device `capacitor`.
- `pdiode` Equivalent to the normal device `diode`.

Each special device has a `symbol` cellview, which becomes the default referenced in the extracted layout so that it can be viewed. Each special device also has an `lvs` cellview for the LVS netlister, and a SPICE cellview to permit full simulation. If any other cellview is required, you must copy it, without change, from the normal cellview of that device and add it to the cell.

Netlist Format

The netlister program produces a text format netlist. You can view this netlist to check the netlister control has performed as desired. To understand the format of the netlist, this format guide is provided.

This netlist format is generated by the system device library model entries using the `lvs` cellview.

The netlist is always flat, that is, all hierarchy is removed. Each line of the netlist defines an item. The items are as follows:

- Net name to number translation
Device type definition

Device instance definition

Terminal definition

Device fixing

Net Name to Number Translation

Defines the correspondence between the net numbers in the netlist and the net names in the original layout or schematic. This is optional in LVS.

```plaintext
net < net_number> [ = ] <net_name>
```

- **net**: This word introduces the net definition line. The word *net* can be abbreviated to the character `n`.

- **net_number**: The number of the net in the netlist.

- **=**: An optional character to indicate equivalence between net number and net name.

- **net_name**: The name of the net in the layout or schematic.

Device Type Definition

You must define each different device type used in the netlist. Use the following format

```plaintext
d <dev_type> <term_list> <perm> <property>
```

- **d**: Introduces the device definition line.

- **dev_type**: Name of the device type model in the system device library.

- **term_list**: List of terminal names that defines the order for net connections to the device.

  This order is arbitrary, unless you are reducing MOS transistors into gate configurations, in which case the order must be *source-gate-drain*.

  If the terminal names contain parentheses, for example “pin(1)”, the parentheses must be escaped by proceeding each with a
backslash ("pin\(a\))

perm

Defines device terminal permutability.

The definition has as two operators: \(f\) (fixed) and \(p\) (permutable).

A typical property for permuting a transistor source and drain is

\[
(p \ S \ D)
\]

An example of a combination fixed and permuted terminal definition is

\[
(f \ (p \ t1 \ t2) \ (p \ t3 \ t4))
\]

This means that terminals 1 and 2 are permutable, and terminals 3 and 4 are permutable, but the terminal pair of 1 and 2 is not permutable with the terminal pair of 3 and 4.

property

This property is a text string containing a series of name pairs that define the default properties of the device type. Each pair consists of the property name and the property contents. The following example contains two pairs, \(w\ 10\) and \(l\ 15\).

"w 10 l 15"

If the property value contains any special characters (including periods), the property must be included in quotes and each quote must be preceded by a backslash. For example

"w 12 l 7 code \"abc.def$\" "

Device Instance Definition

Each different instance of a device in the netlist must be defined with this line.

\[
i \ <inst_num> \ <device_type> \ <terminal_netlist> \ <property>
\]

i

Introduces the device instance line.

inst_num

A unique identification number for this device instance.

device_type

Name of the device type of which this is an instance.
**Terminal Definition**

Defines the terminals of the circuit. LVS uses terminal definitions to define terminal-based correspondence points and to define net connections that enable the *LVS prune* command to correctly handle inputs and outputs.

```
t < net_number > < net_name >
```

- **t** Introduces the terminal definition line.
- **net_number** Number of the terminal net in the netlist.
- **net_name** Name of the terminal net in the layout or schematic.

Terminal names that contain special characters such as “(" used for referencing busses, should be enclosed in double quotes. For example, “pin(1)”. The netlisting should be set up to always enclose terminal names in quotes, in case any name contains these special characters.

**Fixing Devices**

LVS uses two lines to define the devices are to be fixed or unfixed using the *fixing* feature. This only applies to the schematic netlist.

```
f < net_number_list >
u < net_number_list >
```

- **f** Introduces the fix line.
- **u** Introduces the unfix line.
- **net_number_list** List of net numbers used to determine which devices are fixed or unfixed.

"w 10  l 15"
Fixing applies to any device occurring in the netlist between the fix and unfix commands that reference a net to which that device is connected.

The fix and unfix commands are considered pairs. Each time you give a fix command you must also give a matching unfix command.

In the following example, the net 23 is defined twice. Devices connected to net 23 are not considered unfixed until both unfix commands containing net 23 are encountered.

```
f 23 45 56
f 84 23
.....
u 84 23
u 23 45 56
```

Example of Netlist

The following example illustrates the structure of a netlist. It is not meant to represent a real circuit.

```
d dep s g d (p s d) ; Depletion load device
d enh s g d (p s d) ; Enhancement device

t 1 gnd

 t 2 vdd
 t 3 a
 t 4 b
 t 9 clk

 i 1 dep 7 7 2 "l 3 w 2"
i 2 dep 5 5 2 "l 3 w 2"
i 3 dep 6 6 2 "l 3 w 2"
i 4 dep 8 8 2 "l 3 w 2"
i 6 enh 1 4 7
i 7 enh 1 3 7
i 8 enh 1 3 5
i 9 enh 1 4 6
i 10 enh 1 5 8
i 11 enh 1 6 8
i 12 enh 1 7 9
```
Application Techniques

Bipolar Processing

The Diva verification LPE program is technology independent. Most of the examples used in this manual are MOS because this is the easiest technology to understand and apply. This section introduces you to bipolar technologies which are more complex and require a slightly different application technique. The version of bipolar used in the example, though it might not exist, provides an easy-to-understand illustration that can be extrapolated to real technologies.

General Technique

The first step in the encoding of a bipolar technology is to determine all the devices you need to recognize. Within a single bipolar process, using a single set of mask or design layers, many different types of devices can be formed (for example, transistors, different kinds of diodes, resistors, and capacitors) as well as special contacts and ties. If you consider only a subset of these devices, the extraction fails because the layout includes devices not yet handled. To recognize those extra devices might require a complete redefinition of the original device recognition. For each device in the process, you need a “plan” view illustrating the topology relationships and a section showing the vertical layer relationships and current paths.

Minor topological variations change the type of device being represented. A Pbase region is performed as a single diode in one case, but the addition of an Ndiffusion (Ndiff) transforms it into a transistor. A contact in the Pbase is just a base contact until it overlaps into the Ntub, where it becomes a Schottky diode.

The major obstacle in bipolar recognition is the misinterpretation of which topological (layer/mask) relationships perform what functions. The first example shows the plan view of an npn transistor which is commonly misinterpreted. It appears obvious which is the emitter, base, and collector of this device, but in fact it is not.

The shapes in the diagram labeled E, B, and C are not terminals but contacts to interconnect layers that lead to the device terminals. The npn transistor is really two N-P diodes back to
back. The terminal labeled “base” is a contact to the Pbase region which forms the P region of the N-P-N configuration between the two diodes. It also acts as the conductor to the “outside world” from that device region. The terminal labeled “emitter” is in fact a contact to the Ndiff, which acts as a conductor down to the P region where the N-P interface forms one of the diodes.

The “collector” is a contact to another Ndiff, which, in turn, acts as a conductor and a contact to the Ntub of the device which also acts as a conductor to the interface between itself and the P region, forming the other P-N diode. The critical parts for recognizing the device are the N-P interfaces. The remainder of the structure just serves to interconnect those parts. This can be best seen from the sectional view, in which the diodes and contacts are highlighted, and from the extracted circuit diagram.

**Device Recognition Shape Determination**

This verification product’s LPE program requires a recognition polygon to represent the device. Since it is created vertically, there are three possible shapes that can be used. The Ntub seems to be the obvious choice until you consider that more than one transistor can be created within a single tub. Such devices have a common collector net, which is an example of how the tub needs to be considered as an interconnect layer to the collector rather than as the device itself.
What Makes an npn Transistor

If you choose the Pbase region as the device recognition polygon, the existence of multiple Ndiff regions within the base region gives rise to a multiple emitter device. This can in fact be what you want, but remember that each device recognized must be available in the system device library for it to be placed in the extracted cellview. The Ndiff inside the Pbase is the simplest polygon to use for device recognition. Each individual shape of diffusion creates a single three-terminal transistor.
If two or more diffusions are within the same P region, that region acts as an interconnect, giving them a common base region, and the tub acts as a common collector region. There is no difference between a multi-emitter device and multiple three-terminal devices with connected bases and connected collectors.

**Device Recognition Process**

After you have identified the functional parts of all the devices, encode the devices. The simplest approach here is one of elimination. Again, this is best illustrated by example.

Consider a Pbase region used in the definition of a number of different devices such as a diode, transistor, resistor, and so forth. Choose one of these having a unique set of layers. Maybe the only device in which Ndiff sits inside the Pbase is a transistor. Create the commands to recognize that device, and then “and-not” that specific Pbase with the whole to produce a subset of the layer. The transistor is eliminated and can be forgotten.

Now start afresh with the remaining devices. Most device topologies are unique in some manner or other. One by one, choose whatever is unique about the device and use that fact to recognize it, or a part of it. You can then eliminate the recognized features from the remainder. The word “device” in this case includes all structures, including contacts.

Some devices can appear to have no unique features. The terminals of a base resistor look exactly like vertical pnp transistors. Though the topology is identical, the terminals of the resistor form vertical pnp transistors, but they are parasitics and are usually ignored. Another common example is the distinction between the emitter and the collector of a lateral pnp transistor. Which is which depends on the external connections, not on the internal topology. You can distinguish between them, but the computer cannot.

To prevent confusion, add extra information to the layout that enables these devices to be recognized. You can use shapes on some “marking” layer, or you can use text information. In the former case, if the marking layer totally encloses the device, use the `geomAnd` function to pick out that device. If the marking layer is just some symbol which interacts with the device, use the `geomStraddle` function. In the latter case using text information, use the `geomGetTexted` function.

The elimination of recognized shapes from the remainder of the data to be processed applies only to structure recognition. The Pbase region referenced is used to define many device types, but it remains a single interconnect layer throughout. Try not to split connectivity layers into pieces. It must be done when a device such as a resistor breaks what is a contiguous net, but it need not be done anywhere else. In this example, the Ndiff is used to both recognize connections from the metal contact to the Ntub, and from the emitter contact to the N-P interface of the transistor. In both cases it remains the same interconnect layer.
**npn Transistor**

The introduction uses the *npn* transistor as an example, so the detailed explanation of its extraction can be omitted from this section. All that remains is the command syntax necessary to accomplish the recognition.

The simplest set of commands you need to recognize a single transistor is as follows (the line numbers are added for annotation purposes and must not appear in the final command input stream).

```plaintext
1   npn = geomInside( Ndiff Pbase )
2   contc = geomAndNot( Ndiff Pbase )
3   contb = geomAndNot( Contact Ndiff )
4   contn = geomAnd( Contact Ndiff )
5   geomConnect(via( contn metal ndiff )
    via( contc ndiff buried )
    via( contb metal Pbase )
6   extractDevice( npn Ndiff ("E") Pbase ("B") buried ("C") "npn" )
```

Lines 1 and 2 split the Ndiff into two functions, one forming devices and one forming collector contacts. The contacts are similarly split (lines 3 and 4) into those connecting to the base and those connecting to the Ndiff. The `geomConnect` statement (line 5) ensures that all layers through which current flows are correctly connected. Note that the “buried” layer was used as the interconnect to the collector instead of to the Ntub. In terms of connectivity (parametric characteristics aside), they perform the same function. Usually, both must be present underneath the device; either can be used. There are occasions, when separate tubs are connected by a single buried layer, in which case the buried layer must be considered as interconnect.

In some technologies, a good path from the collector contact to the buried layer is ensured by a separate deep diffusion, in addition to the Ndiff, under the contact. In these cases, no change needs to be made to the commands other than to add a line which checks that every `contc` has a related deep diffusion.

Line 6 defines the transistor itself. In this example, the command is written in full, but it can be shortened by accepting the defaults for the terminal counts and the model name. The three layers referenced as terminal connections must have been previously referenced in a `geomConnect` statement. The model name *npn* must exist in a system device library, and the terminal names referenced must match those in that library device.

To simplify the example, and because only one device type is considered, no attempt has been made to validate the layers used. For example, the Pbase must be inside the Ntub for it to form a transistor, and the Ntub must have a buried layer underneath it. No check has been
made for contacts connecting directly to the Ntub without the interposition of Ndiff. If they existed in the technology, these do not form Schottky diodes; they must be flagged as errors.

**pnp Transistor**

The *pnp* transistor recognition uses exactly the same techniques as described in the general technique for the *npn* transistor, except that the example used cannot be applied literally. The example illustrates the construction of the *pnp* transistor in the same way that the previous example illustrated an *npn* transistor.

**pnp Bipolar Device**
The *pnp* transistor is different from the *nnp* transistor in two major respects. The first
difference is the distinction between the emitter and the collector. Although, in most cases,
the human eye can differentiate between these two terminals, the computer cannot. The
differences are one of pattern recognition rather than topological relationship. When both
terminals are the same size and shape, they can only be differentiated by tracing the circuit
and determining their relative potential, which the computer cannot do.

To distinguish one terminal from the other, you must supply additional information on the
layout. You can add a polygon on a special layer to mark one or other of the terminals.

```plaintext
emitter = geomAnd( Pbase masklayer )
collector = geomAndNot( Pbase masklayer )
```

or you can add text

```plaintext
emitter = geomGetTexted( Pbase "textlayer" "emit" )
collector = geomAndNot( Pbase emitter )
```

Because it is usual to create standard bipolar devices in a library, it is only necessary to mark
each device master once, rather than marking each instance of the device in the layout.

The second difference between the two transistors is that the *pnp* terminals have a lateral
relationship, and the *nnp* terminals have a vertical relationship. This can best be seen in the
previous example.

In this example, the buried layer can be used to represent the device, in which case the result
is a two-collector *pnp* transistor. Regardless of the number of collectors, a single transistor is
recognized. Each permutation of collectors must be represented in the system device library
for the extraction to be performed successfully. Problems arise when multiple transistors are
created in a single tub. The designer considers such transistors as separate devices having
a common base, whereas the program device recognition using the buried layer generates a
single device having multiple emitters, and collectors which also appear in the library.

To overcome this problem, use one of two solutions:

- Create another layer for device recognition so that multiple transistors inside a single tub
  are recognized as such. Each transistor can have multiple collectors.

- Use the collector layer for device recognition resulting in multiple single-collector
  transistors.

In the first solution, you need a layer that currently does not exist. In the second solution, you
need a way to relate the collectors laterally to the emitter.

The same basic technique is used for both solutions. Create a layer which is the oversize of
the emitter. Use enough sizing to ensure that the resultant shape interacts with the collectors.
The sequences of operations for both solutions are
For the first solution

1..emit = geomGetTexted( Pbase "text" "emit" )
2. Pbasei = geomAndNot( Pbase emit )
3. pnp = geomSize(emit 3 )
4. contb = geomAndNot( Ndiff Pbase )
5. contn = geomAnd( contact Ndiff )
6. contp = geomAnd( contact Pbasei )
7. conte = geomAnd( contact emit )
8. geomConnect(via ( contn metal ndiff )
   via ( contb ndiff buried )
   via ( contp metal Pbasei )
   via ( conte metal emit )
9. extractDevice( pnp emit ("E") Pbasei ( "C" "C" )
   buried ("B") "pnp-2" )

In this example, the oversized emitter is used directly as the device recognition polygon. It interacts, forming collector terminals, with all Pbase that is within the sizing distance of the emitter. In this case, the distance chosen was 3 units. Any Pbase region beyond that distance is not considered as a collector of that device.

Lines 1 to 3 identify the emitter, remove it from the Pbase interconnect, and size it to produce the device recognition polygon. Lines 4 and 5 recognize the two contacts to the buried layer in the same way as for npn. But in this case, the connection is to the base rather than to the collector. Line 6 recognizes the contact to the general Pbase interconnect Pbasei, which in this example represents only the collectors, though it can be used as general interconnect. Line 7 recognizes the contact to the emitter. The geomConnect statement in line 8 ensures the correct connections from metal to all the other interconnect layers. The device statement in line 9 is structured in exactly the same way as the npn device except that one terminal (the collector) has two terminals.

The model name references a device which has two collectors. If other pnp transistors in the circuit have other numbers of collectors, then a separate device statement needs to be used for each to identify the correct library model. There is no processing penalty for having multiple definitions of a single device since all devices with the same recognition layer are processed in a single pass of the data.

For the second solution

1. emit = geomGetTexted( Pbase "text" "emit" )
2. Pbasei = geomAndNot( Pbase emit )
3. bridge = geomSize( emit 3 )
4. pnp = geomStraddle( Pbasei bridge )
5. contb = geomAndNot( ndiff Pbase )
6. contn = geomAnd( contact ndiff )
7. contp = geomAnd( contact Pbasei )
8. conte = geomAnd( contact emit )
In this example, the oversized emitter polygon is used as the emitter terminal of the transistor and as a connectivity bridge from the emitter terminal to the emitter Pbase, which defines the connectivity to the metal.

Lines 1 to 3 identify the emitter, remove it from the Pbase interconnect, and size it to produce the emitter terminal. Lines 4 to 9 are similar to the previous example and define the contacts and connectivity. Line 10 defines the device, differs from the previous example, and requires one collector terminal. The library model referenced is the “standard” three-terminal pnp transistor. Since all pnp devices recognized by this definition has a single collector, this is the only model required in the library. This differs from many models (two collector, three collector) required by the previous example.

Resistor Terminal Recognition

Normally, the specification of device terminals in the extractDevice statement is simple. The layer forming the terminal is defined, and the program automatically forms a terminal from any relationship between the device recognition polygon and that layer.

The specification of resistors and some other devices is not as simple. The two problems are as follows:

- The two ends of a resistor can be terminated in different ways which need different terminal layers. Any single resistor can have any combination of those terminal types.
- A resistor can be terminated by a contact to an interconnect layer.

The first problem occurs because a resistor can be terminated in several ways, such as butting an interconnect layer of the same material as the resistor, contacting interconnect metal with a normal metal contact cut, and in the case of a polysilicon resistor, contacting diffusion with a buried contact. Any resistor can have any combination of these terminations, so specifying a “poly interconnect” terminal does not find both terminals of a resistor with one end contacted to metal.

The next example illustrates three ways of terminating a resistor.

The second problem involves the operation of the geomConnect statement. A terminal of a device must be an interconnect layer. This means it must have been defined in a
The `geomConnect` statement, which relates a series of interconnect layers by a contact layer, or by a function which propagates that connectivity. If a resistor is formed from the polysilicon layer, the shape forming that device must be removed from that layer to create the interconnecting polysilicon.

**Resistor Terminations**

![Diagram of resistor terminations](image)

All resistors are shorted out by the layer that forms them. In this case, the contact has no polysilicon interconnect to connect to and is badly formed. In addition, there is no interconnect layer to form the terminal. If the metal was used, all metal crossing resistors is connected to the resistor.

The solution offered here is not necessarily the only one, but it does solve both problems at the same time.

The contact structures are separated from the resistor structure, where a dummy shape is generated on the interconnect layer to act as the resistor terminal. This is best described by an example. Consider the case of a resistor which can have its ends terminated by a metal contact or by butting polysilicon interconnect. The resistor is differentiated from the interconnect by a text string of `res` on the layer `textlayer`. This example does not consider “buried contacts.” The line numbers in the following commands are for reference only and are not part of the file.

```plaintext
1  resistor = geomGetTexted( polysilicon "textlayer" "res" )
2  polyr = geomAndNot( polysilicon resistor )
3  stubs = geomAnd( resistor contact )
4  polyi = geomOr( polyr stubs )
5  geomConnect( via( contact polyi metal ) )
```
Line 1 identifies the resistor from among the polysilicon shapes and line 2 removes it from that layer, leaving polyr. Line 3 copies all contacts that are over the resistor onto layer *stubs*. Line 4 merges these stubs with the main polysilicon, polyr, to form the final polysilicon interconnect layer, polyi.

The *geomConnect* statement in line 5 can now successfully make a connection through the contact that was over the resistor. The connection is from one part of the polyi, which was polyr to the other part of polyi which was stubs. The connectivity is now complete without involving the resistor. Both types of terminal on the resistor now relate to the polysilicon interconnect layer polyi, one through direct butting and one through overlap with the part of the interconnect that was derived from the contact.

Line 6 recognizes the device with just a single terminal type. If “buried contacts” had been involved also, additional lines can be added to treat that contact in exactly the same way as the metal contact, leaving the device statement exactly as shown.

The next example illustrates the layer relationships involved for the metal contact to the resistor.

**Contacting a Resistor**
Extraction Device Library Entries

The device recognition facility in this verification product’s LPE program requires the definition of a device model. For each device recognized, an instance of that device is placed in the extracted cellview. The cellview type of the device instance is `symbol` by default. If that extracted cellview is going to be used within iLVS, a cellview must be available for that device to correctly netlist. The choice of which cellviews to use for the netlist is achieved through the SI switching and stopping specifications. You might be able to use the SPICE cellviews for this purpose. In many cases, they netlist correctly for iLVS; in some cases, they do not. In the case of schematics that contain the device “cap,” the switching is incorrect for iLVS because a cap has a SPICE cellview but contains an instance of a capacitor, which also has a SPICE cellview.

The safest cellview type for iLVS is `lvs`. Each of the low-level devices in the system device library has an `lvs` cellview. The netlisting properties of these cellviews are compatible with the iLVS nlpglobals cellview (called `lvs`) and netlist correctly. The SI switching and stopping lists for the following cellviews needs to be defined.

```plaintext
simNlpGlobalLibName = "basic"

lvsLayoutViewList = list( "lvs" "schematic" "symbolic" )

lvsSchmaticViewList = list( "lvs" "schematic" "gate.sch"
                          "cmos.sch" )

lvsLayoutStopList = list( "lvs" )
lvsSchematicStopList = list( "lvs" )
```

The next example illustrates the symbol representations for the devices having `lvs` cellviews. The `pmos` and `nmos` devices are schematics that reference the `nfet` and `pfet` devices, are shown here for clarity, and are considered low level. Note that they have their back-gate connected to ground and power respectively. They must not be used when their back-gate is connected to any other signal.
Device Library Entries

- **Capacitor**
- **Diode**
- **Resistor**
- **Inductor**

- **N-FET** (n-channel enhancement mode MOSFET)
- **N-FET** (n-channel depletion mode MOSFET)
- **P-FET** (p-channel enhancement mode MOSFET)
- **P-FET** (p-channel depletion mode MOSFET)

- **NPN Transistor**
- **PNP Transistor**
- **NPNBPS** (n-channel power MOSFET)
- **PNPBPS** (p-channel power MOSFET)

**Symbols:**
- PLUS
- MINUS
- D
- G
- S
- C
- E
- SUB

**Notes:**
- Capacitor
- Diode
- Resistor
- Inductor

**Device Types:**
- N-FET
- P-FET
- NPN
- PNP

**Technology:**
- PMOS
- NMOS
Substrate, Background, and Ground

This section clarifies the concepts of substrate, background, and ground with regard to the connectivity of a circuit being extracted.

Concepts

The graphics editor has a layer called bkgnd that identifies the complete area of the circuit. The layer color forms the background to the layout, and, when active, the layer can be used to select shapes on all layers. This verification product’s LPE program uses the same name, bkgnd, to represent a single polygon whose size encompasses all other shapes in the layout. The background layer can be manipulated as any other graphics layer, but does not have to be created by you.

Substrate is used to represent the physical material underlying the circuitry of the chip. Generally, the substrate is considered to completely encompass the physical circuit and pass beneath all structures. In some technologies, the substrate is subdivided into different areas to represent different electrical functions. For example, in CMOS, the substrate can be divided into well and substrate.

Ground is a name applied to those parts of the circuit which conduct current at ground potential (which can be whatever potential you declare it to be).

Relationships

The main purpose of this verification tool’s LPE bkgnd layer is to represent the substrate of the circuit in cases where the substrate takes part in the electrical connectivity. It can be subdivided, as in the CMOS example, by the normal layer manipulation functions geomAnd, geomAndNot, and so forth. In some cases, the substrate can be connected to ground, so the bkgnd becomes part of the ground net. In this case, bkgnd, substrate, and ground are synonymous. Substrate can be connected instead to any other net or potential of the circuit. Except in those cases where functions are used to subdivide the bkgnd into separate pieces, any nets of the circuit connected to it (via a geomConnect statement) are shorted together.

A measureParasitic statement, generating parasitic devices can use bkgnd as a terminal for the parasitics. This is accomplished by using the layer bkgnd in this way.

\[
\text{cap} = \text{measureParasitic}(\ \text{area}( \text{polysilicon over bkgnd over aluminum}) \ \text{two_net})
\]

The bkgnd must be connected. Although both polysilicon and aluminum are interconnect layers, the polysilicon and the bkgnd are used as the device terminals because they are the first two interconnect layers from left to right in the statement. If the bkgnd is connected to the
circuit by a contact (such as a substrate contact), then one terminal of the parasitic devices is connected to that net. If there is no real connection to bkgnd, then all the parasitics are connected together to a common net. This net has no connection to the remainder of the circuit and is totally independent of any net in the circuit.

This measureParasitic statement has a different result.

\[
cap = \text{measureParasitic}( \text{area (polysilicon)} \text{ two\_net } )
\]

In this example, there is no second interconnect layer to which the parasitic device can be connected, so the program generates a new, unique net to connect to one of the parasitic terminals. One terminal is connected to the polysilicon net and the other to a single net, which is common to all parasitic devices generated this way. This unique net is not, and cannot be, connected to the remainder of the circuit. It is independent of any net in the circuit.
Symbols
..., in syntax 15
... in syntax 15
/ in syntax 15
[] in syntax 14
{} in syntax 14
| in syntax 14

A
abutment 155
active layers 582
algorithms, LVS 496
alignment modifiers 272
AnalJobMonitor 571
analyzing parameters 497, 507
angles
  limit 204
  range 204
antenna checks 286
app command 302
apposition 303
arcs 138
area command 292
area mode, in DRC 257, 258
area processing 261
area, specifying 74
arrays 260
attachParasitic command 411

B
backannotation 498, 536
  executing 536
  limitations 537
background, running Diva in 28
batch mode 33
  prerequisites 20
bends 351, 370
  measurement 362, 369
  resistance 429
bipolar devices and processing 627
Bipolar rules file 109
black box 326
blocks 616
Boolean functions
  edges 135
  example 103
braces in syntax 14
brackets in syntax 14
building logic gates 455

C
Cadence environment 28
Calculate command 368
calculateParameter command 348, 354
  relation to measureParameter 358
calculateParasitic command 373
  calculation functions 385
  figure measurements for attachParasitic 411, 412
  relation to measureParasitic 391, 400
  relation to saveParasitic 415
Cancel button
  on options forms 26
  on standard form 25
capacitance
  junction 418
capacitance, distributing evenly 434
capLimit (optional argument) 447
cell exclusion 60
cell inclusion 60
Cell-Based Options command 277
cells
  adding run mode property 39
  changed 258, 330
  defining macros 127
  ignoring 60
cellviews 345
extracted 560, 563
schematic 559, 563
change layer 258
changedLayer 258
changeLabel command 112, 113
overview 249
check area 541
check point
creating 43
directory 87, 90
checkAllLayers command 279
checkConnected command 452, 473
checkDeviceNetCount command 452, 474
checkFanOut command 452, 476
checkFloatingDevices command 452, 479
checkFloatingNets command 452, 480
checking
area 258
by properties 279
complex 271
data integrity 139
design rules 539
dimensional 329
electrical rules 549
enclosure 17
flat 543
  DRC form 540
  ERC form 551
  Extract form 548
  LVS form 562
full 258
hierarchical 259
incremental 258
layer 281
networks 18
checking limits
by area 541
full 540
incremental 540
checkLayer command 281
checkNotConnected command 452, 481
checkOneNetDevices command 452, 482
checkOneTerminalNets command 452, 483
checkPoint command 43
used by ivRestart 87
checkPullDown command 452, 484
checkPullUp command 452, 485
checkPullUpAndDown command 452, 486
checks, electrical 549
circuit netlist 450
CMOS rules file 107
coincident segments 212
combining shapes on input layers 148
commands
app command 302
area command 292
attachParasitic command 411
Calculate command 368
calculateParameter command 354
calculateParasitic command 373, 391
Cell-Based Options command 39, 277
changeLabel command 112, 113
checkAllLayers command 279
checkConnected command 452, 473
checkDeviceNetCount command 452, 474
checkFanOut command 452, 476
checkFloatingDevices command 452, 479
checkFloatingNets command 452, 480
checkLayer command 281
checkNotConnected command 452, 481
checkOneNetDevices command 452, 482
checkOneTerminalNets command 452, 483
checkPoint command 43
checkPullDown command 452, 484
checkPullUp command 452, 485
checkPullUpAndDown command 452, 486
compareDeviceProperty command 507, 509, 526
compareFet command 108
compareRes command 110
complexParasitic command 377
copyGraphics command 239
diffNet command 304
DisplayErrors command 510
displayFeedbackLoops command 453, 487
displayGateErrors command 453, 488
drc command 539
drcAntenna command 286
parallel command 311
parallelMOS command 108, 109
parallelRes command 109
parameterMatchType command 528
permuteDevice command 508, 530
Probe command 573
pruneDevice command 505, 533
raw command 325
reduceDevice command 453, 460, 489
removeDevice command 535
sameNet command 312
saveDerived command 243, 328
saveInterconnect command 332, 345
saveParameter command 358, 364
saveParasitic command 391, 400, 415
saveProperty command 346
saveRecognition command 332, 333, 347
sep command 298
seriesRes command 110
setGround command 451, 468
setInput command 451, 469
setOutput command 451, 470
setPower command 451, 471
shielded command 313
squareGrow command 323
testDeviceProperty command 464, 493
testGateProperty command 464, 495
twoWayPath command 452, 472
Verify menu
  DRC command 539
  ERC command 549
  Extract command 545
  LVS command 556
  Probe command 573
verifyArea command 111, 129, 258
width command 300
with_perp command 315
compareDeviceProperty command 507, 509, 526
comics 135
check 142
connections
  check one connection to circuit 483
  connected layers 304
  contact connection 246
  counting 452
  forming 246
gemConnect command 251
  ground 468
  output 470
  power 471
to specific net 452
connectivity
  checks 18
defining 251
extraction 545
  extraction limitations 247
  pin connectivity 329
  selecting shapes from net 219
contact resistance
  area resistance 426
  resistance models 427
conventions
  user-defined arguments 14
  user-entered text 14
copy
  data 239
  shapes 345
copyGraphics command 239
corners, dead 430
correspondence points 84, 498
defining 516–518, 562
file 517
form 566
improving error isolation 496
terminal names 517
cross coupling and fringe capacitance 431
cross-probe 332, 574
currentCell switch name 266
Diva Reference

D

data
  integrity checks 139
  manipulation 211
dead corners 430
default error region 267
Defaults button
  on options forms 26
  on standard forms 25
deleting
  layers 241
derived layers 97
  definition 132
  example 100
  in DRC 271
  saveDerived command 243
  saving 265
Design Framework II 21
Design Framework II environment 21, 512
designed device 326
device recognition 331
  layer 346, 347
  shapes 347
devices
  combining 508
  comparing 509
  comparing parameters 450, 496
  connected to nets 474
  designed vs. parasitic 326
  displaying 332
  extraction 545
  extraneous 505
  fixing 84, 498, 515
    requirements 515
    schematics 516
    turning on/off 504
  floating 479
gates 456, 462, 501
  ignoring 516, 533
  ignoring for netlisting 497
  ignoring terminals 497
  instances 66
    creating devices 412
  measured values 411
  measurements, saving 364
  MOS 456, 462, 501
  not connected 481
  pairs 457, 501
  parallel 455, 499
  parameters 493, 516
  parasitic 621
  permutability 459, 515, 530
  probing 573
  properties 493, 516
  pruning 505, 533
  reducing 454, 508
    AND and OR reduction 462
    circuit netlist 489
  removing 505, 507, 535
  series 455, 499
  simplifying 530
  terminals 474
  unmatched 505
diffNet command 304
diffNet option 139
dimensional checks 329
disk space reduction 327
display layers 582
displaying
gate errors 488
  loops 487
  nets 332
distribution of capacitance 434
Diva
  prerequisites 20
  product flow 19
  products 17–18
  running 28
    batch mode 33
    interactively 30
    remotely 34
  running remotely 34
donut holes 236, 237
Diva Reference

Dracula to Diva Translator 591
DRC
  conditional execution 45
  enclosure checks 270
  form 540
  grouping commands 95
  hierarchical
    improving performance 608
log file 115
looking through the wall 270
operation modes 257
  area 258
  flat 259
  full 258
  hierarchical 259
  incremental 258
properties 285
restarting 87
running 543
setting options 540, 543
spacing checks 270
starting from UNIX 89
starting using SKILL function 73
DRC command 539
DRC/Extract
  command 539
  processes and products used 19
    rules file sample 99
drc? switch name 45, 100
drcAntenna command 286
drcCreateLogFile command 41
drcExtractRules() command 95, 96
drcLogFile command 115
drcZeroHalo command 117
dubiousData command 137, 140

E
  edge booleans 135
  edge format, defined 134
  edge keyword (optional) 198
  edge modifiers 317
  edge segments 211
  edges
  adjacent 202, 268
  angle 204
  apposition 302
  butting 183, 187, 212
  coincident 167, 187, 212
  contiguous 215, 216
  different net 304
  DRC checking 266
  facing 293
  geomGetAngledEdge command 204
    inside 212
    inside/outside 266
    length 215, 305
    limit 134, 211
    not parallel 307
    outside 212
    parallel 311
    perpendicular 308, 315
    reducing 200
    relating to other edges 211
    same net 312
    selecting 204, 211
    selecting non-parallel edges 221
    shapes 133
    stretching 200
  empty layers 132
  enc command 293
  enclosed shapes 173
    limits 163, 174
  enclosure
    checks 17, 270
    limits 293
    measuring 293
  environment
    Cadence 28
    initialising 565
    UNIX 28
  environmental variables 58
  ERC 450–466
    defining connections 451
    displaying errors 573
    error output files 453
    error report 551
    explaining errors 556
Diva Reference

form 550
generating netlist 550
invoking 549
log file 551
monitoring jobs 551, 554
netlist file 551
network reduction 454
output file 551
probing devices and nets 573
processes and products used 19
running 552
setting options 550
showing run information 551
  form 555
  using 555
starting using SKILL function 77
ERC command 549
ercRules() command 105
error layer 271
error report, displaying for ERC 551
errors
  explaining 324
  finding 257
excell 329
exclusive option 138
expanded mosaics 39, 277
explaining errors 324
Extract
  command 545
  conditional execution 45
  flat 546
  form 546
  grouping commands 95
  incremental hierarchy 546
  macro cell 91
  restarting 87
  running 548
  setting options 546
  starting from UNIX 89
  starting using SKILL function 79
  using 548
extract? switch name 45, 102
extractDevice command 337
device recognition 331
device recognition
  bipolar technology 627, 631
  fracturing lines 446
  hierarchical ??–330
  incremental 330
  macro cell 91
  restrictions 329
  tools 328
  version of circuit 331
  view of circuit 62, 326, 345
extractMOS command 341
  device recognition 331
  ivpcel command 332
  measuring parameters 348
  optimization 371
  parasitic measurement commands 365
  relation to attachParasitic command 412
  relation to saveRecognition command 347
extraneous device removal 497

F
feedback loops 453, 487
fig modifiers 320
file mapping
  automatic 37
  manual 36
  Remote Diva 35
files
  .simrc 520
  audit.out 510, 521
Diva Reference

chip.data 82

correspondence points 517

layout netlist 572

line 90

line, viewing 87

lvs.cpoint 517

minEnclosure 281

minNotch 281

minOverlap 281

minSpacing 281

minWidth 281

myCorr 85

myrules 76, 82, 93

netlist 454, 522

nlpglobals 507

prune.out 521

rsf 80, 90

run-specific commands 542, 547

schematic netlist 572

si.log 454, 522, 555

si.out 454, 522, 555

xref.out 85

fill patterns 582

finding vertices 142

flat mode (device extraction) 326

flat mode (DRC) 257, 259

fly-lines 332

fnlSearchPropString 437

foreground, running Diva in 28

forms

Add Correspondence Points 566

Analysis Job Monitor 571

Correspondence Points 566

Display Specific Correspondence Points 568

DRC 540

ERC 550

Extract 546

Initialize Environment 565

LVS 558

LVS Error Display 569

LVS Form Contents Different 558, 563

options forms 24

Probing 574

Remove Correspondence Points 567

standard forms 24

using 25, 30, 32

fringe capacitance 431

deriving a single measurement 383

fringe measurements

calculating 384

defining a relationship 388, 409

deriving a single measurement 383

fringe capacitance 383

rejecting measured values 388, 409

sidewall capacitance 383

specifying limits 388, 409

storing results 383

terminal 341

generating geometric shapes 233

gates

building 455

creating 489

displaying errors 488

errors 453

forming 453

parameters 453

creating 495

testing 495

properties 495

reduction 464, 495

reduction example 464

gemLineEnd 178

G

geomAnd command 144, 328

geomAndNot command 146

geomAvoiding command 156

geomBkgnd command 234

geomButting command 155

geomButtOnly command 158

geomButtOrCoin command 161

geomButtOrOver command 164

geomCat command 97, 142, 148
geomCoincident command 167
geomCoinOnly command 170
geomConnect command 48, 103, 251
device recognition 331
establishing connectivity 246
limitations 247
multiple via sections 247
one via section 248
geomEmpty command 235
geomEnclose command 173
geomErase command 241
geomGetAdjacentEdge command 202
geomGetAngledEdge command 204
geomGetByLayer command 208
geomGetEdge command 211
geomGetHoled command 214
geomGetLength command 215
geomGetMacro command 217, 328
geomGetNet command 219
geomGetNon45 command 221
geomGetNon90 command 222
geomGetPolygon command 223
geomGetPurpose command 224
parasitic measurements 329
specific shape selection 131
geomGetRectangle command 226
geomGetTexted command 227
geomGetUnTexted command 230
geomGetVertex command 232
geomHoles command 236
geomInside command 176
geomLineEnd function 178
geomNoHoles command 237, 328
geomNot command 149
geomOr command 150
geomOutside command 183
geomOverlap command 184
geomSize command 190
geomSizeInTub command 193
geomStamp command 255
geomStraddle command 187
geomStretch command 200
geomXor command 152
globalLabel command 112, 118
overview 250
graphics layers, original 131
ground net, defining 120
groundNet command 120
overview 250
grouping commands 96
grow. See sizing 191

H
halo around checking area 117
halo value 261
hdrc boundary layer 608
hdrc property 39
Help button
   on options forms 26
   on standard forms 25
Hide button, on options forms 26
hierarchical mode, in DRC 257, 259
   limitations 262
hierarchical optimization 260
hierarchical processing, overview 329
hierarchical programs
   DRC 17, 257, 259
   LPE 18
hierarchy
   analysis 74
   design methodology 329
   extraction ??–330
holes 236, 237
holes in rectangles 129

I
if-then-else for ERC and LVS 105
ignoreTerminal command 504, 527
ignoring cells 60
ignoring device terminals 497
inactive layers 582
inclusion/exclusion 74
incremental analysis 74
incremental hierarchical mode (device extraction) 330
incremental mode (DRC) 257, 258
Inherited Connections 572
initializing environment 565
input layers, combining 148
inside of shapes
   shapes inside other shapes 176
inside/outside 266
installation instructions 14
interactive mode 28, 30
   prerequisites 20
interconnect 332
inverting shapes 149
ivCallProc command 54
ivCompilingFor 57
ivCreatePCells command 71, 333, 334
ivDRC command 39, 73, 112, 277
ivERC command 77
ivExtract command 79, 112, 330
ivIncludeValue property 61
ivlf command 45, 96
ivLVS command 83
ivpcell command 332
ivRestart command 87
ivVerify command 39, 89, 277

J
jobs
   checking DRC progress 545, 549
   monitoring 551, 554
   priority 85
joinableNet command 112, 122
   overview 249

K
keywords 14

L
labels
   adding 247
   adding to text layer 124, 126
   changing 249
   changing names 113
   label origin location 253
   pin names 249
   placement 581
   rules 253
   treating as if on top-level cell 118
layer property commands 97
layers 281
   active/inactive 582
   conducting 251
   connected 132
   contact 246
   definitions 345
   deleting 241
   derived 132, 251
   edge 200
   empty 132
   erasing 241
   instance 334
   interconnect 246, 252
   keeping unmerged 325
   label 253
   original graphics 131
   polygon 200
   purpose 131
   saving 238
   terminal 337
   text 124
layerText command 111, 124, 126, 249
layouts, comparing 496, 556
length
   limit 215
   measuring 369
   range 215
length commands 305
Library Browser 551, 560
lines, flagged as warnings 138
locating shorts 582
log file
   creating 41
   log file, creating 115
   log file, displaying for ERC 551
   logical functions 143
   logical operations, implied 352
   looking through the wall 270
   LPE, hierarchical 18
   LVS 621
algorithms 496
backannotation, executing 536
command 556
correspondence points
  adding 566
  displaying 568
  removing 567
defining extracted cellviews 560
defining schematic cellviews 559
displaying errors 523, 569, 573
  merged nets 523
  pruned devices 525
  pruned nets 525, 570
  unmatched instances 523, 570
  unmatched nets 523, 570
  unmatched parameters 524, 570
  unmatched terminals 524, 570
explaining errors 570
form 558
Inherited Connections 572
invoking 556
options 561
outputs 454, 522
probing 562, 566
probing devices and nets 573
processes and products used 19
running 562
setting options 558
setting search paths 565
showing run information 562
starting using SKILL function 83
using 562
LVS command 556
LVSAddCorrPoints 566
LVSCorrPoints 556
LVSDspCorrPoints 568
LVSRemCorrPoints 567
lvsRules( ) command 105, 512
LVSShowRunInfo 572

M
macro cell mode (device extraction) 62–66, 326
macro cells 520
cell boundary 62, 326
creating file of cell names 127
defining 62
extraction 65, 91
g geomGetMacro function 217
hierarchy, controlling 65
ivCellType property 328
LVS comparison 516, 520
processing 62
macroCellFile command 127
marker layer 142
markers
  creating log file of shapes 41
  listing all shapes 115
mask misalignment, simulating 322
mask registration displacement, simulating 323
master cell 259, 334
matching nets 18
Measure command 368
measured figures, association with device figure 413
measureFringe command 365, 368, 383
measurement commands 349, 371
  consolidation criteria 353
measurement values
  creating 373
  defining property names 412
  higher than real value 430
  manipulating 373
  mathematical operators 373
  mathematical precedence 374
  mixing measurements 374
  rejecting calculated values 375
  sharing values 413
  storing 373, 413
measurements
  bends 350
  calculator 350
  changing square microns to farad capacitance 395
  optimization techniques 352
  parameter 350
  parasitic 350
  saving 364
Diva Reference

SPICE requirements for area 395
measureParameter command 358
  device characteristics 348
  example 104
  relation to calculateParameter command 354
measureParasitic command 391
  measurement process 368
  name limit 385
  prerequisite to attachParasitic 412
  prerequisite to saveParasitic 415
  relation to calculateParasitic command 373
measureResistance command 420, 435
  output layer 431
menus
  general description 23
  slider menus 23, 31
  Tools 30, 31, 32
  Verification 31, 32
  Verify 23
    from a layout window 30
    from a schematic window 32
merged data, defined 133
merged output 255
  shapes 150
merging all shapes 150
message command 324
models, defining 342
MOS
  gate measuring 370
    grouping 457, 501
    reduction 456, 462, 501
mouse, using 22
multi-edged polygons 135
multiLevelParasitic command 400
multiplication factor for coupling effects 418

N
negative sizing 190
nesting commands 97
netlister
  Diva support 21
  schematic 520
  spice 520
netlists
device type definition 84
device type line 618
format 618
generating for ERC 550
generating for LVS 496
  generating new netlist 559
instance line 619
LVS program 450
parasitic devices 621
working file 454, 522
nets
  changing names 249
  finding 483
  floating 480
  joining 122, 249, 543
  labeling 118
  labeling errors 580
  matching 18
  measurements, saving 415
  merging 74, 91
  names 122, 253
  names, changing 113
  numbers 176, 255
  probing 573
networks
  checking connectivity 18
  terminal processing 498
nlAction property 366, 497
NLPcompleteElementString 620
NLPcreateModelString 618
NLPElementPostamble 520, 620
nlpExpr 618
nlpglobals 618, 620
NLPModelPreamble 520, 618
no-graph mode 28, 33, 73, 79
normalGrow command 322
normalGrow spacing methodology 273
notch
  checking 295
  command 295
  limits 295
notParallel command 307
npn transistor 627
O
offGrid command 142
OK button, on standard forms 25
one-way paths 467
oneWayPath command 452, 467
only_perp command 308
Open Simulation System (OSS) 21
operators, precedence 355
opposite command 309
optimizing commands 98
options forms 26
Or-bars in syntax 14
original graphics layers 131
output file, displaying for ERC 551
output modifiers 284
outside edges 183
overlap
  checks 17
  limits 296
  measuring 296
  shapes 144, 184
ovlp command 296
parasitics
  calculating 365
  defining 365
  extracting 18
  extracting an R-C network from an
  interconnect layer 442
  interconnect layer resistivity 443
  naming a device model 443
  naming R-C models 435
  netlisting devices 435
  netlisting R-C models 435, 436
  network 423
  path resistance 423
  processing a layer 442
  property containing resistor value 443
  providing a device library element 435
  reducing the network 430
  removing low value resistors 444
  saving resistor body shapes 444
  specifying capacitance parameters 444
  specifying contact resistance parameters 446
parameters
  analyzing 497, 507
  calculated values 348, 349
  comparing 509–514
  consolidating 507, 508
  extracting 18
  IL functions 512
  limit 356
  LVS 507
  manipulating 348
  measured values 348, 349
  measurement commands 348, 349
  of reduced devices 508
parasitic devices 326, 621
  creating 395
  defined devices 366
  ignoring for netlisting 366
  measuring shapes 395
  model definition 415
  polarized option 415
  versions 366
parasitic measurements 329
  associating net numbers 395–396
  mixture 368
  rejecting measured values 398
  summing measurements 398
parasitic resistance 421
  calculating polygons 424
  circuit interconnect extraction 422
  current flow 423
  distributing capacitance value 435
  extracting 18
  extracting an R-C network from an
  interconnect layer 442
  interconnect layer resistivity 443
  naming a device model 443
  naming R-C models 435
  netlisting devices 435
  netlisting R-C models 435, 436
  network 423
  path resistance 423
  processing a layer 442
  property containing resistor value 443
  providing a device library element 435
  reducing the network 430
  removing low value resistors 444
  saving resistor body shapes 444
  specifying capacitance parameters 444
  specifying contact resistance parameters 446
parasitics
  calculating 365
  defining 365
  extracting 18
  figure option 411
  manipulating 365
measuring 365, 373, 391
devices for nets 574
measuring parasitic properties of shapes 411
devices or nets 574
net option prerequisite 391
nets for devices 574
node options requirement 393
displaying
saving 365, 415
errors 569
storing 365
information 576
paths 135
ERC options 573
checking 142
explaining errors 551, 552, 575
one-way 467
LVS 570
two-way 472
tracing 467, 469, 472
two-way 472
permuteRule command 459, 503
pin instances 335
processing pins 334
devices for nets 577
pins
nets for devices 577
names, changing 113
changing labels 574
permutability 503
fix 516
physical 334
instancesLastChanged 330
processing 334
ivCellType 327
nlAction 366, 497
ivIncludeValue 542, 547
minEnclosure 279, 281
minNotch 279
minOverlap 279
minSpacing 279
minWidth 279
nlAction 366, 497
NLPElementPostamble 520
NLPModelPreamble 520
parameters 348
parameters 348
permuteRule 459, 503
recognitionShape 334
pruneDevice command 505, 533
saving 346
pull-up/pull-down
load/logic structure 464
load/logic structure 464
non-traceable paths 484, 485
non-traceable paths 484, 485
non-traceable to power or ground 486
purpose 224

raw command 325
raw data, defined 133
rectangles
  defining 129
  selecting 226
  zero-width 138
reduceDevice command 453, 460, 489
reduction
  disk space 327
  gates 464, 489
  MOS 456, 462, 489
  MOS example 462
  parallel 454, 455, 500
  run times 62
  series 454, 455, 499
  series/parallel 461, 489
region 267
registration modifiers 272
regular expression wildcards 98
relational functions 153
  exclusive option 138
Remote Diva 34
  file mapping 35
  prerequisites 34
  setting up machines 34
removeDevice command 507, 535
removing devices 507
removing layers 238
resistors 635
restarting a run 87
Return key 25, 26
rewiring
  automatic 84
rules file
  creating 95
  default file name 96
  derived layers, specifying 97
  DRC/Extraction sample 99
  ERC rules 105
  ERC sample 105
  grouping commands 96
  LVS rules 105
  LVS sample 107
  nesting commands 97
rules, switching 45
run 74
run directory 77
run modes
  overriding on cells 39
runs, saving partial 43
run-specific commands
  DRC 542
  Extract 547
run-specific file example 130
S
  sameNet command 312
  sameNet option 139
  saveDerived command 243, 328
  saveInterconnect command 332, 345
  saveParameter command 364
    associating device characteristics 348
    parameter manipulation 358
    similar to attachParasitic 411
  saveParasitic command 391, 400, 415
    relation to measureFringe 383
    relation to measureParasitic 391
  saveProperty command 346
  saveRecognition command 332, 333, 347
saving
  derived layers 265
  device recognition polygons 332
  layers 238
  partial runs 43
  properties 346
scaling 190
schematics, comparing 556
segments
  endpoints 418
  selection functions 201
  self-intersecting paths 138
  sep command 298
separation
  limits 298
  measuring 298
separation spacing methodology 272
series reduction 455, 499
series/parallel reduction 461, 489
seriesRes command 110
Set Options menu, user ability settings 26
setGround command 451, 468
setInput command 451, 469
setOutput command 451, 470
setPower command 451, 471
shapes 133–138
  abutting or coincident shapes 161
  changing dimensions 190
  coincident 155, 161
  conics 135
  copy 345
  edge-to-edge coincidence 164
  enclosure 173, 293
  interconnect 246
  invalid 137
  invalid, highlighting 140
  inverting 149
  merged 190, 191
  merging all 150
  negative sizing operation 191
  new 146, 152
    generating 144
  no overlap at abutment 155
  overlapping
    coincident edges 167
    edge-to-edge coincidence 164
    enclosed shapes 296
    limits 185
    new shapes from overlap 144
    no overlap at abutment 155
    overlapping shapes 184
    range 185
  paths 135
  selection by geomButtOrCoin 161
  selection by geomGetAngledEdge 204
  selection by geomGetPolygon 223
  selection by geomOverlap 184
  self-intersecting 137
  separation 298
  sizing 190–192
  straddling 187
  synonymous with polygon 134
  texted 227

vertex 232
  width checking 300
shielded command 313
short location
  display layers 582
si.env file 565
si.log file 555
si.out file 555
simple wildcards, using 98
simulation requirements 616
simulator support 21
sizing 190–192
  edges 191
  functions 189
  negative 190
  oversize 190
  positive 190
  undersize 190
SKILL
  commands
    using to run Diva 32
    routines 460, 464
    using to run Diva 28, 30
SKILL access 54
SKILL functions, syntax conventions 15
soft (keyword soft) 254
soft layer connections 254
source drain 341, 451
spacing checks 17, 270
special characters, example 102
SQUARE check 604
squareGrow command 323
squareGrow spacing methodology 273
straddling shapes 187
stretching
  edges 200
  negative 200
  positive 200
switch names
  currentCell 266
drc? 45, 100
extract? 45, 102
switches, setting 74, 80, 90, 541
syntax conventions 14
| T | terminals 455, 499  
back-gate 341  
correspondence points 84, 519  
definition 337  
gate 341  
ignoring 504, 527  
processing 518  
testDeviceProperty command 464, 493  
testGateProperty command 464, 495  
testing  
devices 464  
example 465  
gates 464  
text  
labels 219  
labels, adding 124  
layers 124  
strings 227, 324  
Tools menu 30  
tracing  
ground 484, 486  
power 485, 486  
transition resistance 430  
transition, width change 430  
translating Dracula files 591  
truncation 191  
twoWayPath command 452, 472 |
|---|---|
| ERC | displaying log files 554  
setting job priorities 554  
showing run information 551, 555  
showing status 551, 562  
LVS | defining cellviews 559  
displaying log files 571  
probing 562, 566  
setting job priorities 571  
showing status 571  
verification rules  
creating file 96  
creating file for DRC/Extract 95  
creating file for ERC and LVS 105  
sample file for ERC 105  
sample file for LVS BIPOlar 109  
sample file for LVS CMOS 107  
Verify menu. See also commands using 23  
Verify menu. See also commands using 30  
verifyArea command 111, 129, 258  
vertex 232  
checks 142  
finding 142  
vertical bars in syntax 14  
vias, connecting 246 |
| W | wells, checking electrical integrity 193  
width  
checking 300  
command 300  
limits 300  
wildcards 98  
limitations 99  
with_perp command 315  
Z | zero width  
paths 138  
rectangles 138 |