EMControl User Guide

Product Version 14.2
January 2002

Contents

Preface

This preface discusses the following:

- About This Guide
- How to Use This Guide
- Brief Outline of Different Chapters
- Typographic and Syntax Conventions

About This Guide

This user guide shows you how to use the EMControl tool to check the electromagnetic compliance of systems. This ensures that electronic systems operate in their environment without affecting any other system.

The user guide explains all the necessary concepts and procedures required for using the rules of the EMControl tool.

The EMControl tool checks for electromagnetic compliance through a set of rules. This guide describes these rules and also explains the language (ARL) in which these rules are written. An understanding of ARL enables the user to write his own rules as well as edit the existing rules to suit his requirements.

How to Use This Guide

The user guide is organized in a way that it begins with a brief introduction of the EMControl tool followed by one chapter each on the various high level tasks that can be performed by the tool.

The purpose behind this guide is to:

- provide conceptual understanding of the tool
- explain the various rules and predicates which are used in EMControl
- enable the user to use the rules according to his design requirements

- enable the user to write new rules

If you are a new user and do not have any prior working experience with the EMControl tool, then start your learning process from the first chapter and continue exploring the different tools in the sequence as covered in the user guide. If you are using the user guide as reference, then you may directly reference any chapter corresponding to a particular topic. Refer details in the Brief Outline of Different Chapters section.

Brief Outline of Different Chapters

This guide is organized into five chapters:

1. Chapter 1: Introduction to EMControl

   This chapter gives a brief introduction to the EMControl tool. It discusses the use model for EMControl, the various tasks that can be performed by EMControl and the tasks performed by different users of the tool.

2. Chapter 2:Setting Up the EMControl Environment

   This chapter describes the EMControl environment, And also talks about how you can go about changing the default settings. It discusses the basic concepts required to use the tool.

3. Chapter 3: Performing EMControl Rule Checking

   This chapter discusses the various stages at which the various rules of the EMControl rule can be used. Then, it also talks about the tasks to be performed for checking the design for electromagnetic compliance.

4. Chapter 4: Resolving EMC Rule Violations in Your Design

   This chapter discusses various steps after the rules have been executed on the design. It talks of how the results are displayed and what information about the execution results to find where.

5. Chapter 5: Writing Rules

   This chapter describes the steps for writing new rules, how to write predicates, manipulate parameters, and get a rule working. It also lists some sample rules.

Typographic and Syntax Conventions

This list describes the syntax conventions used for tools used in the CheckPlus User Guide.

`literal (LITERAL)`	Nonitalic or (UPPERCASE) words indicate key words that you must enter literally. These keywords represent command (function, routine) or option names.
`argument`	Words in italics indicate user-defined arguments for which you must substitute a value.
|	Vertical bars (OR-bars) separate possible choices for a single argument. They take precedence over any other character. For example, `command` argument | argument
[ ]	Brackets denote optional arguments. When used with OR-bars, they enclose a list of choices. You can choose one argument from the list.
{ }	Braces are used with OR-bars and enclose a list of choices. You must choose one argument from the list.
...	Three dots (...) indicate that you can repeat the previous argument. If they are used 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 argument, but more are possible` `[`*argument*`]...: you can specify zero or more arguments`
,...	A comma and three dots together indicate that if you specify more than one argument, you must separate those arguments by commas.
`Courier font`	Indicates command line examples.

1

Introduction to EMControl

EMControl Overview

Systems can adversely impact each other due to electromagnetic interference (EMI), or due to unwanted coupling of energy between conductors, components, and systems. Electromagnetic compatibility (EMC) is the ability of electronic systems to function as expected within their intended environment without adversely affecting other systems.

An effective way to help meet EMC requirements is to use EMC design guidelines or EMC rules to screen the design for potential problems. You can use the EMControl tool to detect problem areas in your design early in the design cycle and take immediate steps to resolve those problems.

EMControl provides this capability of detecting problems by enabling you to repeatedly check your design against selected sets of EMC rules chosen by a user who has expertise in EMC.

The EMControl product includes several default sets of EMC rules. You can also write your own rules to verify specific design, environment, and regulatory requirements. Running EMControl early in the design cycle often helps to detect potential EMC problems before they can significantly impact product development.

The typical use model for EMControl to check electromagnetic compatibility (EMC) in high-speed printed circuit board (PCB) design is:

- Initialize
- Property Setup
- Rule Select
- Customize Rule Parameters
- Audit
- Execute
- Results/Reports

Initialize

- Specifies EMC run directory.

Property Setup

- Sets the critical components, nets, and regions. The setup can be done automatically or interactively.
- Identifies EMC components by attaching the EMC_COMP_TYPE property.

Rule Select

- Specifies the parameter paths and rule paths.
- Writes new rules, if required.
- Selects the rules to be run.
- Customizes the rule parameters, if required.
- Specifies the scope of the design to be checked.

Customize Rule Parameters

- Sets rule parameters.

Audit

- Performs a check to verify whether or not the required properties are setup.

Execute

- Executes the selected rules on the specified design.

Results/Reports

- Views the execute and audit reports.
- Cross probes the EMC violations using the markers utility.

EMControl Users

EMC verification tasks can be performed by three types of EMControl users:

- EMC expert
- Design engineer
- Layout designer

The EMC expert determines the EMC requirements that must be met by the completed design and then maps these requirements to EMC rules. A combination of past experience and knowledge of the requirements of the design influence the EMC expert in determining:

- which EMC rules can be used as is
- which Cadence-supplied EMC rules must be modified
- whether or not any new EMC rules must be implemented in order to ensure that the design meets EMC requirements.

After the EMC expert has identified the EMC rule sets to be used to verify the design, the design engineer can set up the EMControl tool. The design engineer then implements the required EMC rule set(s) and associated variables; classifies the components, nets, and regions in the design according to their various levels of criticality; and attaches the Allegro and EMControl properties required by the selected EMC rules.

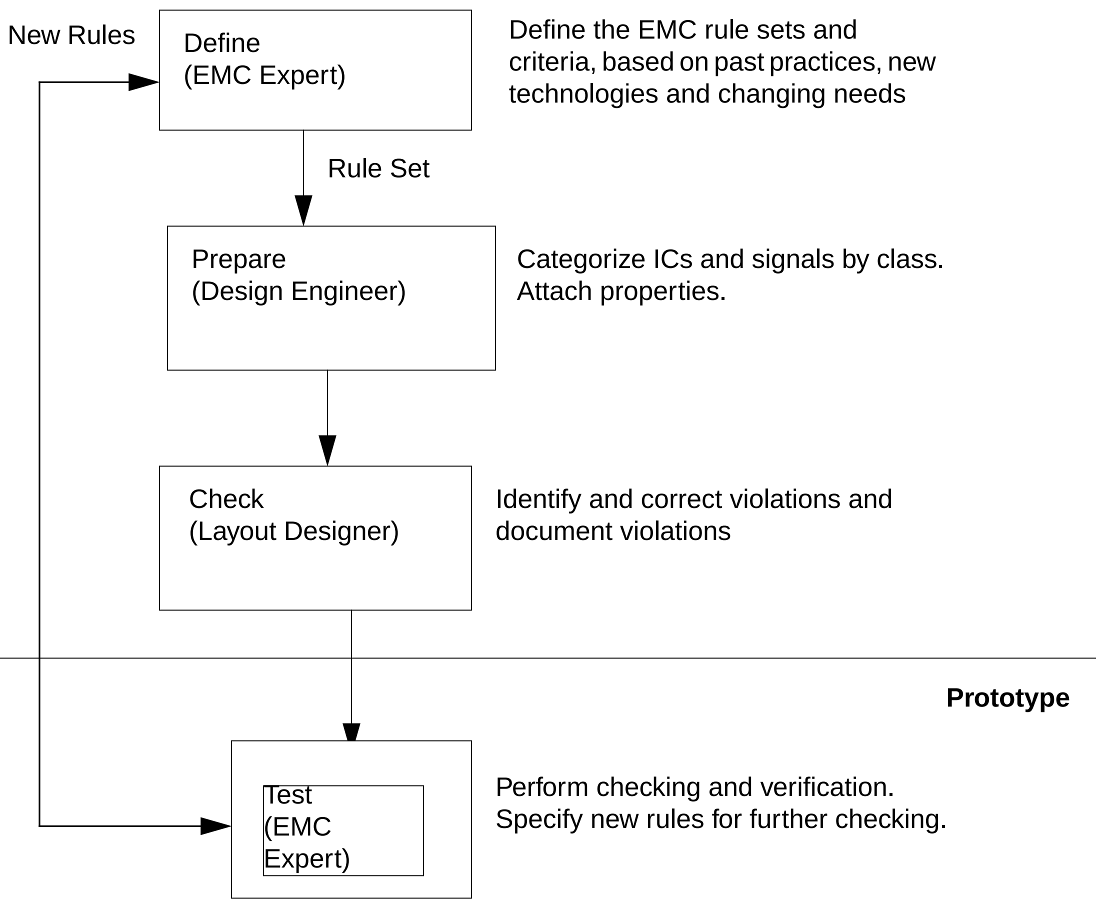
Once the rule set has been identified and property assignments are made, the layout designer can set up and run EMC rule checks. The layout designer identifies and correct any EMC rule violations reported during EMC rule checking.

After a prototype is prepared using this method, the EMC expert can verify the design and identify new sets of EMC rules for further EMC rule checking.

The use of EMControl for design development an iterative process. Throughout design development, the EMC expert, the design engineer, and the layout designer confer to ensure that all constraints are being met. They also define trade-offs between constraints, and verify whether or not the completed design will meet all EMC requirements.

Figure 1-1 on page 16 illustrates a typical workflow that you might use when designing for EMC.

Figure 1-1 EMC Checking Flow


New Rules
Define (EMC Expert)
Define the EMC rule sets and criteria, based on past practices, new technologies and changing needs
Rule Set
Prepare (Design Engineer)
Categorize ICs and signals by class. Attach properties.
Check (Layout Designer)
Identify and correct violations and document violations
Prototype
Test (EMC Expert)
Perform checking and verification. Specify new rules for further checking.

EMControl Tasks

EMControl enables the design engineer and the layout designer to perform the following tasks:

- Select specific EMC rule sets to check against the design.
- View a help file for each rule to determine:
    - Whether or not the rule should be used for an EMC rule-checking run.
    - Whether or not the rule should be customized.
    - The properties and the property values required by the rule.

  - ❑ The variables and their values required by the rule.

- Select the scope of the design that requires checking.
- Use Audit commands to verify that the properties required by the selected rules have been properly defined.
- Run the EMC rule checker to search for EMC violations.
- View the results of the check in a list of rule violations.
- Highlight an individual violation within the design.
- View the feedback from the Markers for correcting the EMC violations.

An EMC expert can use EMControl to customize the default rule-checking in the following ways:

- Customizing default EMC rules locally by editing variable values through the `emc_custom.par` file.
- Customizing default EMC rules for an entire site of users by editing variable values in the `emc_param.par` site parameter file.
- Writing and compiling new EMC rules, in the Cadence Advanced Rule Language (ARL). These rules can be applied in EMC rule checking.

Checking for EMC Rule Violations

The EMC expert, design engineer, and the layout designer have specific tasks to perform for verifying whether or not a design is EMC compliant.

EMControl can be used during:

- placement (pre-route) stage
- routing stage
- post-route stage

Tasks to be Performed

EMC expert

- ❑ Use established EMC standards and past experience to determine the EMC requirements for the design.
- ❑ Specify or develop project-specific, customized EMC rules and modify system-provided default EMC rules, if necessary.
- ❑ Select EMC rule sets for the different stages of design verification.
- ❑ Review the results of EMC rule-checking runs and specify new EMC rule sets for further testing.

Design engineer

- ❑ Set up the EMControl product, if this has not already been done.
- ❑ Customize EMC rule variables (parameters) where required. The EMC expert can provide key inputs here.
- ❑ Assign relevant EMControl and Allegro properties to components and set the initial property values.
- ❑ Set up the EMC rule-checking environment.
- ❑ Specify the set of EMC rules to use during each rule-checking run.
- ❑ Specify the portion of the design to check for each rule set.
- ❑ Audit the EMControl tool setup and design preparation.
- ❑ Confer with the EMC expert to resolve any setup issues.

Layout designer

- ❑ Run EMC rule checking.
- ❑ View EMC rule violation messages and correct problems in the design.
- ❑ Confer with the design engineer and the EMC expert as required while resolving violations.
- ❑ Run rule checking again, if required.

Using SigNoise with EMControl

Some of the signal routing and signal quality rules provided with EMControl use SigNoise simulations and SigNoise device models. These EMC rules enable design engineers to begin evaluating their designs for electromagnetic interference early in the design process and with increasing accuracy throughout the design development.

Before running EMC rule-checking, you need to perform the following SigNoise setup tasks:

- Initialize the SigNoise run directory.
- Specify the SigNoise model libraries to be used.

Assign the `SIGNAL_MODEL` property to components.

2

Setting Up the EMControl Environment

Default EMControl Installation Directory Structure

The EMControl installation directory is located on your system at <*your_install_dir*>/share/pcb/signal/emc, where *your_install_dir* specifies the path of the directory where the design software is installed.

The EMControl installation directory <*your_install_dir*>/share/pcb/signal/emc contains the following subdirectories:

- rule_src: This directory contains six source rule files (source rule files) and the corresponding rule-verification files:
  - emc_placement.arl and emc_placement_verify.arl
  - emc_bypass.arl and emc_bypass_verify.arl
  - emc_pwr_gnd_dist.arl and emc_pwr_gnd_dist_verify.arl
  - emc_dc_route.arl and emc_dc_route_verify.arl
  - emc_sig_route.arl and emc_sig_route_verify.arl
  - emc_sig_qual.arl and emc_sig_qual_verify.arl
- rules: This directory contains the compiled Cadence mapping file (emc_allegro.env). This rules directory also contains the six complied rule files, and the corresponding compiled rule-verification files:
  - emc_placement.rle and emc_placement.rle.verify
  - emc_bypass.rle and emc_bypass.rle.verify
  - emc_pwr_gnd_dist.rle and emc_pwr_gnd_dis.rle.verify
  - emc_dc_route.rle and emc_dc_route.rle.verify
  - emc_sig_route.rle and emc_sig_route.rle.verify
  - emc_sig_qual.rle and emc_sig_qual.rle.verify

- `include`: This directory contains the following files and directories:
  - `emc_param.par`: This file defines the Cadence supplied user-definable parameters and is used to assign them default values.
  - `emc_custom.par`: This is a dummy file copied into the user's directory. Initially, it is empty. When the user modifies the values of any parameters, the changes are reflected in this file.
  - `site` directory: This directory contains the site parameter file, `emc_param.par`. You can use the `emc_param.par` file to customize the values of different parameters for your site. However, you must have write permissions to modify this file.
- `help`: This directory contains the help (`.hlp`) files for the rules. You can display the help file contents in the EMC Rule Selection form rule browser. For more information about displaying help file contents, refer Chapter 3, "Performing EMControl Rule Checking,"
- `data`: This directory contains miscellaneous files.
- `symbols`: This directory contains miscellaneous files.

The EMControl Mapping File

EMControl uses a mapping file to define the various aspects of the rule-checking environment.

Mapping File Functions

The mapping file `emc_allegro.env`:

- Identifies the objects in the layout physical environment that are used by the EMC rules.
- Identifies the EMControl predicates and the objects that they use.

  EMC rules are implemented using predicates to perform their functions. For reference information on the EMControl predicates, see Appendix C, "EMControl Predicates."
- Maps EMControl predicates to the SKILL/C functions that perform their tasks.
- Defines message severity levels for EMControl rule violations.

Message Severity Levels

The severity levels assigned to rule violations are defined in the mapping file. You can use any of the following severity levels to display a violation message in the Markers dialog box:

- `FATAL` (terminates rule execution)
- `ERROR`
- `WARNING`
- `OVERSIGHT` (a probable warning)
- `INFO`

Note: Severity levels are listed in the descending order of importance.

EMControl Basics

The following sections describe some basic EMControl tasks and provide general information you will need to use EMControl.

- Accessing EMControl on page 23
- Initializing EMControl on page 24
- Initializing SigNoise on page 26

Accessing EMControl

To access EMControl from SPECCTRAQuest:

➤ Select *Tools > EMControl Rules* from SPECCTRAQuest menu bar.

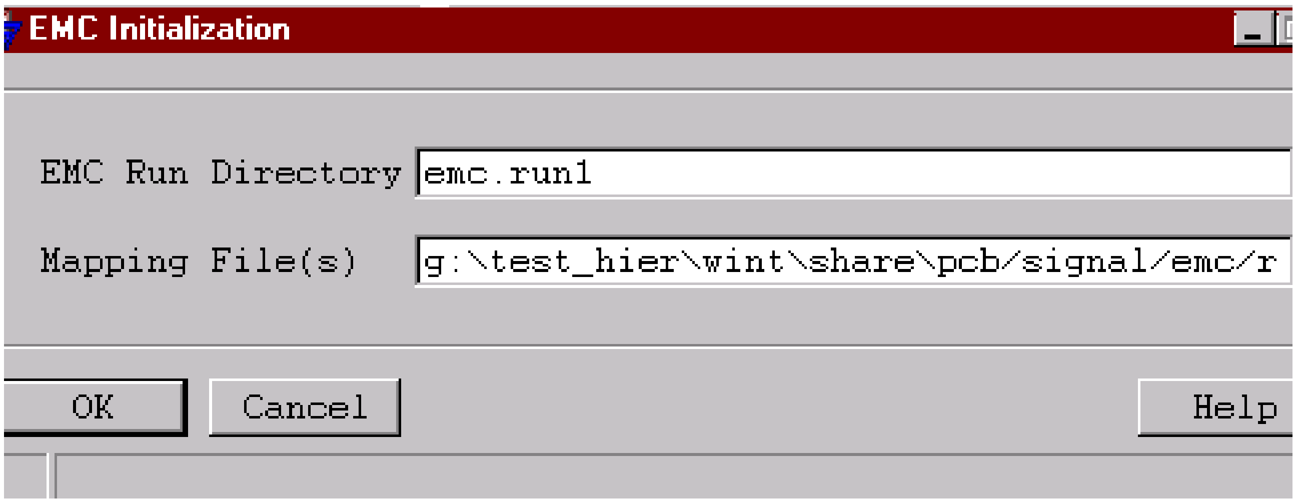
Initializing EMControl

Before you perform EMControl checking on a design, you may want to modify the default setup in the EMC Initialization dialog box.

To open the EMC Initialization dialog box:

➤ Select *Initialize* from the EMC menu.

The EMC Initialization dialog box appears.



EMC Run Directory: emc.run1

Mapping File(s): g:\test_hier\wint\share\pcb/signal/emc/r

Initializing the EMC Run Directory

To create an EMC run directory, you need write privileges. To run EMControl, you need read privileges for the mapping files and the SigNoise run directory.

EMControl uses the EMC run directory to read and store the files that it generates for rule checking of the current design. These files include:

- `emcrc.log`: This file contains a summary of the EMControl session, including the name of the design, the rule names selected, the start and stop time, and the total number of violations detected.
- `emcrc.ini`: This file contains information on the rules that you have selected for the check run.
- `emcrc.setup`: This file contains the rule file information settings used during the latest rule check.
- `emcrc_execute.msg`: This file contains the short and advisor messages that describe the violations encountered in an EMC rule-checking session.

- `emcrc.mkr`: This file contains the information required by the markers utility for each rule violation.
- `emcrc.verify`: This file contains the rule file information settings used during the latest run to verify (audit) the property setup for the selected rules.
- `emcrc_verify.msg`: This file contains the short and advisor messages that describe the violations encountered during property verification by the *Audit* command.
- `emcrc.res`: This file contains the number of violations recorded against each rule in the latest run. This file is created when you use Allegro in no-graphic mode.
- `emcrc_propedit.log`: This file contains the messages displayed in the most recent run of automatic property tagging.

To define the EMC run directory

When EMControl is used for the first time, the default run directory is assigned the name `emc.run1`. Notice that the name has suffix `1`. Subsequently, whenever you run EMControl, the number in the default run directory is incremented by `1`. Therefore, you have all EMC run directories for all rule-checking runs you perform. You can, at any point of time initialize any run directory and view the results. By default, EMControl creates the run directory. You can change this location. To change the location:

1. Specify the full path in *EMC Run Directory* to place the directory at another location.

   Note: You need write privileges to change the path of the run directory.

2. Click *OK*.

   Saves the settings and closes the EMC Initialization dialog box.

Note: The EMC run directory name is stored in a design-level property called `EMC_RUN_DIR`.

Specifying the Mapping File

Cadence supplies the default mapping file called `emc_allegro.env`. In addition to this fie, you can also write your own predicates and create a new `env` file for specifying mappings to SKILL functions.

Multiple mapping files can be specified in the EMC Initialization dialog box.

To define the environment for checking

➤ You specify the location of the default mapping file and any other mapping files that you have created in the EMC Initialization dialog box. The mapping files have to separated by spaces.

- The.`env` file name extension is required.
- By default, EMControl searches for a mapping file in the software installation hierarchy. Edit the path if the file is at another location.

Initializing SigNoise

To run any of the EMControl rules that require information from SigNoise, first initialize SigNoise as outlined in this section. Appendix B, "EMControl Rules," describes the EMControl rules in detail.

Before running EMControl rule checking, you need to perform the following SigNoise setup tasks:

- Initialize the SigNoise run directory.
- Specify which SigNoise model libraries to use.
- Assign the `SIGNAL_MODEL` property to components.

To initialize the SigNoise run directory:

➤ In the *SigNoise* menu, select *Initialize*.

This should be the same SigNoise run directory you provided in the EMC Initialization dialog box. (See Initializing EMControl.)

To specify which SigNoise model libraries are to be used:

1. In the *SigNoise* menu, select *Library* to display the *Signal Analysis Library* Browser. Load the required libraries.
2. Select *OK* in the Signal Analysis Library Browser.

   This `librefs.dat` file is created in the SigNoise run directory. This file contains paths to the specified SigNoise libraries to lookup signal integrity model data.

EMControl rules that are SigNoise-dependent use the `librefs.dat` file to locate the relevant signal integrity models. EMControl locates the `librefs.dat` file by looking in the directory specified in the SigNoise run directory field of the EMC Initialization form. If the file does not exist, the default SigNoise library file from the software installation is used:

```
$CDS_INST_DIR/share/pcb/signal/cds_iocells.dml
```

To assign the `SIGNAL_MODEL` property to components:

1. In the *SigNoise* menu, select *Model* to display the *Signal Model Assignment* dialog box.
2. Assign the `SIGNAL_MODEL` property to required components.
3. To retain the `SIGNAL_MODEL` properties, save the design.

EMControl Properties

To determine which design objects to check ,the EMC rules examine certain property values associated with design objects. For example, the EMC `bypass_critical_IC` rule searches for ICs with the `EMC_CRITICAL_IC` property assigned. For each critical IC it finds, EMControl searches for a minimum number of bypass capacitors attached to the component. EMControl identifies the bypass capacitors by verifying whether or not the `EMC_COMP_TYPE` property is assigned the `BYPASS_CAP` value. EMControl also evaluates each bypass capacitor by its assigned `TOL` and `VALUE` property values and reports any problems.

Table 2-1 on page 28 alphabetically lists the properties used by the system-provided EMC rules. It also lists the EMC rules that examine each property.

Table 2-1 Properties Used by the System-Supplied EMControl Rules

Property Name	Description	Used in Rules
`EMC_COMP_TYPE`	String that specifies a variable to identify the component type for an IC. Assigns one of the following predefined values: FAST_SWITCH_TRANSISTOR BYPASS_CAP LINE_DRIVER LINE_RECIEVER BIDIR_TRANS CLOCK_GEN FILTER GND_SCREW FENCE NONE	bypass_cap_type bypass_critical_IC bypass_drvr_rcvr_bidir bypass_fast_sw_trans bypass_pwr_trace central_clock clock_spectral_content critical_IC_loop_area critical_IC_3caps_C_2C_4C decouple_emc_regions fence_off_emc_regions filtered_IO_signals filters_to_clean_ground gnd_under_clock gnd_screw_between_clock_and_conn nets_over_clean_gnd shield_clock_nets single_diff_mode_EMI sum_diff_mode_EMI return_path_near_signal_via
`EMC_CRITICAL_IC`	String that identifies the class of a critical IC	bypass_critical_IC conn_in_low_freq_regions critical_IC_loop_area critical_IC_3caps_C_2C_4C return_path_near_signal_via

Property Name	Description	Used in Rules
`EMC_CRITICAL_NET`	String that identifies the class of a critical net	clock_spectral_content critical_net_card_edge_dist critical_net_exp_length critical_net_man_ratio critical_net_ringing critical_net_termination critical_net_via_count critical_net_via_pin_ratio max_critical_net_xtalk no_critical_net_thru_IO_comps critical_net_hole_dist critical_net_return_path return_path_near_signal_via
`EMC_CRITICAL_REGION`	String that identifies the class of a critical region.	conn_in_low_freq_regions decouple_emc_regions fence_off_emc_regions
`PINUSE`	String that identifies pin use.	bypass_drvr_rcvr_bidir central_clock clock_spectral_content critical_IC_loop_area critical_IC_3caps_C_2C_4C critical_net_via_pin_ratio shield_clock_nets single_diff_mode_EMI sum_diff_mode_EMI
`TOL`	String that determines the tolerance of a bypass capacitor.	bypass_critical_IC bypass_cap_type

Property Name	Description	Used in Rules
`VALUE`	String that determines the capacitance of a bypass capacitor.	critical_IC_3caps_C_2C_4C bypass_critical_IC bypass_cap_type
`VOLTAGE`	String that identifies a nonsignal net. For ground, use VOLTAGE=0. For power, use VOLTAGE= a nonzero value.	bypass_pwr_trace comp_to_conn_dist filters_to_clean_ground gnd_under_clock max_pwr_gnd_resistance nets_over_clean_gnd no_critical_net_thru_IO_comps pwr_gnd_trace_width shield_clock_nets bypass_cap_para_plane_square
`VOLTAGE_SOURCE_PIN`	String that identifies the voltage source when no independent voltage plane is available. Attach property `VOLTAGE_SOURCE_PIN` to a voltage source pin.	max_pwr_gnd_resistance

For more information on individual EMC rules and the properties that each rule uses, you can:

- Refer the help information available for each rule in the rule browser (see Chapter 3, "Performing EMControl Rule Checking,").
- Refer to the individual rule descriptions in Appendix B, "EMControl Rules."

Note: All the properties used by EMControl are standard layout properties.

Assigning EMControl Property Values

EMC regions are areas within a design where components and nets with similar EMC characteristics are clustered together. EMC regions are defined using the Allegro and SPECTRAQuest room mechanism.

Schematic implementation is the recommended time for the design engineer to assign EMC-specific classes and other required properties to components and nets. Property requirements are determined by the specific rules included in an EMC rule-checking run.

EMControl supplies up to five critical frequency classifications to describe the EMC characteristics of the ICs and nets in a design. You can either accept these default classes or you can redefine the boundary values to create up to five critical frequency classes that better describe the EMC characteristics of the components and nets comprising your design.

For example, to run EMC rules based on EMC regions:

1. To define the required EMC regions in your design, use the design environment software to create rooms in your design.
2. Assign `EMC_CRITICAL_REGION` property values to rooms on the board according to the frequency level of the components and nets clustered together within each region.
3. Assign an appropriate `EMC_CRITICAL_IC` and `EMC_CRITICAL_NET` property value to all ICs and nets located in each region to identify the EMC characteristics of each component and net.

You can perform either an automatic or a manual process to assign EMControl property values to objects.

- To automatically assign property values to design objects, use the *Auto Setup* EMC menu option.

  See Automatically Attaching Properties to Design Objects for information on attaching properties automatically.

- To manually assign property values to selected design objects, use the *Manual Setup* EMC menu option.

  See Manually Attaching Properties to Selected Design Objects for information on attaching properties manually

Stackup definition occurs at the layout stage of the design process. The definition of physical and electrical constraints can occur either in the layout stage or in the schematic design stage.

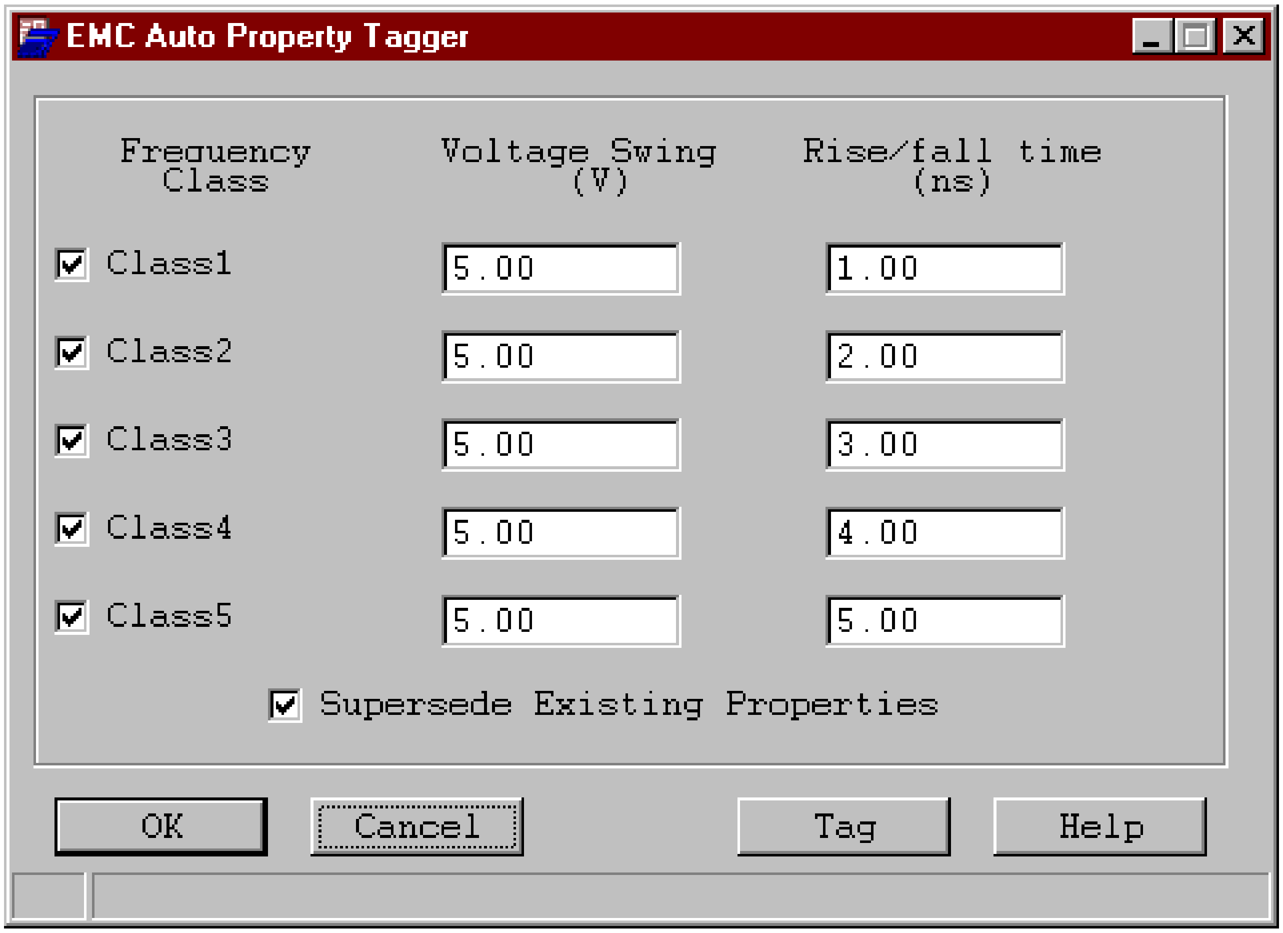
Automatically Attaching Properties to Design Objects

You can attach the critical frequency classes to your design objects.

To attach EMC properties to your design:

1. Select *Auto Setup* from EMC menu.

   The EMC Auto Property Tagger dialog box appears.



EMC Auto Property Tagger

Frequency Class	Voltage Swing (V)	Rise/fall time (ns)
✓ Class1	5.00	1.00
✓ Class2	5.00	2.00
✓ Class3	5.00	3.00
✓ Class4	5.00	4.00
✓ Class5	5.00	5.00

✓ Supersede Existing Properties

2. Select the critical *Frequency Class* toggle buttons to specify the number of classes.

   You can have up to five consecutive critical frequency classifications for components, nets, and regions. For example, when you deselect CLASS4, CLASS5 is also deselected. You are left with CLASS1, CLASS2, and CLASS3.

3. Specify the *Voltage Swing* and *Rise/Fall Time* values, if required.

   The values you enter are saved as a property on the design but are invisible to users.

4. Click the toggle button to select or deselect the *Supersede Existing Properties*.

   When the *Supersede Existing Properties* check box is selected, any existing frequency class boundary values are replaced. When the *Supersede Existing Properties* check box is not selected, new frequency classes are added but the existing frequency class boundary values are not changed.

5. Click *Tag*.

   The Auto Tag Report appears.

   Note: The `EMC_CRITICAL_IC`, `EMC_CRITICAL_NET`, and `EMC_CRITICAL_REGION` properties are attached and the EMC AutoPropertyTag log file (`emcrc_propedit.log`) is created.

Viewing the Log File for Automatic Property Tagging

When you automatically attach critical frequency class properties to objects in your design, EMControl creates the EMC AutoPropertyTag log file (`emcrc_propedit.log`) in the EMControl run directory and displays the file.

The log file contains the following sections:

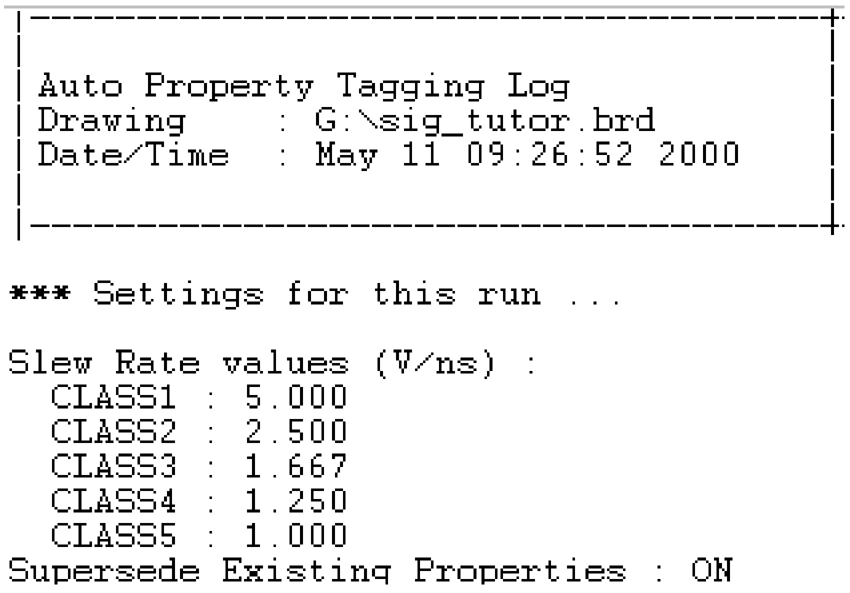
- Header Information
- Information About Critical Nets
- Information About Components on Critical Nets
- Information About Critical Regions
- Summary of Tagging

Header Information

The EMC AutoPropertyTag Log file contains the following header information:

- The name of the design for which the report was generated
- The date and time the report was generated
- The settings for the run:
  - The names of critical frequency classes used
  - The slew rate associated with each class
  - Whether *Supersede Existing Properties* was on or off

The following is an example of a log file header:



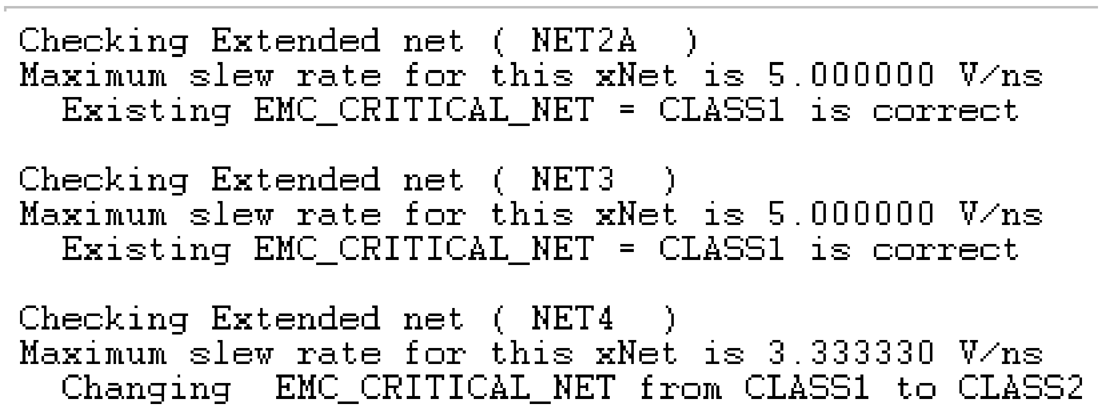
```
|--------------------------------------+
|                                      |
|Auto Property Tagging Log             |
|Drawing    : G:\sig_tutor.brd         |
|Date/Time  : May 11 09:26:52 2000     |
|                                      |
|--------------------------------------+

*** Settings for this run ...

Slew Rate values (V/ns) :
  CLASS1 : 5.000
  CLASS2 : 2.500
  CLASS3 : 1.667
  CLASS4 : 1.250
  CLASS5 : 1.000
Supersede Existing Properties : ON
```

Information About Critical Nets

The `Checking Nets` section of an EMC AutoPropertyTag log file contains the following information:

- The name of the net
- The maximum slew rate on the net
- The critical frequency class name value for the `EMC_CRITICAL_NET` property if it is attached to the net

The following is an example from the `Checking Nets` section of an EMC AutoPropertyTag log file:



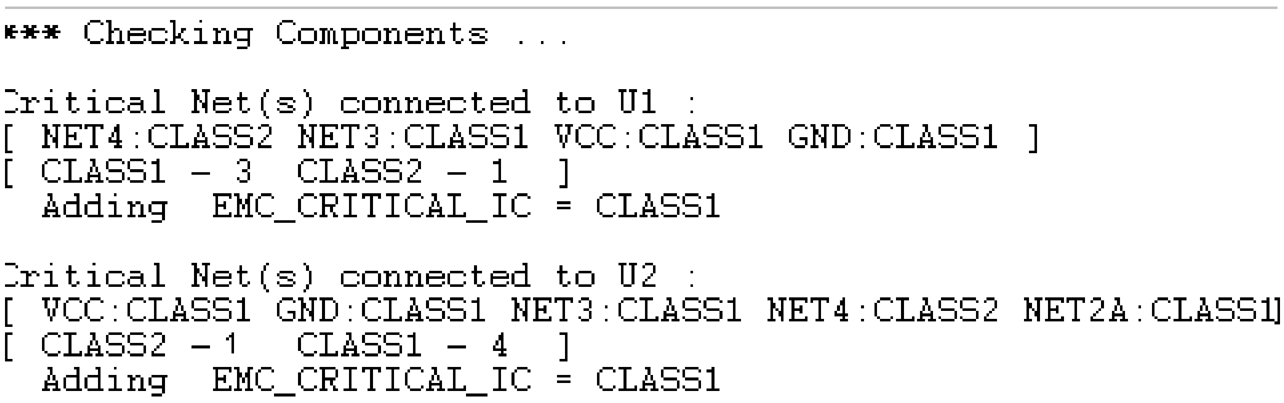
```
Checking Extended net ( NET2A  )
Maximum slew rate for this xNet is 5.000000 V/ns
  Existing EMC_CRITICAL_NET = CLASS1 is correct

Checking Extended net ( NET3  )
Maximum slew rate for this xNet is 5.000000 V/ns
  Existing EMC_CRITICAL_NET = CLASS1 is correct

Checking Extended net ( NET4  )
Maximum slew rate for this xNet is 3.333330 V/ns
  Changing  EMC_CRITICAL_NET from CLASS1 to CLASS2
```

Information About Components on Critical Nets

The `Checking Components` section of an EMC AutoPropertyTag log file contains the following information for every component:

- The name of each component
- The name and critical frequency class for all critical nets that connect to the component
- The number of nets of each critical class.
- The critical frequency class name value for the `EMC_CRITICAL_IC` property attached to the component (this is the highest value of the EMC_CRITICAL_NET property attached to the nets connected to the component)

The following is an example of the `Checking Components` section of an EMC AutoPropertyTag log file:



```
*** Checking Components ...

Critical Net(s) connected to U1 :
[ NET4:CLASS2 NET3:CLASS1 VCC:CLASS1 GND:CLASS1 ]
[ CLASS1 - 3  CLASS2 - 1  ]
  Adding  EMC_CRITICAL_IC = CLASS1

Critical Net(s) connected to U2 :
[ VCC:CLASS1 GND:CLASS1 NET3:CLASS1 NET4:CLASS2 NET2A:CLASS1]
[ CLASS2 - 1  CLASS1 - 4  ]
  Adding  EMC_CRITICAL_IC = CLASS1
```

Information About Critical Regions

The `Checking Regions` section of an EMC AutoPropertyTag log file contains the following information:

- A list of components in the room with critical frequency class name values attached
- The total number of components tagged with each critical frequency class
- The critical frequency class name value for the `EMC_CRITICAL_REGION` property attached to the room.

The following is an example taken from the Checking Regions section of an EMC AutoPropertyTag log file:

```
 *** Checking Regions ...

 Critical Component(s) in ROOM1 :
 [ ZC5:CLASS2 ZB5:CLASS2 ZC6:CLASS2 ZB6:CLASS2 ZC7:CLASS2 ZB7:CLASS2 ]
 [ CLASS2 - 6  ]
   Existing EMC_CRITICAL_REGION = CLASS2 is correct
```

Summary of Tagging

The summary of tagging section contains a summary of the tagged critical components, critical nets, and critical regions. An example of the summary of Tagging section is displayed:

```
------------------------------------------------------------
                     Autotag log summary
------------------------------------------------------------
 Critical nets tagged
 CLASS1:
        NET2A
        NET3
 CLASS2:
        NET1A
        NET1
        NET4
------------------------------------------------------------
 Critical components tagged
 CLASS1:
        U2
        U1
------------------------------------------------------------
```

```
Critical regions tagged
CLASS1:
        room1
        clock_room
------------------------------------------------------------
```

Manually Attaching Properties to Selected Design Objects

A component can be selected for property tagging by its reference designator, device type, or property.

To the selected design objects, you can attach:

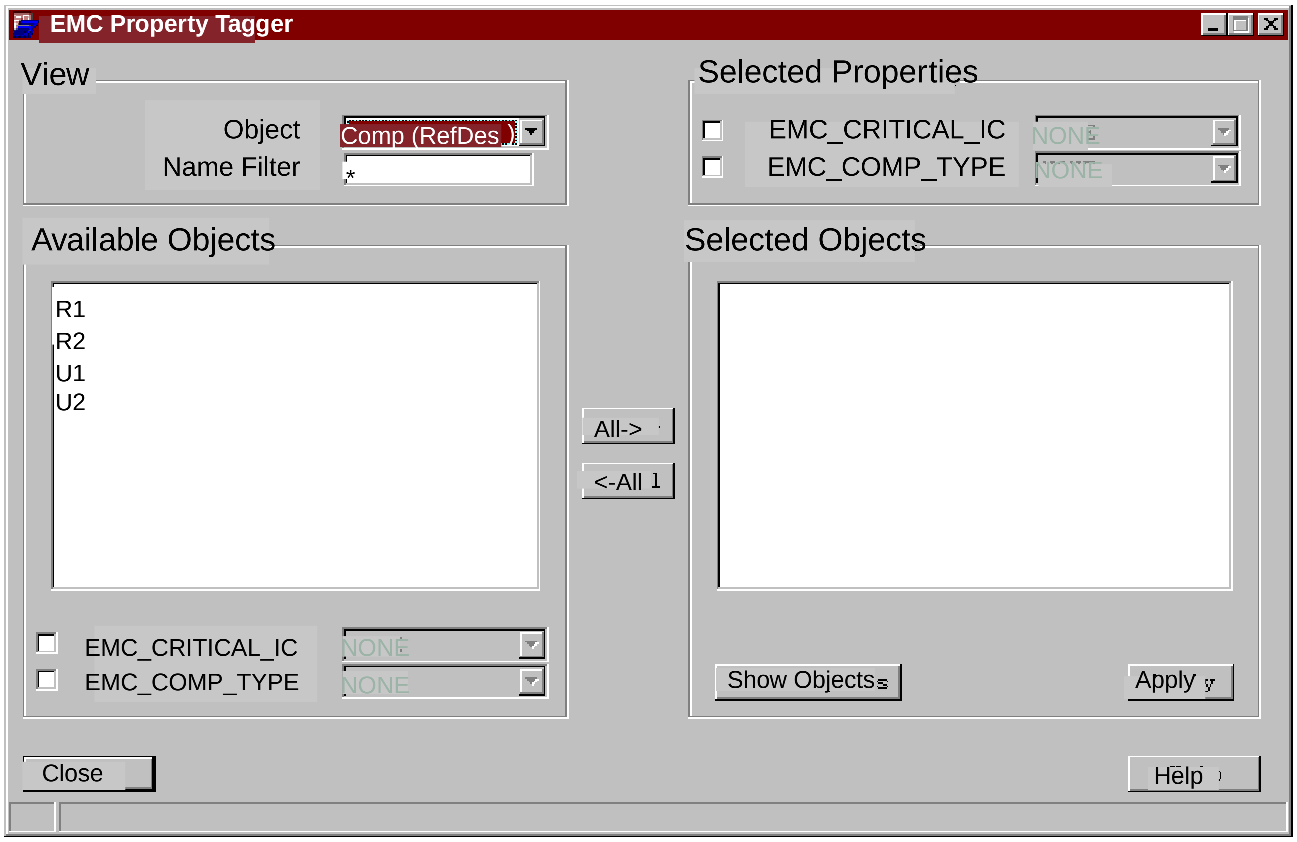
- Component Properties
- Net Properties
- Room Properties

Component Properties

To attach component properties to a design object

1. Select *Manual Setup* from the EMC menu.

The EMC Property Tagger dialog box is displayed..



2. Specify the *Object* field as `comp`.

   The components are loaded in the *Available Objects*.

3. Filter the available objects by specifying *Name Filter***.**

   Note: Filtering can be done on the basis of property by toggling on the properties given in the *Available Objects* list.

4. Filter the available objects further by selecting the *EMC_CRITICAL_IC* and *EMC_COMP_TYPE* check boxes and by setting the class and type respectively.

5. Click the component in the list.

   This transfers the component from the *Available Objects* list to *Selected Objects* list.

   Note: *All* > and < *All* can be used for transferring all components between the *Available Objects* list and the *Selected Objects* list.

6. Specify the property value in *Selected Properties***.**

7. Click *Apply*.

All the components in *Selected Objects* list are tagged with the property value displayed in the *Selected Properties* list.

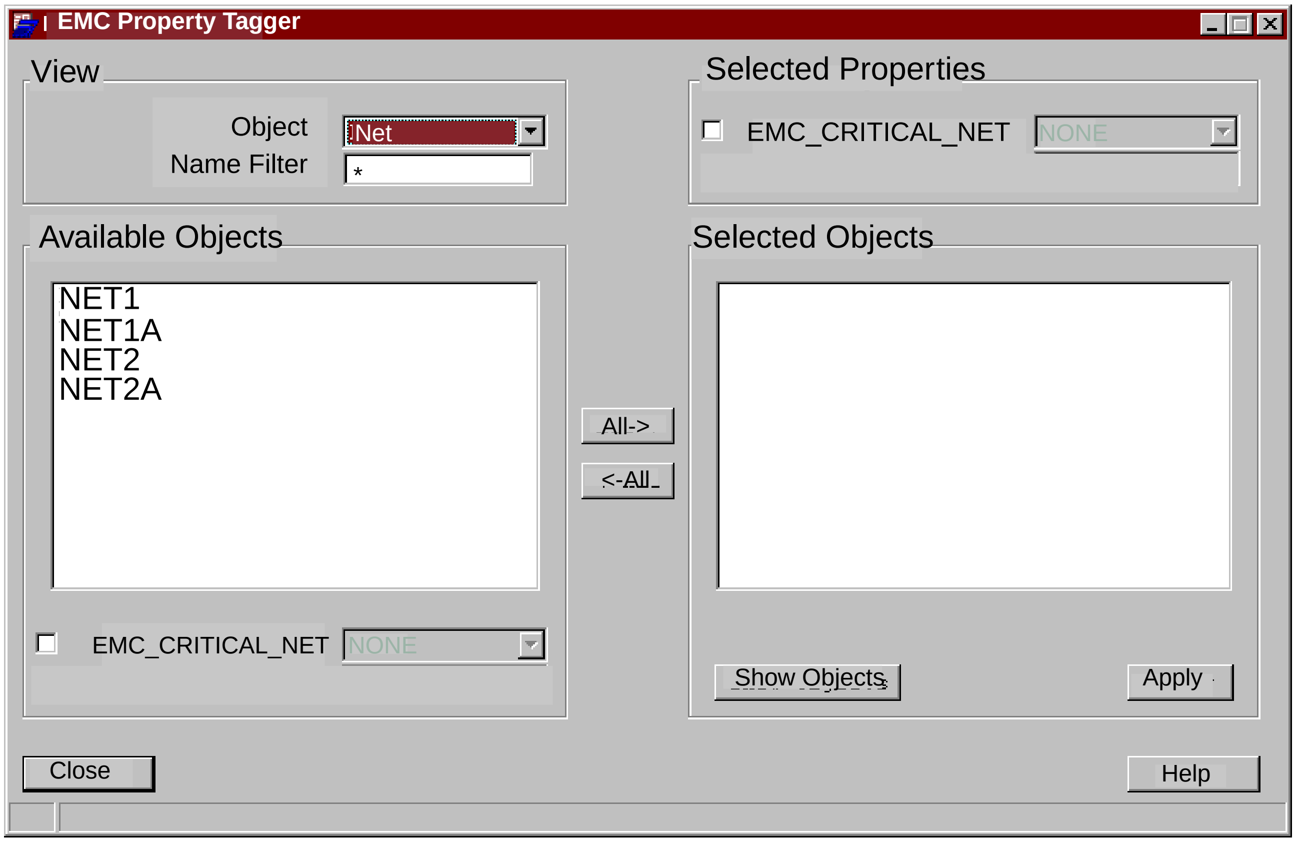
Note: If you assign a property the value `NONE`, EMC will delete the property.

Net Properties

To attach net properties

1. Select *Manual Setup* from EMC menu.

   The EMC Property Tagger dialog box is displayed.



2. Specify the *Object* field as `net`.

   The selected nets are loaded in the *Available Objects*.

3. Filter the available objects by specifying *Name Filter*.

   Note: Filtering can be done on the basis of property by toggling the properties given in the *Available Objects* list.

4. Filter the available objects further by checking the *EMC_CRITICAL_NET* check box and setting the class.
5. Click the net in the list.

   This transfers the net from the *Available Objects* list to *Selected Objects* list.

   Note: *All* > and < *All* can be used for transferring all nets between the *Available Objects* list and the *Selected Objects* list.
6. Specify the property value in the *Selected Properties*.
7. Click *Apply*.

   All the nets in *Selected Objects* list are tagged with the property value given in the *Selected Properties* list.

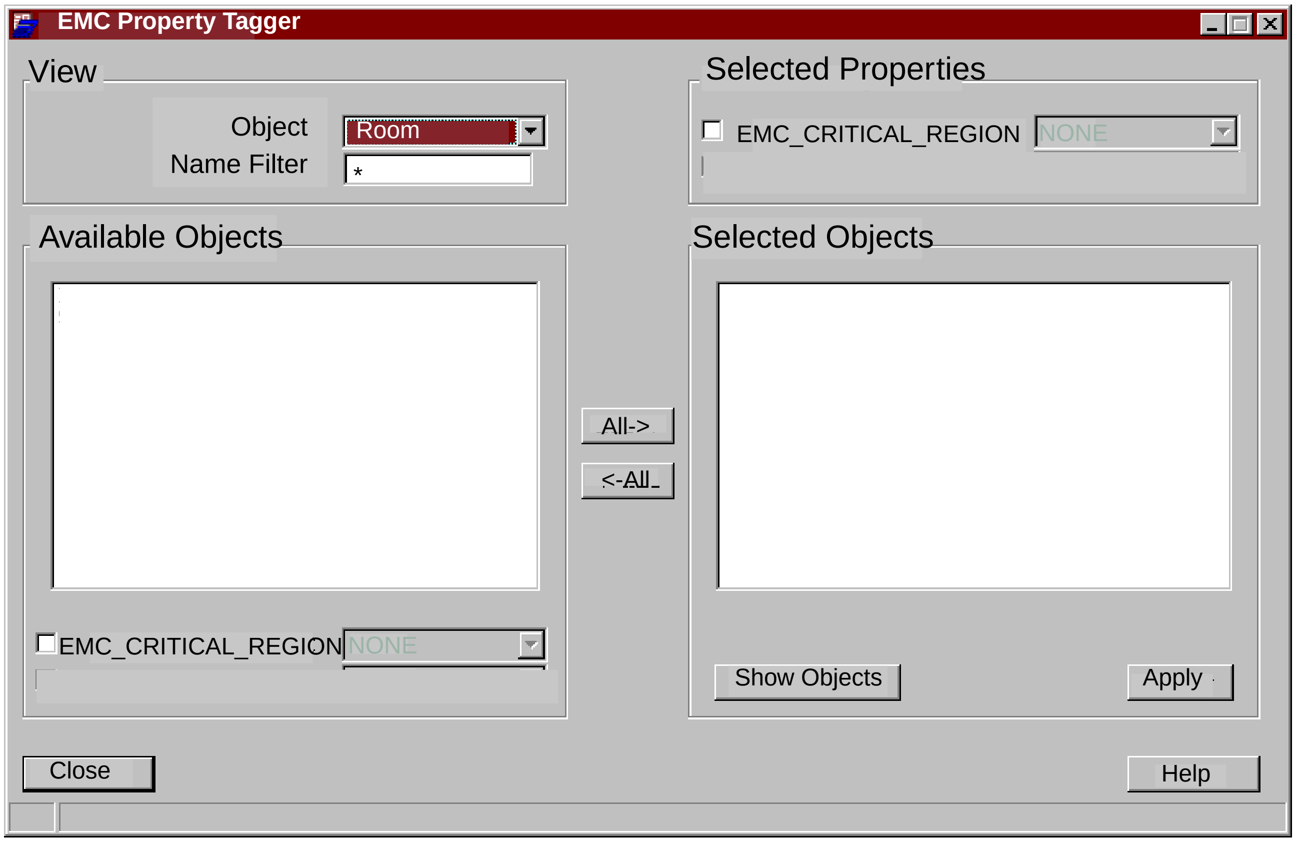
   Note: If you give a property the value `NONE`, EMC will delete the property.

Room Properties

To attach room properties

1. Select *Manual Setup* from EMC menu.

The EMC Property Tagger dialog box appears.



2. Specify the *Object* field as `room`.

   The selected rooms are loaded in the *Available Objects*.

3. Filter the available objects by specifying *Name Filter***.**

   Note: Filtering can be done on the basis of property by toggling on the properties given in the *Available Objects* list.

4. Filter the available objects further by checking the *EMC_CRITICAL_REGION* check box and setting the class.

5. Click on the room in the list.
   This transfers the room from the *Available Objects* list to *Selected Objects* list.

   Note: *All* > and < *All* buttons can be used for transferring all rooms between the *Available Objects* list and the *Selected Objects* list.

6. Specify the property value in the *Selected Properties*.

7. Click *Apply*.

All the rooms in *Selected Objects* list are tagged with the property value given in the *Selected Properties* list.

Note: If you give a property the value `NONE`, EMC will delete the property.

EMControl Variables

In a design, you may use default variables or, based upon your requirements, edit default variables to create user-defined variables.The `emc_param.par` file contains the default definitions for all user-definable EMControl variables. Your design may require that you edit the default variable values, or you may choose to use the default values as provided.

EMControl requires that you assign a `critical frequency class` property to significant or critical components and nets before you perform EMC rule checking. Critical components include ICs that are relevant to rules, which examine critical ICs, line drivers, line receivers, fast-switching transistors, or bidirectional transceivers.

Table 2-2 on page 42 describes the user-modifiable EMControl variables that are used by the EMC rules. The table lists the default variable values and lists the EMC rules that use each variable. The variables are listed in the order in which they appear in `emc_param.par`.

In Table 2-2 on page 42, a variable name that is followed by a double asterisk (**) identifies a variable that has been parameterized. A parameterized variable defines a sequence of up to five values, each of which applies to a corresponding critical frequency class definition. See Editing Parameterized Variables to Describe Critical Frequency Classes for information on modifying parameterized variables.

Table 2-2

Variable Name	Description	Default Value	Used in Rules
MIN_BYPASS_CAPS**	Minimum number of bypass capacitors for critical ICs	3 2 1 1 1	bypass_critical_IC
EMC_BYPASS_CAP_PWR_PIN_DIST	Maximum distance between the power pin and bypass capacitors	300	

Variable Name	Description	Default Value	Used in Rules
BYP_CAP_SENS_DIST**	Bypass capacitor sensitive distance	300 350 400 450 500	bypass_critical_IC bypass_drvr_rcvr_bidir bypass_fast_sw_trans bypass_pwr_trace critical_IC_loop_area critical_IC_3caps_C_2C_4C decouple_emc_regions
POWER_TRACE_SENS_DIST	Distance between power trace and bypass capacitors	300	bypass_pwr_trace
ALL_BYPASS_CAP_TYPE**	Permitted bypass capacitor types	“CAP-1:82UF:5%” “CAP-2:0.01uf:5%” “CAP-3:0.01uf:3%” “CAP-2:10uf:4%” “CAP-4:1.0uf:2%” “CAP-5:5uf:5%”	bypass_cap_type
CRITICAL_IC_BYP_CAP_TYPE**	Bypass capacitor types allowed for critical ICs	“CAP1:82UF:5%” “CAP-3:0.01uf:3%” “CAP-2:10uf:4%” “CAP4:1.0uf:2%” “CAP-5:5uf:5%” “CAP-5:5uf:5%” “CAP-5:5uf:5%	bypass_critical_IC

Variable Name	Description	Default Value	Used in Rules
EMC_CRITICAL_EXPOSED_LEN**	Maximum exposed length allowed on a critical net	2000 3000 4000 5000 6000	critical_net_exp_length
CRITICAL_TO_MHATTAN_LEN_RATIO**	Ratio of critical net length to manhattan length	1.1 1.2 1.3 1.4 1.5	critical_net_man_ratio nets_over_clean_gnd
GND_PWR_GND_SEPARATION	Minimum power and ground trace width	20	pwr_gnd_trace_width
MIN_PWR_GND_SEPARTATION	Minimum z-axis distance between power and ground plane	5	pwr_gnd_plane_separation
MAX_PWR_GND_SEPARATION	Maximize z-axis distance between power and ground plane	50	pwr_gnd_plane_separation
EXT_NET_EDGE_CRITICAL_DIST**	Minimum distance between critical nets on external layers and copper edge	500 550 600 550 700	critical_net_card_edge_dist
INT_NET_EDGE_CRITICAL_DIST**	Minimum distance between critical nets on internal layers and copper edge	300 350 400 450 500	critical_net_card_edge_dist
EMC_COMP_CONN_DISTANCE	Minimum distance between components and the connector	1000	comp_to_conn_dist

Variable Name	Description	Default Value	Used in Rules
EMC_VIA_COUNT**	Maximum number of vias for each critical net	7 8 9 10 11	critical_net_via_count
CAP1_DIST_FROM_PWR_PIN**	Distance of capacitor C1 from the power pin	300 350 400 450 500	critical_IC_3caps_C_2C_4C
CAP2_DIST_FROM_PWR_PIN**	Distance of capacitor C2 from the power pin	200 250 300 350 400	critical_IC_3ca_ps_C_4C
CAP3_DIST_FROM_PWR_PIN	Distance of capacitor C3 from the power pin	100 150 200 250 300	critical_IC_3ca_ps_C_4C
POWER_TRACE_BYPASS_DIST	Length of the power trace for which a bypass capacitor should exist	500	bypass_pwr_trace
GUARDING_DISTANCE	Maximum distance between a guard trace and a clock net	100	shield_clock_nets
COMP_CONN_MIN_DISTANCE	Minimum distance between a connector and components not connected to it	1000	comp_not_conn_dist
COMP_COMP_MIN_DISTANCE	Minimum distance between components not connected to a connector and components that are connected to it	500	comp_not_conn_dist

Variable Name	Description	Default Value	Used in Rules
LOOP_AREA_COEFFICIENT**	Maximum ratio of actual area to reference area	1.5 2.5 3.5 4.5 5.5	critical_IC_loop_area
MAX_VIA_PIN_RATIO	Maximum ratio of vias to pins on a critical net	1.5 2.5 3.5 4.5 5.5	critical_net_via_pin_ratio
PIN_PIN_RESISTANCE	Maximum pin-to-pin resistance	0.0025	max_pwr_gnd_resistance
PIN_PLANE_RESISTANCE	Maximum pin-to-plane resistance, for a design with dedicated plane layers	0.15	max_pwr_gnd_resistance
XTALK_WINDOW	Maximum distance to search for neighbor nets when calculating crosstalk	50	max_critical_net_xtalk
MAX_PEAK_XTALK**	Maximum allowed crosstalk from any individual neighbor net	0.05 0.1 0. 15 0 0.2 0.25	max_critical_net_xtalk
MAX_OVER_UNDERSHOOT**	Ratio of maximum allowed overshoot and undershoot	0.1 0.15 0.2 0.25 0.3	critical_net_ringing
BYP_CAP_SEP_DIST	Defines search area for boundaries	1500 2000 2500 3000	decouple_emc_regions
FENCE_BOUNDARY_DIST	Sensitive distance between boundary and fence	500 750 1000 1250	fence_off_emc_regions
FENCE_BOUNDARY_RATIO	Minimum overlap between fence and boundary segments	0.8 0.7 0.6 0.5	fence_off_emc_regions

Variable Name	Description	Default Value	Used in Rules
GND_SCREW_SENS_DIST_RATIO	Ratio of length to width for the area searched for a ground screw	0.25	gnd_screw_between_clock_and_conn
GUARD_TRACE_VIAS_PER_LAMBDA	Number of vias per lambda length on a guard trace	4	shield_clock_nets
MAX_CLOCK_SPECTRAL_CONTENT	Staircase limit for frequency content on critical clock nets	“0MHz:5.0V’ “30MHz:0.9V” “75MHz:0.7V” “500MHz:0.1V”	clock_spectral_content
PULSE_DUTY_CYCLE	Pulse frequency and duty cycles for EMC_CRITICAL_NET property values	“100MHz: 50% 75MHz: 50% 50MHz: 50%”	clock_spectral_content
MAX_SIMULATION_CYCLE	Number of simulation cycles performed by SigNoise	2	clock_spectral_content
MAX_RETURN_VIA_DIST	Maximum distance between jumping via and return capacitor/via	200	return_path_near_signal_via

Customizing EMControl Rule Variable

In a design, you may use default variables or, based upon your requirements, edit default variables to create user-defined variables.The `emc_param.par` file contains the default definitions for all user-definable EMControl variables. Your design may require that you edit the default variable values, or you may choose to use the default values as provided.

If you wish to change the values for EMControl variables, write them into the `emc_custom.par` file. It is a dummy file copied into your directory. Initially, it is empty. When you modify the values of any parameter, the changed values are reflected in this file.

You can also change values of certain parameters for a specific site through the site parameter file, `emc_param.par` in the site directory. If you have write permissions, this file can be written onto to reflect the modified values for that site.

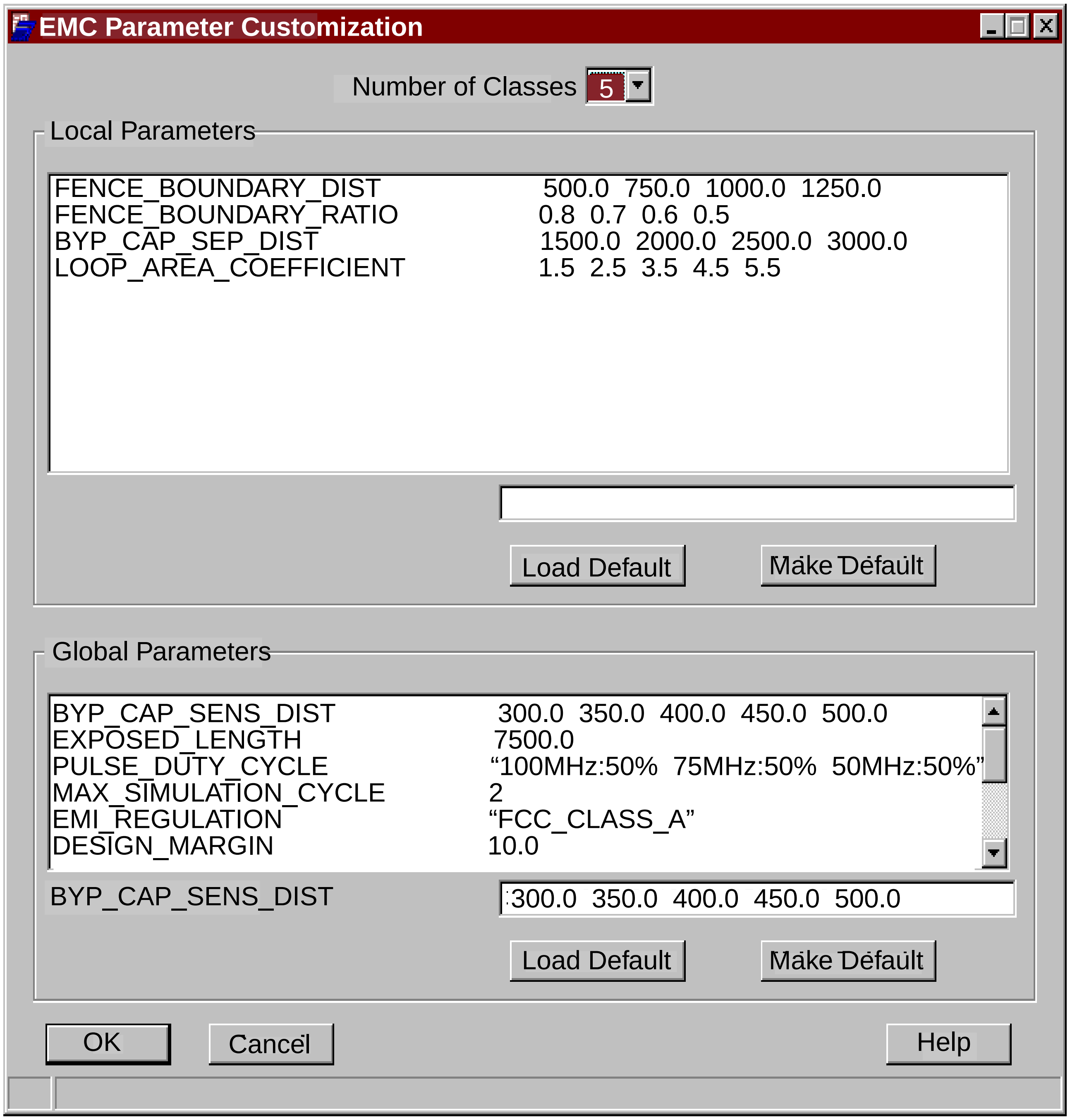
EMControl requires that you assign a critical frequency class property to significant or critical components and nets before you perform EMC rule checking. Critical components include

ICs that are relevant to rules, which examine critical ICs, line drivers, line receivers, fast-switching transistors, or bidirectional transceivers.

To customize EMControl variables:

1. Select the rules you want to customize in the tree view.
2. Click *Customize* in the EMC Rule Selection dialog box.

The EMC Parameter Customization dialog box is displayed.



FENCE_BOUNDARY_DIST	500.0 750.0 1000.0 1250.0
FENCE_BOUNDARY_RATIO	0.8 0.7 0.6 0.5
BYP_CAP_SEP_DIST	1500.0 2000.0 2500.0 3000.0
LOOP_AREA_COEFFICIENT	1.5 2.5 3.5 4.5 5.5

BYP_CAP_SENS_DIST	300.0 350.0 400.0 450.0 500.0
EXPOSED_LENGTH	7500.0
PULSE_DUTY_CYCLE	“100MHz:50% 75MHz:50% 50MHz:50%”
MAX_SIMULATION_CYCLE	2
EMI_REGULATION	“FCC_CLASS_A”
DESIGN_MARGIN	10.0

BYP_CAP_SENS_DIST 300.0 350.0 400.0 450.0 500.0

3. Select the variables from *Local Parameters*.

The selected entries are displayed in the edit box of the *Local Parameters* section.

4. Select the variables from *Global Parameters*.

The selected entries are displayed in the edit box of the *Global Parameters* section.

5. Customize the values as needed in the edit box.
6. Click on *Make Default* if you wish to keep the new settings. This writes the new values into the site parameter file.

   or

   Click on *Load Default* if you wish to keep the default settings.

   Note: You need write permissions for `emc_param.par` site parameter file to change the default settings for a site.
7. Click *OK*.

   The settings are saved in the `emc_param.par` file and closes the EMC Parameter Customization dialog box.

Note: If you change values of certain parameters and click *OK* directly, the changes are reflected in the `emc_custom.par` file, which is user-specific.

Editing Parameterized Variables to Describe Critical Frequency Classes

The EMC rules use parameterized variables to associate a sequence of variable definitions with corresponding categories, or classes, of critical ICs, nets, and regions. The definition for a parameterized variable consists of a string of up to five values, one value for each critical frequency class of ICs, nets, or regions that is available for EMC rule checking. Table 2-3 on page 51, Table 2-4 on page 51, and Table 2-5 on page 52 list the default values associated with critical IC, net, and region classes, respectively.

The default `emc_param.par` file, EMControl provides five class definitions each for critical ICs and nets. For example, in Table 2-4 on page 51, the default `LOOP_AREA_COEFFICIENT` variable used to calculate the constraint area for a critical net is defined as follows:

```
#define LOOP_AREA_COEFFICIENT "1.5   2.5   3.5   4.5   5.5"
```

In this example, the first value (1.5) defines the tightest constraint and applies to nets tagged as CLASS1 nets. Intermediate values define increasingly more relaxed constraints and apply to CLASS2, CLASS3, and CLASS4 nets, respectively. The fifth value (5.5) defines the most relaxed constraint and applies to CLASS5 nets.

In other words, the class-related variables have been defined in such a manner that the class number (1, 2, 3, 4, or 5) corresponds to the position (first, second, third, fourth, or fifth) in the variable definition of the corresponding value.

Table 2-3 on page 51 describes the default values for all five default critical IC classes. These values define the limits placed on the critical ICs based on the designation of the critical frequency class.

Table 2-3 Default Variable Values for Critical IC Classes

Variable Name	Class 1 Value	Class 2 Value	Class 3 Value	Class 4 Value	Class 5 Value
MIN_BYPASS_CAPS	3	2	1	1	1
BYP_CAP_SENS_DIST	300	350	400	450	500
CRITICAL_IC_BYP_CAP_TYPE	“CAP-1:82UF:5%! CAP-3:0.01uf:3%”	“CAP-2:10uf:4%”	“CAP-4:1.0uf:2%! CAP-5:5uf:5%”	“CAP-5:5uf:5%”	“CAP-5:5uf:5%”
CAP1_DIST_FROM_PWR_PIN	300	350	400	450	500
CAP2_DIST_FROM_PWR_PIN	200	250	300	350	400
CAP3_DIST_FROM_PWR_PIN	100	150	200	250	300

Table 2-4 on page 51 describes the default values for all five default critical net classes. These values define the limits placed on the critical nets based on the designation of the critical frequency class..

Table 2-4 Default Variable Values for Critical Net Classes

Variable Name	Class 1 Value	Class 2 Value	Class 3 Value	Class 4 Value	Class 5 Value
EMC_CRITICAL_EXPOSED_LEN	2000	3000	4000	5000	6000
CRITICAL_TO_MHATTAN_LEN_RATIO	1.1	1.2	1.3	1.4	1.5
EXT_NET_EDGE_CRITICAL_DIST	500	550	600	650	700
INT_NET_EDGE_CRITICAL_DIST	300	350	400	450	500
EMC_VIA_COUNT	7	8	9	10	11
LOOP_AREA_COEFFICIENT	1.5	2.5	3.5	4.5	5.5

Table 2-4 Default Variable Values for Critical Net Classes

Variable Name	Class 1 Value	Class 2 Value	Class 3 Value	Class 4 Value	Class 5 Value
MAX_PEAK_XTALK	0.05	0.1	0.15	0.2	0.25
MAX_OVER_UNDERSHOOT	0.1	0.15	0.2	0.25	0.3
MAX_VIA_PIN_RATIO	1.5	2.5	3.5	4.5	5.5

Table 2-5 on page 52 describes the default values for all five default critical region classes. These values define the limits placed on the critical EMC regions in the design based on the designation of the critical frequency class of the IC and the net.

Table 2-5 Default Variable Values for Critical Region Classes

Variable Name	Class 1 Value	Class 2 Value	Class 3 Value	Class 4 Value	Class 5 Value
BYP_CAP_SENS_DIST	300	350	400	450	500
BYP_CAP_SEP_DIST	1500	2000	2500	3000	
FENCE_BOUNDARY_DIST	500	750	1000	1250	
FENCE_BOUNDARY_RATIO	0.8	0.7	0.6	0.5	

Use the EMC Auto Property Tagger form to change the number of available frequency classes or to change the voltage swing and rise and fall time boundary values between critical frequency classes. See Automatically Attaching Properties to Design Objects for more information.

To change the critical class definitions defined by parameterized variables

1. Modify the variable values associated with the classes you want to change.
2. Modify values for parameterized variables by editing the variable's string of values in the `emc_param.par` file.

   When editing variable values, keep the following points in mind:

- The position of a value in the value string must be equivalent to the position of the corresponding class name – CLASS1 through CLASS5.
- The number of values defined for a parameterized variable must equal the number of defined classes.

- Ensure that you leave at least one space (no newlines) between value definitions.
- If the value itself contains a double quote (`“ or ”`) character, append a backslash character (\) before the double quote character.

For example, in Table 2-4 on page 51, the critical net variable EMC_CRITICAL_EXPOSED_LEN is defined as follows:

```
#define EMC_CRITICAL_EXPOSED_LEN      “2000 3000 4000 5000 6000”
```

You can increase the value associated with critical frequency CLASS1 by editing `emc_param.par` to change the first value, 2000, to 2025.

```
#define EMC_CRITICAL_EXPOSED_LEN      “2025 3000 4000 5000 6000”
```

In this example, any nets that you identify as belonging to CLASS1 will have an EMC_CRITICAL_EXPOSED_LEN value of 2025. Nets in CLASS2 have a value of 2050. Nets belonging to CLASS3 have a value of 3000.

You can modify the values for the parameterized variables (which are described in Table 2-2 on page 42 and defined in `emc_param.par`) to suit your requirements. Your EMC expert can provide information on optimal variable values for your application.

Identifying ICs and Nets in a Class

You need to identify the critical frequency class to which each significant IC and net belongs before you run related EMC rules.

Each EMC rule will use the index value, determined using the class name that you specify, to find the value for any parameterized variables that the rule uses.

To identify the class of each critical IC

➤ Use the appropriate class definition to define the `EMC_CRITICAL_IC` property for each critical IC in your design.

  For example, all critical ICs that belong to CLASS1 must have the property `EMC_CRITICAL_IC` set to the value `CLASS1`.

To identify the class of each critical net

➤ Use the appropriate class definition to define the EMC_CRITICAL_NET property for each critical net in your design.

  For example, all nets that belong to CLASS3 must have the property EMC_CRITICAL_NET set to the value CLASS3.

For information on how to assign the properties required by EMControl, see Automatically Attaching Properties to Design Objects or Manually Attaching Properties to Selected Design Objects

3

Performing EMControl Rule Checking

EMC Rules

EMControl provides six sets of EMC rules:

- Placement Rules
- Bypass Rules
- Power and Plane Ground Rules
- DC Routing Rules
- Signal Routing Rules
- Signal Quality Rules

The EMC rule sets are used at different stages of design development. For reference information on the rule sets and for detailed rule descriptions, see Appendix B, "EMControl Rules,". Rule descriptions are arranged alphabetically within each rule set. Each rule description includes requirements (design setup, variable values, and property assignments) and other helpful information.

Checking for EMC During Placement

Before you perform EMC Rule checking, ensure:

- Stackup and constraints are defined.
- Pre-route signal integrity checking has been performed.
- EMControl setup has been audited and any problems encountered are resolved.

EMC analysis performed during component placement includes:

- Initial Placement Checks

- Detailed Placement Checks

Initial Placement Checks

The initial placement checks serve to verify EMControl tool setup and design preparation. Efficient placement strategy generally involves placing critical components, including external connectors, first. After critical components are placed, design analysis can begin.

The results of the first EMC rule-checking run provides you baseline EMC information about your design.These reports can be used to locate EMC problem areas on the board and to estimate the extent to which your design requires modification to facilitate EMC compliance.

Cadence recommends that you run both thermal and signal integrity analysis prior to checking EMC rules. If thermal and signal integrity issues are not resolved before you check EMC rules, your design is not likely to pass the EMC check.

Use the following rules during the initial placement EMC rule-checking runs:

1. The placement rules in the `emc_placement.rle` file.
2. The bypass rules in `the emc_bypass.rle` file.

Detailed Placement Checks

Depending on requirements, you can perform EMC checks at any time throughout the placement stage. It is particularly recommended that you perform pre-route analysis after the placement is complete.

You can use these iterative checks to begin trade-offs between constraints and to adjust property setup including EMControl property values assigned to critical components, nets, and rooms. You can also use this information to determine power and ground plane design. As you perform the iterative EMC checks, tighten electrical constraints to approach EMC requirements.

Use the following rules during the detailed placement EMC rule-checking runs:

1. The placement rules in the `emc_placement.rle` file
2. The bypass rules in the `emc_bypass.rle` file

Checking for EMC While Routing

After placement is complete, routing of critical nets (such as clock nets) begins. Again, depending on the requirements, the critical signal EMC checks can be performed throughout this stage.

Continue to perform constraint-driven routing during this stage as you refine constraints. Thorough analysis of critical signals also includes thermal and signal integrity checks (including crosstalk analysis).

Use the following rules during the critical signal rule-checking runs:

1. The DC routing rules in the `emc_dc_route.rle` file
2. The signal routing rules in the `emc_sig_route.rle` file

You can follow the critical signal EMC checking with post-route signal integrity analysis.

Checking for EMC After Routing

This stage of EMC checking begins after the design has been routed and signal and thermal analysis has been performed for the last time.

Post-routing activities can include refining constraints and confirming that all EMC requirements have been met. Your EMC expert can review a final EMControl report to confirm this.

Use the following rules during the post-routing rule-checking runs:

1. The power and ground plane rules in the `emc_pwr_gnd_dist.rle` file
2. The DC routing rules in the `emc_dc_route.rle` file
3. The signal routing rules in the `emc_sig_route.rle` file
4. The signal quality rules in the `emc_sig_qual.rle` file

Rule Checking Tasks

The following tasks are required to check your design for electromagnetic compliance:

- Setting Up the EMControl Run
- Defining the Scope of the Check

- Auditing EMControl Rules
- Executing EMControl Rules
- Viewing Results
- Saving Run Results

Setting Up the EMControl Run

You need to set up EMControl before you run the design checking process. The *Rule Select* option in the EMC menu lets you:

- Define the location of the parameter file, `emc_param.par`.

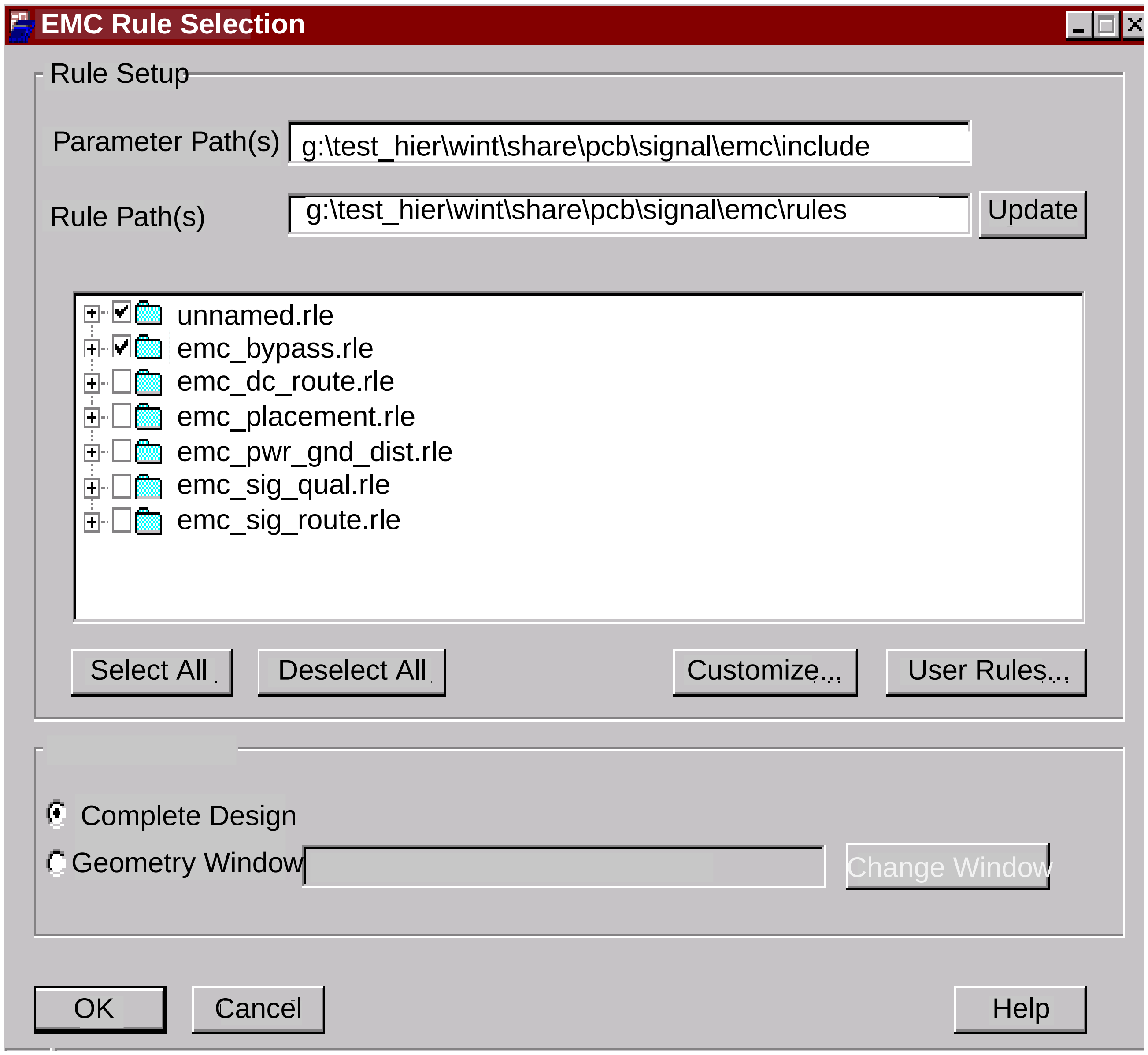
  The parameter file contains user-editable values for the EMControl variables used by the EMC rules.
- Select the rules that will be used to check the design.
- Specify the portion of the design to be checked.
- Customize Parameters
- Invoke the Rule Developer dialog box to edit rules, create new rules, or edit predicates.

Note: If you want to change any of the system defaults that define the EMControl environment, be sure to edit the EMC Initialization dialog box before you run EMControl checking.

To set up the rules that will be used to check the design:

➤ Select *Rule Select* in the EMC menu.

The EMC Rule Selection dialog box is displayed. This dialog box lets you set up EMControl.


EMC Rule Selection

Rule Setup

Parameter Path(s): g:\test_hier\wint\share\pcb\signal\emc\include

Rule Path(s): g:\test_hier\wint\share\pcb\signal\emc\rules

- [✓] unnamed.rle
- [✓] emc_bypass.rle
- [ ] emc_dc_route.rle
- [ ] emc_placement.rle
- [ ] emc_pwr_gnd_dist.rle
- [ ] emc_sig_qual.rle
- [ ] emc_sig_route.rle

[✓ Complete Design] Geometry Window

Note: The EMC expert at your site may have created custom rule files that are also available for checking your design.

When you open the EMC Rule Selection dialog box for the first time, only the names of the compiled rule files in the default search path are displayed. The files are in the unexpanded condition (that is, no groups or rules within that file are visible). At any other time, the browser is displayed in its state at the end of the last working session; in addition, any rule files that you have added to the search path since the last session are displayed in their unexpanded condition.

EMControl rules use variable definitions that you can modify. For information on the rule variables and how and when to modify variable values, see Chapter 2, “Setting Up the EMControl Environment,”.

To change the display of rules in the rule browser

1. Specify the path to the directory where the rule files are located. Compiled rule files have the file name extension `.rle`.
2. Update the dialog box to show the rule files in the specified path.

   For further information, see Adding Rule Files to the Rule Browser on page 60. For information on expanding a rule file to display the rules within the file, see Displaying the Rules in a File on page 61.

When EMControl checks your design for EMC problems, it uses only the rules that you have selected in the rule browser.

Identifying the Location of Rule Parameter Files

If you are using custom rules at your site, you must specify the path for any parameter files that you are including in your rule files, if they are not located in your current directory. Parameter files define variables that are being used by EMControl rule files.

1. Type the paths of the parameter files in the *Parameter Path(s)* field at the top of the Rule Selection dialog box.
2. Separate the paths of parameter files with a space.

An example of a parameter file is `emc_param.par`. This file defines the user-definable variables used by the EMControl rules. The `emc_param.par` file is specified in the default rule file by means of an `include` statement. By default, the `emc_param.par` file is located in the system include directory. The path for the `include` directory is the default definition for the *Parameter Path(s)* field. For more information on the default EMControl directory structure, see Chapter 2, “Setting Up the EMControl Environment,”

Adding Rule Files to the Rule Browser

You can update the default display in the rule browser with additional rule files. EMControl searches for compiled rule files in the directories located in the search path, which you specify in the EMC Rule Selection dialog box. You need to modify the default search path definition if the files that you are adding are at another directory location.

If you are using customized rule and parameter files, ensure you have the requisite change permissions for these files.

To change the display of rules in the rule browser:

1. Provide one or more search paths in the *Rule Path(s)* field.
   - ❑ Specify either full or relative paths.
   - ❑ Separate each entry with a space.
2. Click on *Update* to display all the `.rle` files in the search path.

   EMControl updates the rule browser with the names of all the compiled rule files in the paths. When a new rule file is added to the browser list, it is displayed in an unexpanded condition (that is, no groups or rules within that file are visible). Otherwise, currently selected rules and expanded rule files are redisplayed in the same condition as before the update, as long as the rule file that contains them is still in the path.

   When the list of rules extends beyond the browser window in either the vertical or the horizontal direction, a scroll bar lets you view the remaining rules.

Displaying the Rules in a File

You can choose either to display all the rules in a rule file (expand the rule), or to turn off (collapse) their display.

To display all the rules in a file when only the file name is currently visible

➤ Click on the + sign of the tree view.

The tree view expands to show all the rules in the rule file. All the rules are displayed either selected (that is, highlighted) or deselected, according to the setting of the file name when you expanded it.

To turn off the display of all the rules in a file

➤ Click on the - sign of the tree view.

The tree view contracts to remove from the display all the rules in the rule file. The file name remains either selected or deselected, according to the setting of the file name when you unexpanded it.

Getting Help on a Rule

To read help information for a rule in the rule browser

➤ Click on the rule name.

A window opens with reference information on the rule.

Selecting Rules to Use for Design Checking

The following conventions apply to selecting and deselecting rule files:

- Selecting the name of a rule file automatically selects all the rules in the corresponding file for checking, whether the individual rules are currently visible or not.
- Deselecting rules by file name deselects all the rules included in that file.

Selecting All Rules

➤ Click on *Select All* in the EMC Rule Selection dialog box.

All the rule files and rule names in the browser are selected.

Deselecting All Rules

➤ Click on *Deselect All* in the EMC Rule Selection dialog box.

All the rule files and rule names in the browser are deselected.

Selecting or Deselecting All Rules in a file

➤ Click the check box corresponding to the file name.

If the check box of the file name is currently checked, it is unchecked. All rules within that file are also unchecked. If it is not currently checked, it is selected. All rules within that file are also checked.

Selecting or Deselecting a Single Rule

➤ Click the check box corresponding to the rule name.

If the rule is currently checked, it is unchecked. If it is not currently checked, it is checked.

Defining the Scope of the Check

You can perform EMControl design checking for:

- The complete design
- A selected window

If you run EMControl for the entire design, you may end checking rules against entire areas of the design where they do not apply. This increases rule checking time. Therefore, it is recommended that you check the design by defining windows, then selecting for each run only the rules that are relevant to that portion of the design.

To check the entire design:

➤ Select *Complete Design* in the *Scope of Check* section of the EMC Rule Selection dialog box.

To check a portion of the design

1. Select *Geometry Window* in the *Scope of Check* section of the EMC Rule Selection dialog box.
2. Click on *Change Window*, at the right edge of the Scope of Check section.

   You now need to specify the extent of the design area to check (the bounding box).
3. Click in the layout window to select a corner of the bounding box.
4. Click again to select the opposite corner (to define the diagonal extent) of the bounding box.

   The coordinates that you selected are displayed in the display-only field to the right of the *Geometry Window* button.

   If you do not specify a bounding box for the rule check, EMControl uses the coordinates of the previous check run, if any. If you have not defined coordinates and none of them are available from the previous run, EMControl checks the entire design.

Auditing EMControl Rules

Cadence recommends that you verify that you have fulfilled the property requirements for the selected rules before you execute rule checking.

The Audit Report describes the properties that have not been properly configured for the EMC rules you are using for design checking. The Audit Report reflects the contents of the ASCII file called `emcrc_verify.msg`.

Before you run EMC rule checking, generate an Audit Report to verify that your design is properly set up for the EMC rules you are checking against your design. EMControl generates an Audit Report whenever you execute an *Audit* command.

To generate an Audit Report:

1. Select the rules using the Rules Selection dialog box.
2. Select *Audit* from the EMC menu.

The Audit Report is displayed in the EMC Report window.

Report information assists you in locating and correcting rule violations. The Audit report contains the following sections:

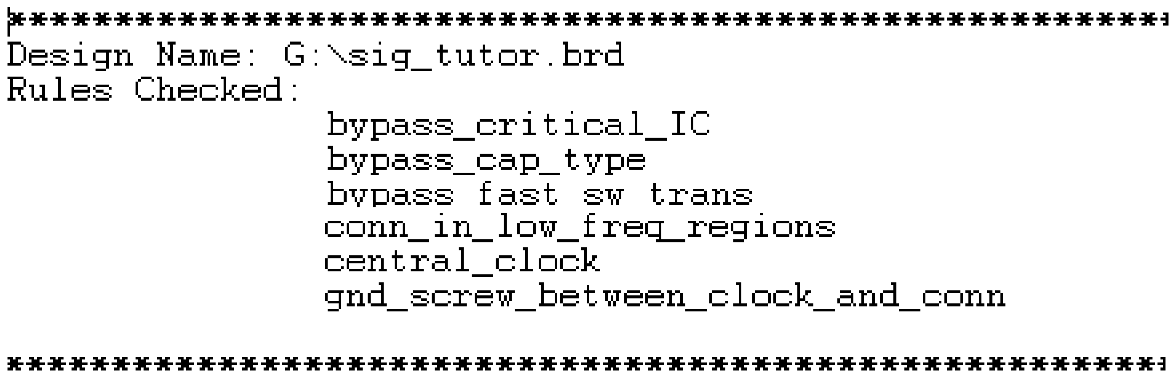
- Header Information
- Rule-Specific Information
- Summary Information

Header Information

The upper portion of an Audit Report

- Lists the complete path of the checked design
- Lists the names of the rules that you checked

The following is an example of a report header:



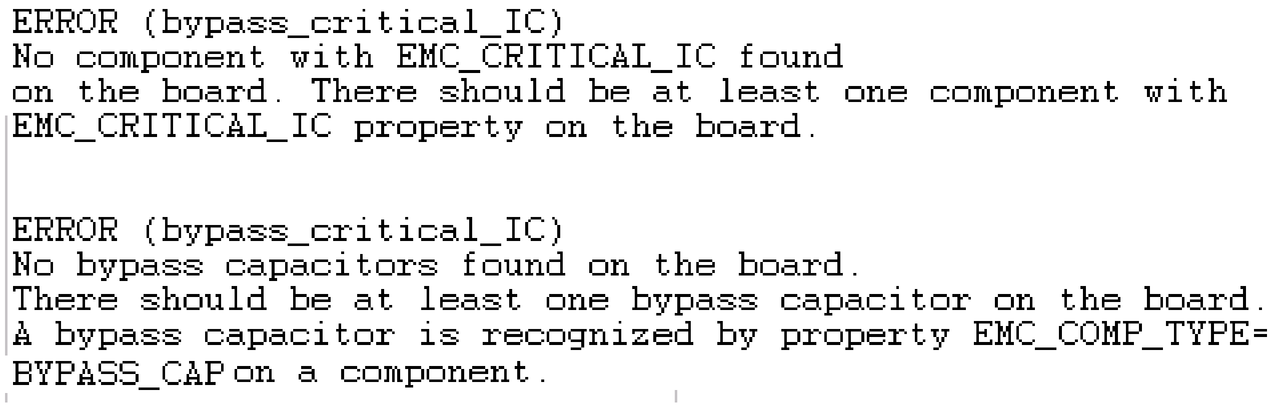
```
******************************************************
Design Name: G:\sig_tutor.brd
Rules Checked:
               bypass_critical_IC
               bypass_cap_type
               bypass fast sw trans
               conn_in_low_freq_regions
               central_clock
               gnd_screw_between_clock_and_conn

******************************************************
```

Rule-Specific Information

The body section of the Audit Report describes the short message (if it exists) and the advisor messages for each violation. These messages list:

- The severity level of the violation
- The name of the rule
- Information about the rule violation

The following is an example of the body section of an Audit Report:

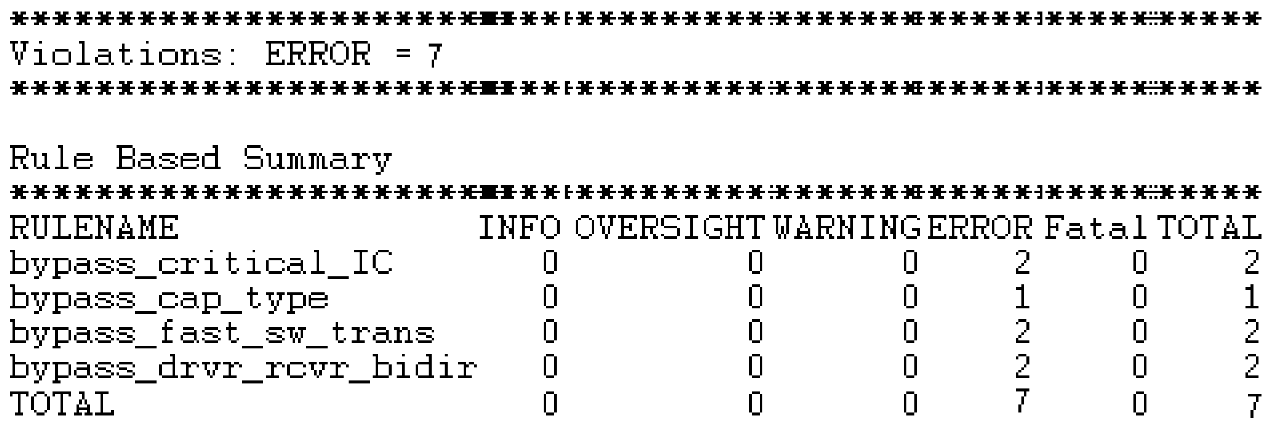


```
ERROR (bypass_critical_IC)
No component with EMC_CRITICAL_IC found
on the board. There should be at least one component with
EMC_CRITICAL_IC property on the board.


ERROR (bypass_critical_IC)
No bypass capacitors found on the board.
There should be at least one bypass capacitor on the board.
A bypass capacitor is recognized by property EMC_COMP_TYPE=
BYPASS_CAP on a component.
```

Summary Information

The bottom section of the Audit Report provides a summary of the total number of rule violations and the number of violations recorded for each rule, by severity.

The following is an example taken from the body of an Audit Report:



```
************************************************************
Violations: ERROR = 7
************************************************************

Rule Based Summary
************************************************************
RULENAME                INFO OVERSIGHT WARNING ERROR Fatal TOTAL
bypass_critical_IC         0         0       0     2     0     2
bypass_cap_type            0         0       0     1     0     1
bypass_fast_sw_trans       0         0       0     2     0     2
bypass_drvr_rcvr_bidir     0         0       0     2     0     2
TOTAL                      0         0       0     7     0     7
```

Executing EMControl Rules

To execute EMC rules

➤ Select *Execute* from the EMC menu.

EMControl starts checking the current design against the rules you selected. The progress of rule execution will be available in the Allegro/SPECCTRAQuest CWI.

The following files are created in the EMC run directory during design checking:

- ❑ `emcrc_execute.msg`
- ❑ `emcrc.mkr`
- ❑ `emcrc.log`

The results file is updated automatically, so it contains the violation messages that were generated during the most recent run.

To abort EMC Rule execution:

➤ Enter *Control-C* in the layout command window to abort EMControl checking.

EMControl checking is terminated.

-or-

➤ Click *Stop* in the Allegro/SPECCTRAQuest status window.

Note: If you abort EMC rule execution, the EMControl files will be left in an unknown state.

Viewing Results

To view the results of EMControl setup verification and rule checking, you can:

- View the list of violation messages found during rule checking.
- Highlight the object in your layout that corresponds to a particular violation.
- View the details of a violation, including a description of the rule that was violated.
- Open a report that summarizes the results of either property auditing or design checking.
- Open a results text file to view descriptive details for each violation message.

For detailed information on viewing and assessing violations in your design, see Chapter 4, “Resolving EMC Rule Violations in Your Design.”

Saving Run Results

The results of setup verification and EMC checking are automatically saved to the run directory that you specified in the EMC Initialization dialog box.

The results from the *Execute* command are stored in the `emcrc.mkr` and `emcrc_execute.msg` files. Having a separate directory for each check run allows you convenient access to the results of previous runs. You can use these directories to compare the results between subsequent check runs.

The report results from the `Audit` command are stored in the `emcrc_verify.msg` file.

4

Resolving EMC Rule Violations in Your Design

Overview of Rule-Checking

When EMC rule checking is complete, you can display the results from the EMC menu. EMControl displays the list of messages for all the violations that were found, and lets you view the objects (pins, nets, or components) in your design that correspond to each violation. EMControl can display up to 1000 messages in the violations list.

EMControl uses colored markers to highlight the objects that are in violation of your selected rules. As you select each violation message, the corresponding marker is displayed in a contrasting color. When more than one object is found for a violation, a pop-up menu lets you select which object to highlight.

As you examine each violation, you can correct it within your design. When you have resolved all violations, you can check the design again, repeating the process as often as necessary.

EMControl also creates a markers file called `emcrc.mkr` when it finds rule violations. The file contains detailed information about each violation. EMControl uses the contents of this file when you request a display of results from the EMC menu.

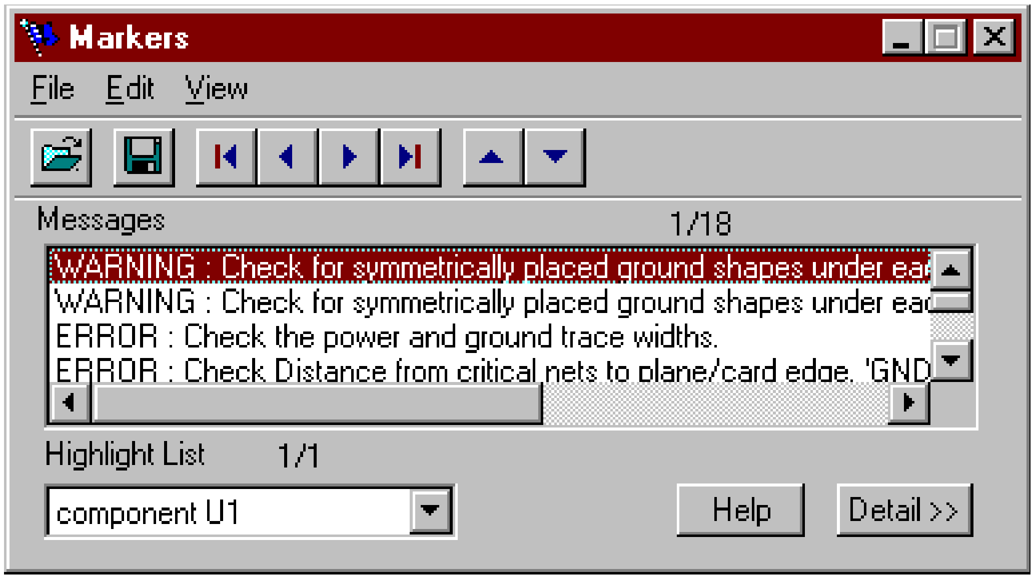
Viewing the Results of Rule-Checking

This section shows you how to locate violations in your design. In addition to viewing violations in the design, you can check the `emcrc_execute.msg` or `emcrc.mkr` file to determine where violations were found in your design, or you can read a formatted report.

To view the results of design checking

➤ Select *Results* in the EMC menu.

The Markers dialog box is displayed.



WARNING : Check for symmetrically placed ground shapes under ea
WARNING : Check for symmetrically placed ground shapes under ea
ERROR : Check the power and ground trace widths.
ERROR : Check Distance from critical nets to plane/card edge. 'GND

component U1

Viewing a Violation

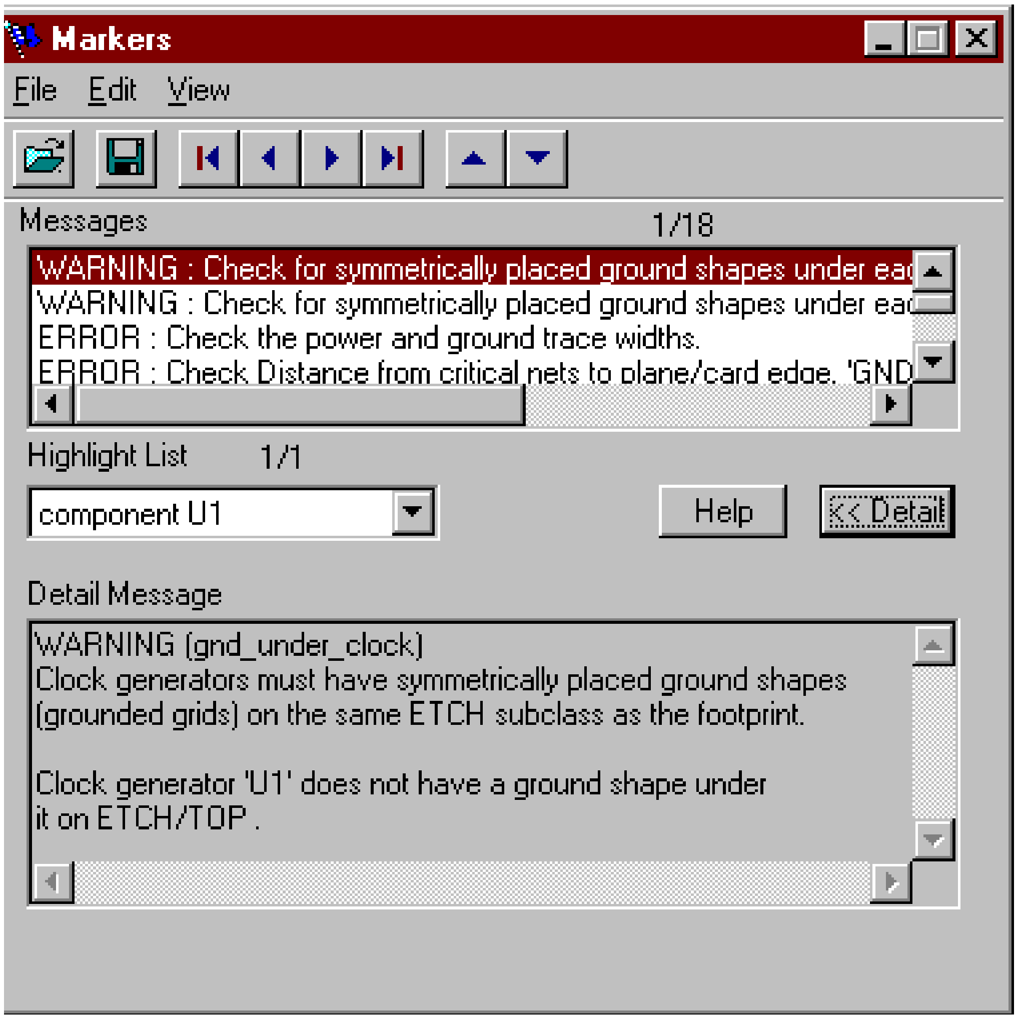
1. Click on the violation message in the *Messages* list of Markers dialog box.

   The selected message is highlighted in the list.

   Note: The left-right arrow icons in the markers toolbar can be used to traverse the list of violations.

2. Click *Detail*.

The details of a violation message appears in the markers *Detail Message* window.



WARNING : Check for symmetrically placed ground shapes under ea
WARNING : Check for symmetrically placed ground shapes under ea
ERROR : Check the power and ground trace widths.
ERROR : Check Distance from critical nets to plane/card edge. 'GND

component U1

WARNING (gnd_under_clock)
Clock generators must have symmetrically placed ground shapes (grounded grids) on the same ETCH subclass as the footprint.

Clock generator 'U1' does not have a ground shape under it on ETCH/TOP .

Highlighting Objects

1. Select the violation message in the Markers dialog box.

   The first object associated with it is automatically highlighted in Allegro/SPECCTRAQuest.

2. Select an object from the *Highlight List* combo box that lists all the violating objects of a violation.

   Highlights the selected object in the Allegro window.

   Note: You can also use the up-down arrow icons in the toolbar to traverse and highlight the violating objects of a message.

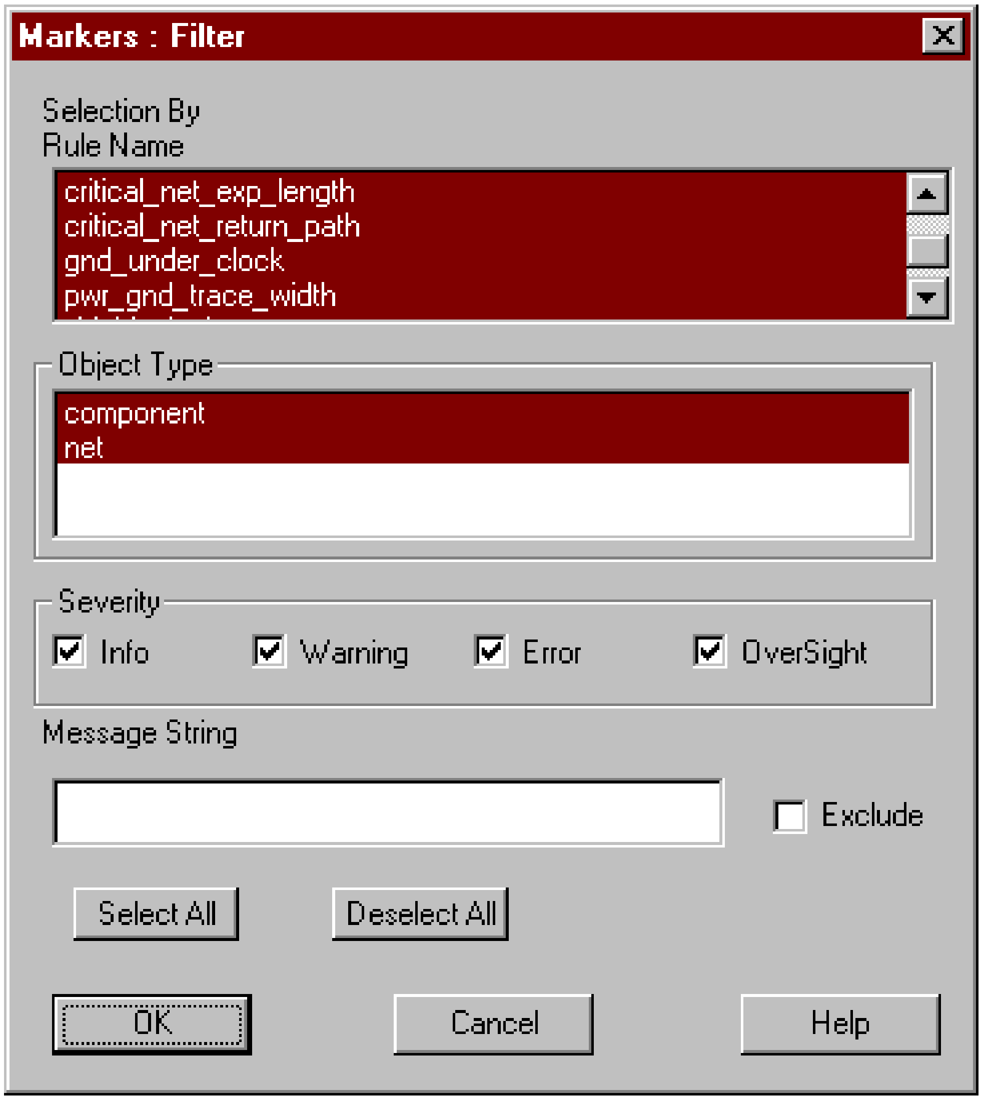
Filtering Violations

The EMC Filters dialog box lets you control which violation messages are displayed in the messages browser. You can customize the display of messages in the browser by rule name, object type, and severity level. You can also elect to display, or not display, messages that match a particular regular expression.

To open the Filters dialog box

➤ Select *View -> Filter Options*... from the Markers menu.

The *Filter* dialog box appears.



Markers : Filter

Selection By
Rule Name
- critical_net_exp_length
- critical_net_return_path
- gnd_under_clock
- pwr_gnd_trace_width

Object Type
- component
- net

Severity
- [✓] Info
- [✓] Warning
- [✓] Error
- [✓] OverSight

Message String

[ ] Exclude

Filtering Messages by Rule Name

You can display violations by the name of the rule associated with them. All rule names are displayed in a list box at the top of the EMC Filters dialog box.

To control the display of violations by rule name

➤ Select (highlight) only the names of the rules for which you want to display violations.

Violations are displayed for the rule if its name is highlighted in the box. By default, all rules are selected.

- ❑ Use *Select All* to highlight all rule names.
- ❑ Use *Deselect All* to remove highlighting from all rule names.
- ❑ Click on a selected rule name to deselect it.
- ❑ Click on a deselected rule name to highlight and select it.

Filtering Messages by Object Type

You can display violations by object type. For example, you might want to view only violations on components or on pins.

To control the display of violations for objects of a particular type

➤ Select (highlight) only the object types for which you want to display violations.

Violations are displayed for the object type if its name is highlighted in the box. By default, all object types are selected.

- ❑ Use *Select All* to highlight all object types.
- ❑ Use *Deselect All* to remove highlighting from all object types.
- ❑ Click on a selected object type to deselect it.
- ❑ Click on a deselected object type to highlight and select it.

Filtering Messages by Severity Level

You can display violations by severity level. For example, you might want to view only messages for violations labeled `ERROR`.

To control the display of violations by severity level

➤ Check the box to the left of one or more Severity definitions.

- ❑ Violations are displayed for a severity if its box is checked.
- ❑ No violations are displayed for a severity if its box is unchecked.

Filtering by Regular Expression

You can use the Message String field to specify a UNIX regular expression to use as a filter criterion for the display of violations. For example, you might want to display only messages that contain `ground` or to exclude from display all messages that refer to `output pins`.

To filter by regular expression

1. Type a regular expression in the *Message String* field.
2. If you want the regular expression to define messages to exclude from display, check *Exclude*.

Examples

To search for messages that contain `ground` anywhere in the message

➤ Type the following in the Message String field:

```
ground
```

To filter out of the messages browser any messages that contain `output pins`

➤ Type the following in the Message String field and check *Exclude*:

```
output pins
```

To search for messages that contain `output` and `pins` (in order, but not necessarily together)

➤ Type the following in the Message String field:

```
output .*pins
```

Applying Your Filtering Selections to the Browser Display

To apply your filtering definition

➤ Select *OK* in the EMC Filters dialog box.

The dialog box closes. The messages browser in the EMC Results dialog box is updated to reflect your selections. If none of the violations pass your filtering criteria, a message informs you that all markers are filtered out.

Cross-Probing Multiple Violations

The Markers dialog box lists multiple violations.

1. Click the first violation to highlight it.
2. Select *Detail* in the Markers dialog box to bring up a detailed description of the highlighted violation.
3. Click *Highlight List* in the Markers dialog box.

   This will open a combo box which lists the offending elements corresponding to the selected violation. Select one of the elements from the combo box. The selected object is highlighted in Allegro/SPECCTRAQuest.

   You can use the toolbar up or down icons to traverse through the list of violations. As you select each message, the detailed violation description in the *EMC Advisor* dialog box is automatically updated. The first element in the *Highlight List* of each violation is highlighted as and when you traverse. Tool bar icons are provided to traverse the *Highlight List* also.

   The numeric designator near the *Highlight List* indicates whether there are multiple objects associated with the current message or not. The second number on the button specifies how many objects are there for this violation, and the first number indicates which object of the sequence is currently highlighted in the design. You can use up or down icon to Markers toolbar to traverse *Highlight List*.
4. Select *Close* in the Markers dialog box when you have finished viewing violations.

Loading a Markers File

1. Select *File > Load* from the Markers menu to invoke the Marker File browser.
2. Select the markers file using the file browser
3. Click *Open*.

   The main markers window is loaded with messages from the specified markers file.

   Note: The corresponding design must be loaded in Allegro/SPECCTRAQuest when you load a marker file.

Hiding a Violation Message

You can remove a violation from the messages browser in the Markers form. You might choose to remove a violation from view if you have determined that the violation is actually acceptable in your design or if you have corrected the violation.

To remove a violation in the messages browser

1. Select the violation message to remove.

   The message is highlighted.

2. Click *Edit > Delete Marker*.

The message is hidden from the messages browser, and the associated object is unhighlighted in the layout. The display in the upper right corner of the EMC Results form is updated to reflect the change in the total number of messages in the browser.

Note: This removes the violation only from the messages browser. The violation is not deleted from the markers file. To permanently remove from the markers file the violations that you have hidden from the browser display, see “Saving a Markers File.”

Saving a Markers File

You can update a markers file to permanently remove any violations that are currently hidden from the messages browser.

To permanently remove hidden messages

➤ Select *File > Save* in the Markers form.

The updated markers file is saved.

Note: All violations that you removed from display using *Edit > Delete Marker* are permanently erased from the saved file.

Reading an Execute Report

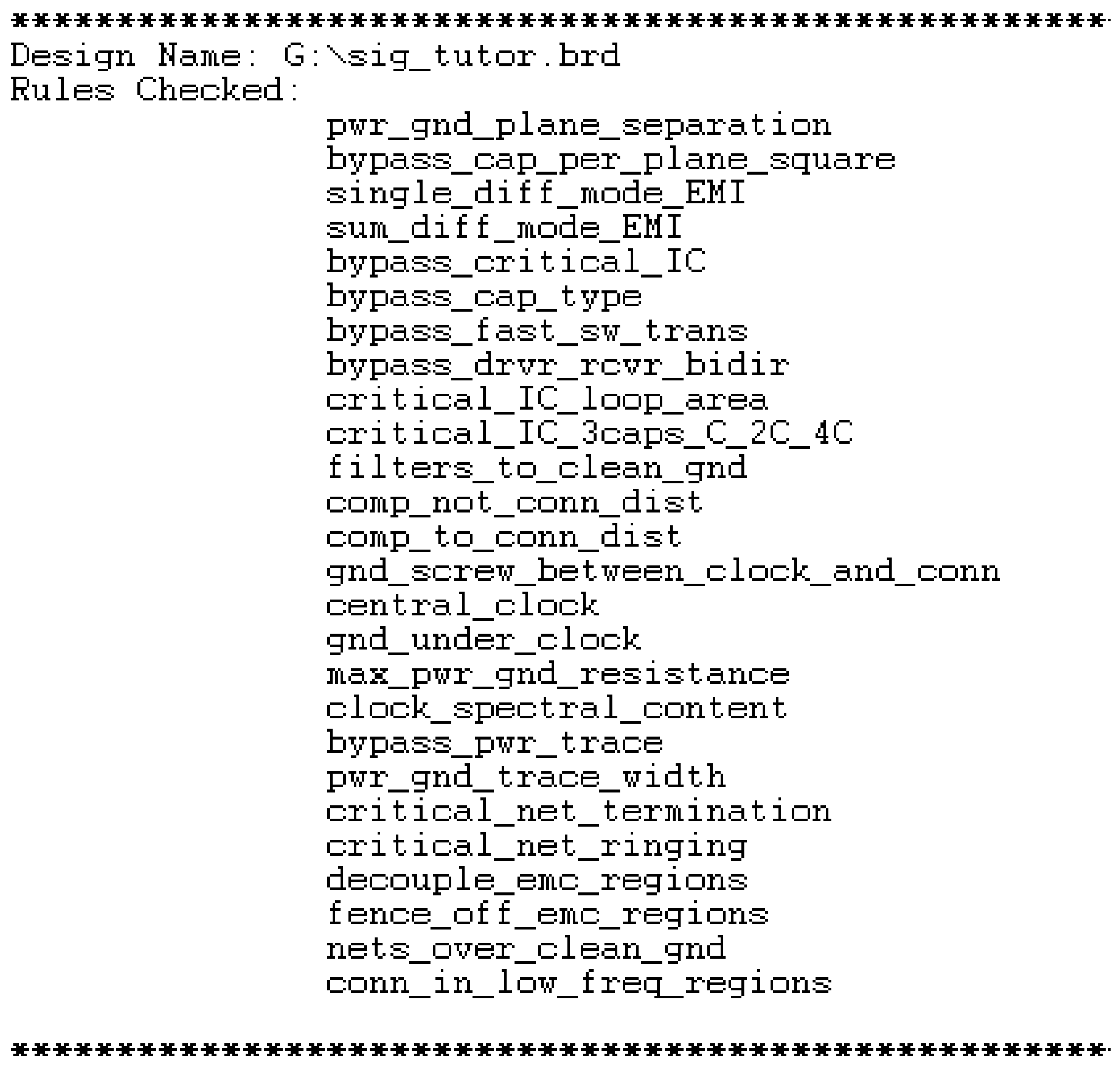
The following sections describe the format of the Execute report.

Header Information

The upper portion of an Execute report

- Tells you which design was checked
- Lists the names of the rules that you checked

The following is an example of a report header:



```
****************************************************
Design Name: G:\sig_tutor.brd
Rules Checked:
               pwr_gnd_plane_separation
               bypass_cap_per_plane_square
               single_diff_mode_EMI
               sum_diff_mode_EMI
               bypass_critical_IC
               bypass_cap_type
               bypass_fast_sw_trans
               bypass_drvr_rcvr_bidir
               critical_IC_loop_area
               critical_IC_3caps_C_2C_4C
               filters_to_clean_gnd
               comp_not_conn_dist
               comp_to_conn_dist
               gnd_screw_between_clock_and_conn
               central_clock
               gnd_under_clock
               max_pwr_gnd_resistance
               clock_spectral_content
               bypass_pwr_trace
               pwr_gnd_trace_width
               critical_net_termination
               critical_net_ringing
               decouple_emc_regions
               fence_off_emc_regions
               nets_over_clean_gnd
               conn_in_low_freq_regions

****************************************************
```

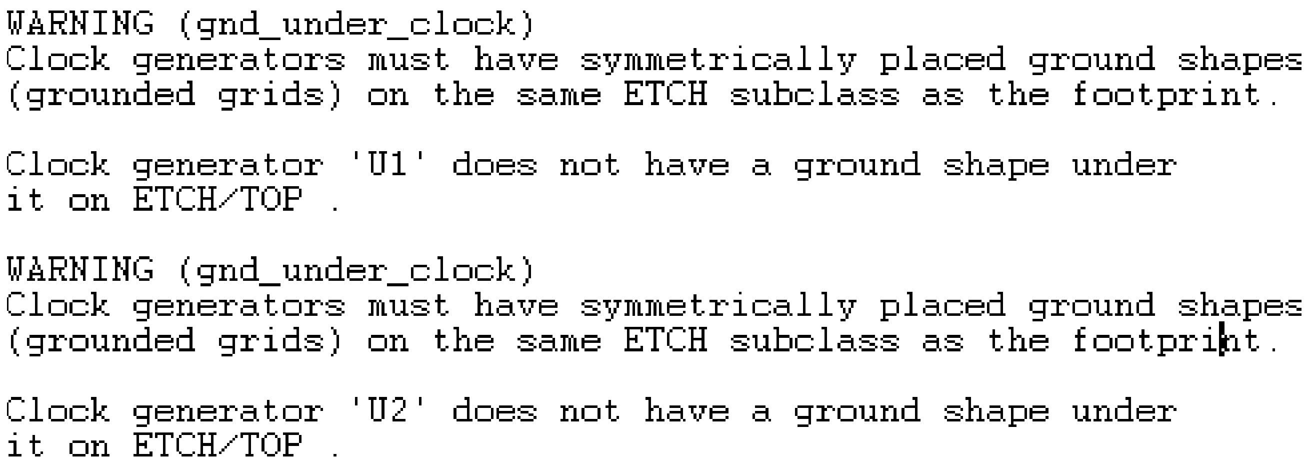
Rule-Specific Information

The body of the Execute report describes the short message (if one exists) and the advisor message for each violation. These messages list

- The severity level of the violation
- The name of the rule

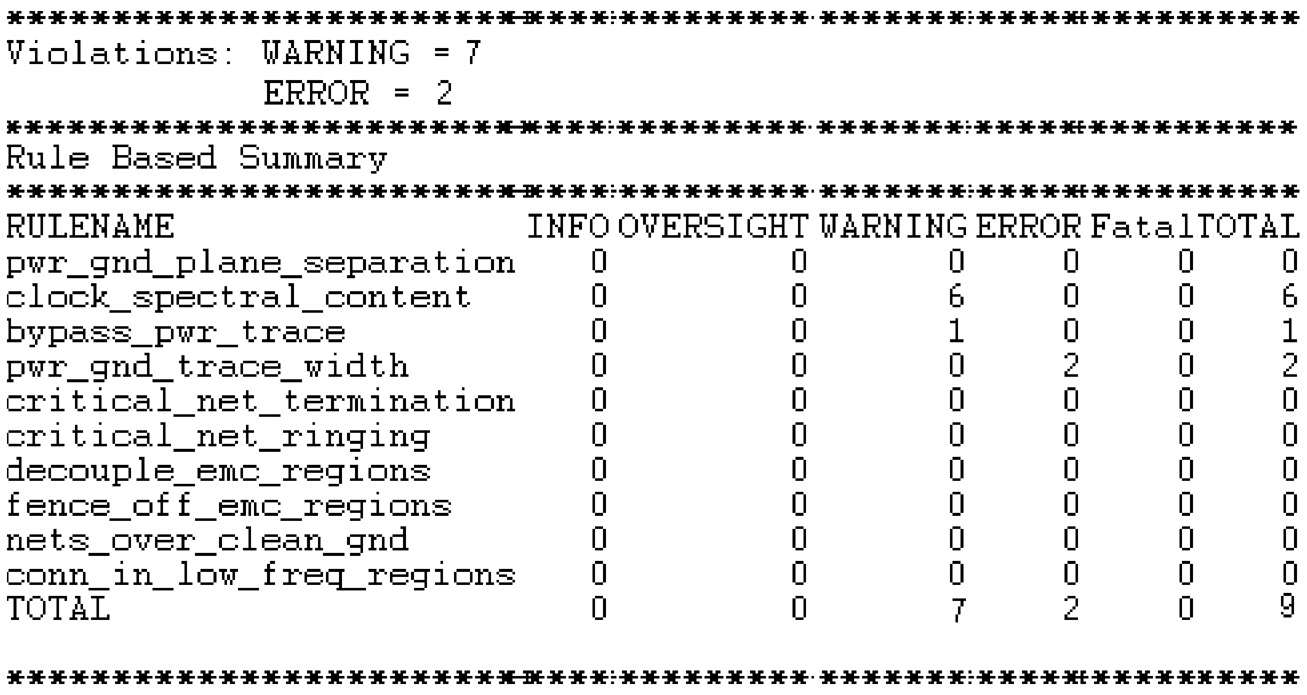
- Information about the violation

The following is an example of rule-specific information. The rule is `gnd_under_clock`:



```
WARNING (gnd_under_clock)
Clock generators must have symmetrically placed ground shapes
(grounded grids) on the same ETCH subclass as the footprint.

Clock generator 'U1' does not have a ground shape under
it on ETCH/TOP .

WARNING (gnd_under_clock)
Clock generators must have symmetrically placed ground shapes
(grounded grids) on the same ETCH subclass as the footprint.

Clock generator 'U2' does not have a ground shape under
it on ETCH/TOP .
```

The bottom of the report provides a summary of the number of violations recorded, by severity, for each EMC rule



```
*************************************************************
Violations: WARNING = 7
            ERROR = 2
*************************************************************
Rule Based Summary
*************************************************************
RULENAME                 INFO OVERSIGHT WARNING ERROR Fatal TOTAL
pwr_gnd_plane_separation    0         0       0     0     0     0
clock_spectral_content      0         0       6     0     0     6
bypass_pwr_trace            0         0       1     0     0     1
pwr_gnd_trace_width         0         0       0     2     0     2
critical_net_termination    0         0       0     0     0     0
critical_net_ringing        0         0       0     0     0     0
decouple_emc_regions        0         0       0     0     0     0
fence_off_emc_regions       0         0       0     0     0     0
nets_over_clean_gnd         0         0       0     0     0     0
conn_in_low_freq_regions    0         0       0     0     0     0
TOTAL                       0         0       7     2     0     9

*************************************************************
```

5

Writing Rules

Overview

This chapter explains what you need to know to write and compile custom rules for use during EMControl rule checking. It also provides sample rules.

EMControl uses rules that have been written using the Cadence Advanced Rule Language (ARL). The Cadence-supplied rule files are:

- `emc_placement.arl`
- `emc_bypass.arl`
- `emc_pwr_gnd_dist.arl`
- `emc_dc_route.arl`
- `emc_sig_route.arl`
- `emc_sig_qual.arl`

The compiled files are

- `emc_placement.rle`
- `emc_bypass.rle`
- `emc_pwr_gnd_dist.rle`
- `emc_dc_route.rle`
- `emc_sig_route.rle`
- `emc_sig_qual.rle`

Developing Rules in EMC

The EMC Rule Developer provides the graphical user interface (GUI) for customizing rules, predicates, and parameters. The Rule Developer lets you

- Navigate your directory structure to locate rule files
- Edit rule files
- Compile rule files
- Edit predicates
- Add parameters
- Export parameters

Cadence supplies a set of rules, but you can write your own rules also. For this, you may use the standard predicates and parameters or define new ones, as per requirement. Parameters are like variables, which have certain default values but you can edit them also. Predicates are analogous to functions in other programming languages.

Once you have written a rule and compiled it, you might want to make it accessible to the users in your site. For this, you need to make the parameters, the predicates used in the rule and the complied rule file (`.rle`) visible.

To make the rule file visible

➤ The`.rle` file should be available in the rule paths. The path can be given in the *Rule File*(*s*) section of the *EMC Rule Selection* dialog box.

To make the predicates visible

➤ The mapping file (`.env`) containing the predicates used in the rule must be available in the Mapping Files' paths. The path can be given in the *Mapping File*(*s*) section of the *EMC Initialization* dialog box.

To make the parameters visible

➤ Export the parameters used in the rule from the custom parameter file (`emc_custom.par`) to the site parameter file (/`site`/`emc_param.par`)

Predicates

Adding a New Predicate

To add a predicate from the EMC Skill Predicate dialog box

1. Select the environment (`.arl`) file where predicates have to be added.

   The *Predicates Defined* tree control displays the predicates defined in the file.

2. In the *Predicate Definition* section, submit the details of the predicate to be added. Give the name of the predicate, its underlying Skill function, the arguments' types and its return type.

   Note: The underlying Skill function needs to be loaded in Allegro before the corresponding predicate is defined.

3. Click *Apply*.

   The specified environment file gets updated with the changes.

4. Click *OK*.

   The changes are saved and the dialog box is closed. The environment (`.arl`) file is compiled to produce the mapping file (`.env`).

Note: You need to have write permissions for adding a predicate in an environment file.

Deleting a Predicate

To delete a predicate from the EMC Skill Predicate dialog box

1. Select the environment file where predicates have to be deleted.

   The *Predicates Defined* tree control displays the predicates defined in the file.

2. In the *Predicates Defined* section, select the predicate to delete.

3. Click *Delete*.

4. Click *Apply*.

   The specified environment file gets updated with the changes.

5. Click *OK*.

   The changes are saved and the dialog box is closed.

Note: You need to have write permissions for deleting any predicates in an environment file.

Editing a Predicate

To edit a predicate from the EMC Rule Developer dialog box

1. Click *Edit Predicate...*

   The *EMC Skill Predicate* dialog box appears.

2. Select the environment (`.arl`) file where predicates have to be edited.

   The *Predicates Defined* tree control displays the predicates defined in the file.

3. Select the predicate to be edited.

   The details of this predicate appear in the *Predicate Definition* section

4. Edit the details of the predicate, the name of the predicate, its underlying Skill function, the arguments' types and its return type.

   Note: The underlying Skill function needs to be loaded in Allegro before the corresponding predicate is defined.

5. Click *Apply*.

   The specified environment file gets updated with the changes.

6. Click *OK*.

   The changes are saved and the dialog box is closed. The environment (`.arl)` file is compiled to produce the mapping file (`.env`).

Note: You need to have write permissions for editing a predicates in an environment file.

Parameters

Adding a Parameter

To add a parameter from the EMC Rule Developer dialog box

1. Select the rule file or the rule in which you wish to add a parameter.
2. Click *Add*.

A new row appears in the grid control. You have to fill the name of the new parameter, its type, its default value, and its scope whether it is global or local.

3. Enter the name of the parameter in the column *Parameter Name.*
4. Enter the type of the parameter in the column *Type*

   The type of the parameter can be `String`, `String List`, `Numeric`, or `Numeric List`.
5. Enter the default value of the parameter in the column *Default Value.*

   Note: If the parameter is being added to a specific rule, the scope is local, while if the parameter is being added to a rule file, the scope is global.
6. Click *OK*.

   The changes are saved in the `emc_custom.par` file and the dialog box is closed.

Deleting a Parameter

To delete a parameter from the EMC Rule Developer dialog box

1. Select the rule file or the rule from which you wish to delete a parameter.
2. Select the parameter row that has to be deleted.
3. Click *Delete*.

   Note: Only the parameters that have been added by the user can be deleted. If you attempt deleting a Cadence-supplied parameter, a message saying `This Parameter cannot be deleted` appears.
4. Click *OK*.

   The changes are saved in the `emc_custom.par` file and the dialog box is closed.

Exporting a Parameter

To export a parameter from the EMC Rule Developer dialog box

1. Select the parameter row that has to be exported.
2. Click *Export*.

   The *EMC Param Export* dialog box appears.
3. Enter the location of the site parameter file in the *Site Parameter File* box.

4. Click *OK*.

   The parameter is added in the site parameter file and the dialog box is closed.

Rules

Editing a Rule File

To edit a rule file from the EMC Rule Developer dialog box

1. Specify the file name in the *ARL Source File*.
2. Click *Edit Rule* in the EMC Rule Developer dialog box.

   The file that you specified is opened in WordPad, with the text editor specified by the EDITOR environment variable. By default, the editor is WordPad. If a new rule is being written, a new file opens.
3. When you have finished working in the text editor, save the rule file and quit the editor.

Compiling a Rule File

After you have edited and saved your rule file, you can compile the file from the EMC Rule Developer dialog box.

To compile a rule file

1. Select the file to compile in the EMC Rule Developer dialog box.
2. Click *Compile*.

   EMControl compiles the file and creates the compiled file in the specified compiled file directory. The file name extension of the compiled rule file is`.rle`.

If the file compiles correctly without any errors, the message `0 Messages` is displayed in the Allegro console window. In case of any errors or warnings, the `EMC Compilation Log` window comes up showing the errors and warnings encountered during the compilation process.

Writing a Rule

The sections Sample unconnected_critIC_pins Rule on page 87 and Sample check_via_count Rule on page 88 show sample rule files written using the Advanced Rule

Language (ARL). For more information on the ARL, refer Appendix A of the EMC User Guide. The sample rules contain explanatory comments, which are set off according to either of two ARL conventions:

- Rule unconnected_critIC_pins (“Sample unconnected_critIC_pins Rule” on page 87) uses the convention in which comments are set apart from the code as follows:

  `/* <comment> */`

- Rule check_via_count (“Sample check_via_count Rule” on page 88) uses the convention in which comments are set apart from the code as follows:

  `// <comment>`

The sample rules call various EMControl predicates to perform required functions (for example, `hasProperty` and `getPropertyValue`).

Remember the following points when writing your own rules:

- It is recommended that you copy and edit one of the Cadence rule files when you are writing your own rules.
- An ASCII rule file name must have the extension `.arl`.

  The compiler converts a file with the `.arl` extension to a `.rle` file.

- Be sure to use an `include` statement to specify any files that the rule needs to access at run time.

  The sample rules use variables that are defined in the `emc_param.par` file.

- You need to use an index to derive the appropriate value from any parameterized variable that you are using in your rule.

  Parameterized variables define values for all the critical IC, critical net, and critical region classes that are specified by EMC_COMP_CLASSES or EMC_NET_CLASSES, respectively.

- Be sure to create a help file that the user can open from the EMC Rule Selection dialog box to get information on using each rule that you write (see “Creating a Help File for a Rule” on page 90).
- EMControl uses a second rule set to verify property requirements for each selected rule when the user selects Audit. A verify version of each Cadence-provided rule exists in the rule files:
  - ❑ `emc_placement_verify.arl`
  - ❑ `emc_bypass_verify.arl`
  - ❑ `emc_pwr_gnd_dist_verify.arl`

- `emc_dc_route_verify.arl`
- `emc_sig_route_verify.arl`
- `emc_sig_qual_verify.arl`

■ Create a verify rule for each new rule that you write so that *Audit* will include your rules in property verification. Note the following:

- It is recommended that you copy and edit the Cadence verify rule files when you are writing your own rules for property verification.
- The name of the verify version of a rule must correspond to the name of the new rule, and must be in a rule file with a corresponding name prefix (see the next item).
- After compiling the verify rule file, move the compiled version of the file to a new name with the following format:

`<`*`compiled EMC rule file name without extension`*`>.rle.verify`

As an example, the Cadence rule file for placement is `emc_placement.rle`, while the verify rule file is `emc_placement.rle.verify`.

- Using the `.rle.verify` extension prevents the verify file from being loaded in the rule browser in the EMC Rule Selection dialog box. The user running rule checking does not need to be concerned with verify rule files.
- There must be a one-on-one correlation between each rule file specified in the EMC Rule Selection dialog box and a verify rule file.

  An EMC rule file and its corresponding verify rule file must exist in the same directory, because EMC looks for the verify file at the location specified for the rule file in the EMC Rule Selection dialog box.

Sample unconnected_critIC_pins Rule

```
include "emc_custom.par"
include "site/emc_param.par"
include "emc_param.par"

use EMCONTROL;

RuleDefine
      Rule unconnected_critIC_pins

hasProperty( component1, "USER_COMP_PROP") AND
                /* Pick up components with USER_COMP_PROP , which is
                      defined here above as EMC_CRITICAL_IC */
 pin1 := getPin( component1 )  AND                                       /*Store
list of pins of component1 in variable pin1  */
    val1 := count( pin1 )  AND                                     /*  Count all
pins of component1 and
store in val1  */
 val3 := count( foreach( pin1,                                    /*   Count all
pins of component1 that are connected... */
       val2 := getAllegroPinUse( pin1 )  AND                        /*     ...to
some net; i.e., do
not have a Pinuse value of... */
        val2 == { "UNSPEC", "NC"} /*index(concat( "UNSPEC", "NC") ,val2)
..."UNSPEC" or "NC". Store in val3 */
        ) ) AND
    val4 := ( val3 * 100)/val1   AND                                     /*
Calculate the percentage
of unconnected pins and store in val4 */
    val5 := UNCONNECTED_PIN_PERCENT   AND                          /*  Store the
value of UNCONNECTED_PIN_PERCENT in variable val5 */
    val4 >  val5                              /*    Check if val4 exceeds val5 */^M

/* If all above conditions are met then report this message: */
  Message( INFO,                                /* Severity of the message    */
        component1,                    /* Object to be highlighted  */

"Check Unconnected CritIC Pins",                    /* Short message displayed in
                                 the messages browser  */
        "Percentage of Unconnected Pins of a Critical IC should not exceed the
\nuser-defined value i.e., ?val5. \nFor ?component1, UNSPEC and NC pins are : ?val3
while total pins are ?val1. \nResulting percentage = ?val4 \n\n" );
/* Preceding is long message description, displayed when View is selected
 for a violation in the messages browser. */


     EndRule unconnected_critIC_pins

EndRuleDefine
```

Sample check_via_count Rule

```
include "emc_custom.par"
include "site/emc_param.par"
include "emc_param.par"

 use EMCONTROL;

 RuleDefine

      Rule check_via_count

// Get the value of the property EMC_CRITICAL_NET.
     // The property value contains the net class; for example, CLASS1.
     // Ignore the case of the class string.
     // Store the property value in a variable (say, valClass).

     valClass := upperCase(getPropertyValue( net1, "EMC_CRITICAL_NET")) AND

     // Get all the allowed classes as defined
     // by the parameter EMC_NET_CLASSES.

     // Ignore the case of the class strings.
     // Store these classes in a variable (say, valPosClass).

     valPosClass := concat(EMC_NET_CLASSES) AND

     // Look for the net-specific class (valClass) in
     // the class list (valPosClass).
     // If found, store the position of the net class (valClass)
     // in the class list (valPosClass) in a
     // variable (say, valIndex); for example, valIndex = 1.

     valIndex := index(valPosClass, valClass) AND

// Read the value of the parameterized variable EMC_VIA_COUNT.
     // Separate out the values of this parameter for various classes.
     // Convert these string values into integer  values.
     // Store this value list in a variable (say,  valViaCntAll).


     valViaCntAll := concat(EMC_VIA_COUNT) AND

     // Get the value that corresponds to the net class.
     // Pick the nth value from the value list (valViaCntAll),
     // where n is equal to valIndex.
     // This value represents the value of EMC_VIA_COUNT
     // for the net class (valClass).

     valViaCnt := nth(valViaCntAll, valIndex) AND

     // Get all the vias on the net under consideration.
     // Count the number of vias and store it in a variable (say, val1)

     val1 := count( getViasOnNet(net1)) AND

     // Compare the value against the allowed  via count(valViaCnt).

   val1 > valViaCnt
```

```
     // In case all the above conditions are true, report this violation.

    Message(ERROR, net1, "Check the number of vias  per  critical  net.", "\n The
number of vias per critical net  should be kept at a \n minimum in order to improve
reliability.  The maximum number of  \nvias allowed =  ?valViaCnt, vias found on
net ?net1 = ?val1  \n");

      EndRule

 EndRuleDefine
```

Setting Violation Severity Levels of Rules

The severity level is assigned to a rule in its Message definition in the rule file. The `emc_allegro`.env file contains the severity level values that you can currently use for rules. Options are `INFO`, `OVERSIGHT`, `WARNING`, `ERROR` and `FATAL` (in recommended increasing order of significance).

The Message definition is located within the RuleDefine section for the corresponding rule. A sample of a message severity definition for a rule is

```
Message (ERROR, ...
```

To specify objects to associate with a violation

You need to specify the list of relevant ARL variables in the Message definition for the rule. Specifying these variables allows the objects that they represent to be highlighted when a violation is recorded for the rule.

A sample Message definition is

```
Message (ERROR, component1 component2, "Check bypassing of high current, high speed
ICs.\n", "All Critical ICs need at least three bypass capacitors");
```

In the above example, component1 and component2 make up the list of variables. When the rule records a violation, the objects represented by these variables will be available for display once you click *Highlight List* in the Markers dialog box. When the violation is highlighted in the messages browser, you can click *Highlight List* to change the associated object that is currently highlighted in the design layout.

Creating a Help File for a Rule

Cadence recommends that you create a help file for each rule that you write. Include in the help file critical information that describes the function of the rule. You will be able to view this information through the EMC Rule Selection dialog box and use it to

- Determine whether a particular rule is relevant to the current design-checking task.
- Learn what properties and other variables are required by the rule.

You may need to attach relevant properties to objects in the design or modify default variable values.

To create a help file

1. Use a text editor to create the help file.

   A help file must have the name `<`*`rule_name`*`>.hlp`, where *`rule_name`* is the name of the rule (or rule file) for which this file provides help. The required extension is`.hlp`.

2. To make the file available for display in EMControl, place it in an appropriate directory location.

   You can do either of the following:

   Set the environment variable `EMC_HELP_PATH` to the path location of the help file. For example:

   ```
   setenv EMC_HELP_PATH /usr1/abc/pcb/emc/help
   ```

   –or–

   Place the file in the following default directory for help files:

   ```
   <your_install_dir>/share/pcb/signal/emc/help
   ```

   EMControl looks for help files in this directory when `EMC_HELP_PATH` is not set.

Running New Rules

Once you have written and compiled a new rule file, make the rules available for selection by the user in the EMC Rule Selection dialog box.

To add rule files to the rule browser in the EMC Rule Selection dialog box

1. Modify the rule path if the compiled`.rle` files are at a location other than the specified search path.

2. Identify the path location of parameter files that are specified in the rule file by means of an include statement.

   You do not need to identify the path if the parameter files are in the current directory.

3. Select the rules to run during EMControl checking.

A

ARL Training Guide

Introduction

EMControl is a CheckPlus based tool. It consists of the CheckPlus core engine ARC (Advanced Rule Checker), the EMControl rule set and the environment which defines the interaction between ARC and Allegro.

ARC is an execution engine which executes rules written in ARL (Advanced Rule Language). In order to utilize the maximum potential of ARL-based tools like EMC, it may be necessary to write customized rules that are focussed on a customer's unique design environment. With the recent enhancements to the core ARL engine, many more customer rule requests are now possible. This training guide will start with the basics of the ARL language and then move into the intricacies of more complex rule-writing.

Language Highlights

ARL is a language suitable for expressing design rules. The major features of the language which distinguish it from other general purpose languages are highlighted below.

- Simple Condition - Action semantics.
- Implicit looping through design objects
- Ability to accept parameters at run-time. This allows the user to customize the rule according to his/her design needs
- Ability to specify the objects to be highlighted by the markers utility.
- Automatic iteration over function (predicate) arguments.

EMControl Objects

ARL is currently used by many different tools at Cadence. The CheckPlus core engine, which executes rules written in ARL, do not have any internal information about the design on which it is checking the rules. (e.g. The CheckPlus core engine does not have any knowledge of Allegro elements).

CheckPlus understands about the target design (e.g. Allegro board) from the environment file provided. The environment file defines the objects in the design that can be manipulated by ARL. It also defines a set of functions (called predicates) which is used to interface with Allegro.

The EMControl environment contains 8 different objects:

Design	The board design upon which EMC is run.
Component	Components in the design.
Net	Nets in the design.
Pin	Pin elements in the design.
Via	Via elements in the design.
Shape	Shape, rectangle and filled-rectangle elements in the design.
Polygon	Polygons in the layout X_Y plane.
Drc	DRCs in the design.

Any element in the design which belongs to the above types can be directly referenced in a rule.

EMControl Predicates

Predicates facilitates the interaction between ARC and Allegro. They are analogous to functions in other general purpose languages. A predicate takes one or more arguments and returns a list as result.

e.g.

component

```
component(design)
```

The predicate component() takes a design object and returns all the components in the design.

pin

```
getPin(component)
```

The predicate getPin() takes a component object and returns the list of pins of the component.

value

```
name (net)
```

The predicate name() takes a net object and returns a string representing the net name.

Predicates internally call C or SKILL functions which compute the desired result by accessing the design database. The complete list of predicates for EMControl is given in the user guide.

Getting a Feel for the Language

In order to get an introduction to ARL we will dissect a few simple rules that demonstrate the fundamentals.

Example 1

Consider a simple case. Suppose you want to list all the nets in a design individually. In pseudo-code we could write it like:

```
for each net in the design

print the name of the net

endfor
```

In ARL it is coded as:

```
RuleDefine                          /* start of rule declaration */
      Rule print_net                /* start of rule <rulename> */
           net1                     /* condition statement */
           Message(Info,net1,"?net1");/* message statement */
      EndRule                       /* end of rule */
EndRuleDefine                       /* end of rule declaration */
```

Basic Language Constructs

There are various parts to a rule. The keywords RuleDefine and EndRuleDefine indicate the start and end of the rule declaration area of a text file. The keywords Rule and EndRule are part of the rule declaration syntax that specify the logical breakup of separate rules. Every rule must be enclosed by its own Rule and EndRule set of keywords. This differs from the RuleDefine and EndRuleDefine which are only required once per text file, but allowed as often as once per rule. A rule name can consist of alphanumeric characters, the underscore and the $sign. The first character cannot be a number.

Note: The language is case insensitive, so `ruledefine` and `RuleDefine` are equivalent. The general syntax that you will see followed here will mix upper and lower case in an attempt to clarify the intended meaning of the Function.

Inside the rule declaration constructs exist the remaining 2 sections of the rule. These sections are the rule conditional statements and the message statement. As you may predict, the conditional statements determine whether the rule reports the information that is supplied in the Message statement. In the above example the conditional section consists of 1 line, net1. As you will see later in this section, the conditional section is normally a logical grouping of statements that get evaluated to `True` or `Nil`(`False`) according to the predicates called and the data the predicates are supplied.

Variables and Base Objects

Every CheckPlus rule contains exactly 1 Base Object. A base object of a rule is a variable which cannot derive its value from any other variable in the conditional section of the rule.

Example 1:

```
RuleDefine

      Rule base_obj
           component1 := component(design1) AND
                           /*component1 is derived from design1 */
           net1 := net(design1) /* net1 is derived from design1 */
           Message(Info,"?design1");
      EndRule
EndRuleDefine
```

In the above example design1 cannot derive it’s value from any other variable. Therefore the base object of the rule is design1.

Example 2:

```
RuleDefine

      Rule base_obj2

           component1 := getCompsConnToNet(net1) AND
                               /* component1 is derived from net1 */
           component2 := hasProperty(component1, "EMC_COMP_TYPE")
                               /* inst2 is derived from inst1 */
           Message(Info,component2 ,"Components having prop EMC_COMP_TYPE are
?component2");
      EndRule
EndRuleDefine
```

In example 2 the base object is net1 as `component1` is derived from it and `component2` is derived from `component1`.

The base object also determines the number of times the rule is executed on the existing design. The EMC environment contains seven different base objects. The Net object is one from this group. Since we have selected Net as our base object, this rule will be executed by the CheckPlus engine 1 time per net in the design. Therefore the maximum number of error messages generated by the rule is equal to the total number of nets in the design. For example, if a design has 10 nets then this rule may result in 0 to 10 messages.

Base Objects and Implied Looping

Base Objects constitute 1 main area of implied looping. In this rule we have not explicitly told the engine to iterate over all nets. But by selecting Net as our base object the engine understands that it must evaluate the rule for every net in the design. In order to understand the implied looping execution, think of the engine working like the following pseudo-code:

```
net1 := First net in the design
While (!end of net list) do
{
      Execute the rule
      print message if condition evaluates "True"
      net1 := Next net in design
}
```

As you can see, the number of messages printed can be anywhere between 0 and the number of nets in the design.

Later in the design we will investigate the rationale behind selecting a base object. In the meantime here is another example to reinforce the idea of Base Object looping

Example:

```
List all the nets of the design in one message.
foreach design
      print all nets in design
endfor
RuleDefine
      Rule print_net11
           net1 := net(design1)
           Message(Info,net1,"?net1");
      EndRule
EndRuleDefine
```

In this example the base object is Design. Within an execution in the EMC environment there is only 1 item that can be assigned to the Design object. This item is the board that was requested as the target of the EMC run. Since there is only 1 value for the base object, this rule will only be executed 1 time.

Predicate Calls

A predicate call in the CheckPlus language is similar to a function call in other languages. It takes 1or more arguments and returns a value. Before writing or understanding rules in a specific environment it is necessary to have a list of predefined predicates available for that environment.

Note: Currently EMControl does not allow users to create their own predicates. You must utilize the predicates that are supplied with EMControl (The complete list is given in the user guide).

The rule in example 2 calls the function, Net(), and passes it the base object variable, Design. By looking up the definition of the Net() function you find out that it accepts a parameter of type Design, and returns a list of nets in that design. Therefore after the execution of the Net() predicate, the variable, net1, will consist of a list of nets in the design.

Variable Typing

This leads us into the discussion of variable types and how variables are assigned and compared. The CheckPlus Variables are used to store values that you compute. There are two types of variables -

- Object Variables: Holds an object (objects are defined in the EMC environment)
- Non Object Variables: Assigned numerical values, strings etc.

Object variables are identified by their names. Object variable names must start with the object name that they will be assigned. So in Example 1 the return value of the predicate, Net(), is a net object, therefore the variable name must begin with "net". Any string may be appended to the object type to create a variable name. Some valid Net variables follow:

“net1", "net2", "net_VCCl", "net_ground", "net_all" etc.

The rule compiler will report errors in the compilation of the rule if this variable naming requirement is not met.

ARL Operators

ARL provides a rich set of arithmetic and logical operators. Logical operators are used to join multiple boolean results to create complex logical expressions. Examples of these operators are AND, OR, and XOR. From example 3 you can predict, this rule will only report information to the user if the total number of signals in the design equals 3. Each of the conditional statements must evaluate to a TRUE, or non-null value since they were joined together with the AND operator. If not the Message construct will not be executed.

The following set of operators can be used in expressions. These operators are grouped according to precedence, from highest to lowest.

Operator	**Description**
isNull, abs,exp	negate, isInputArgumentNull, absolute, exponent
*, /, mod, rem	multiply, divide, modulus, remainder
+, -	add, subtract
and, or, xor	logical operators
{item, item, ...}	one of a list of items
==, /=, <. <=, >, >=	equal to, not equal to, less than, less than or equal to, greater than, greater than or equal to

Exercises

Exercise 1

Consider a design with 10 nets and 2 components. How many messages will the following rules generate?

1.

```
RuleDefine
     Rule print_net
          net1
          Message(Info, "?net1");
     EndRule
EndRuleDefine
```

2.

```
RuleDefine
     Rule print_component
          component1
          Message(Info, "?component1");
     EndRule
EndRuleDefine
```

3.

```
RuleDefine
     Rule print_all
          net1 := net(design1) AND
          component1 := component(design1)
          Message(Info,"?inst1 ?sig1");
     EndRule
EndRuleDefine
```

Exercise 2

Consider a design which has 10 components and 0 nets. How many messages will the above rules generate now?

Exercise 3

What is the basic object in the following rule?

1.

```
Rule ex1
     component1 := getCompsConnToNet(net1) and
     net2 := getNetOf(getPin(component1)) and
     val1 := count(net2)
     Message(Error, "?net2 ?val1");
EndRule
```

Exercise 4

Write a rule to list all the components in the design individually. The number of messages should equal the number of the components in the design.

Exercise 5

Write a rule to list all the components in the design in one message.

Exercise 6

Write a rule to print a count of all components and nets in the design

List Manipulation

The CheckPlus rule language provides lists as its only form of data structure. Every variable type in the language can be used to create a list of 1 or more elements. Up to this point in the training guide we have demonstrated only basic list creation techniques using the implied looping over a list of base objects by the CheckPlus engine. The language also supports other basic forms of list creation. This section will investigate these language characteristics and the operations supplied for manipulating lists.

What are Lists

A CheckPlus list is a collection of one of the following:

- Object Elements
- Non Object Elements

Lists are homogeneous lists i.e. they can contain only one type of object. For example, you cannot have a list that consists of components and nets. Nor can you have a list that contains strings and integers.

Example 1:

```
RuleDefine
      Rule list_def
           component1 := component(design1) AND
           val_type := getPropertyValue(component1, "EMC_COMP_TYPE"))
           Message(Info,"Components ?component1 , comptypes : ?val_type");
      EndRule
EndRuleDefine
```

In the above example `component1` is a list of components (object elements) and val_type is a list of non-object elements (strings in this case). Note that `getPropertyValue`() is passed a list of components. The predicate is called for each component in the list `component1` and the results are put in another list which gets assigned to `val_type`.

The lists are created by predicates. In the previous section we discussed how lists of Base Objects were created by the Engine for use in evaluating each selected rule. This is 1 type of implied list manipulation that does not require any special understanding by the programmer. It should be understood that the length of the Base Object list determines the number of times the rule is executed on the design database. This base object list can be displayed by using the Message() command on every item in the list. In the following example net1 is assigned each element of the base object list, one at a time, for each execution of the rule.

Example 1:

```
RuleDefine
      Rule print_net
           net1
           Message(Info,net1,"Name : ?net1");
      EndRule
EndRuleDefine
```

List Manipulation Routines

The difference between a list routine and other predicates is that while the list routines operate on the full list, other predicates operate on one element of the list at a time.

Example 1:

```
RuleDefine
      Rule ListEg4
           shapes := shape(design1) AND
           shapes_rect := isRectangle(shapes) AND
           shapes_no_rect := remove(shapes, shapes_rect) AND
           val := count(shapes_no_rect)
           Message(INFO, shapes_no_rect, "No. of non-rectangle
                                shapes = ?val");
      EndRule
EndRuleDefine
```

The intent of this rule is to use count() to compute the count of all non-rectangle shapes in the design.

In the above rule `isRectangle`() will be called implicitly once per element in the list `shapes`. But remove() and count() being list routines will be called only once.

For example consider a design with 10 shapes. In this case `isRectangle`() will be executed 10 times (once for each element in the list `shapes`) while remove and count will be executed only once in this line.

(Note: There is an interesting bug in the above rule! If the design does not have any rectangle elements, the rule will not report non-rectangle shapes. In this case, the rule does not give a violation)

Here are some of the predicates which operate on lists. Each of the following predicates except append returns the result. The input list is not modified. So in the following discussion saying remove(list1, list2) removes the elements in list2 from list1. This means that the remove predicate returns list3, which contains all elements of list1 which are not contained in list2. As an exception, append(list1, list2) attaches list2 to list1. So it modifies its input argument.Here is a table of the basic ARL list manipulation routines:

Function	**Description**
car(list)	Returns the first element
cdr(list)	Returns the list without the first element
nth(list, i)	Returns the ith element
remove_i(list, i)	Returns the list without the ith element
min(list)	Returns the smallest element of the list of non objects
max(list)	Returns the largest element of the list of non-objects
sum(list)	Returns the sum of the list of integers/floats
last(list)	Returns the last element of the list

Function	Description
index(list, element)	Returns the index into the list if element exists
count(list)	Returns the length of the list
concat(list1, list2)	Returns a new list that is the concatenation of list1 and list2
union(list1, list2)	Returns a unique list that is the union of list1 and list2
intersection(list1, list2)	Returns a unique list that is the intersection of list1 and list2
remove(list1, list2)	Returns list1 such that it contains no elements of list2
unique(list1)	Returns a list that contains no duplicate items
sort(list1)	Returns a sorted list
concat_str(list1)	Returns a string value that is formed by concatenating the strings from list1
append(list1, list2)	Modifies list1 by appending the values from list2

Here are some examples of how these list manipulation functions work:

Assume a list of Vowels like the following.

Vowels = (a e i o u)

Note that here the parentheses are not recognized as a syntax to specify lists. They are used for clarification only.

What will be the output of the following?

❑ car(Vowels)	(a)
❑ cdr(Vowels)	(e i o u)
❑ nth(Vowels 3)	(i)
❑ car(cdr(Vowels))	(e)
❑ cdr(car(Vowels))	nil
❑ index(Vowels, "a")	(1)
❑ index(Vowels, "p")	nil
❑ count(Vowels)	(5)

> **Note:** A single element return value is these examples actually constitutes a list containing one element.

Foreach Construct

The Foreach construct provides a mechanism for iterating over all members in a list. A foreach construct is generally used to extract a subset of a list. Here is the syntax of the foreach construct:

```
foreach(list, condition)
```

Note: The return value of any Foreach construct is always a new list. Since the Foreach construct is really just another conditional line in a rule, it evaluates to true if the list that is returned is non-null.

The first parameter of the Foreach construct is a list. This list is also used inside the condition statements as the variable. An example best explains this.

Example:

```
List all components that have more than two pins.
RuleDefine
      Rule comps_pins
           component1 := component(design1) AND
           component2 := foreach(component1,
                                 count(getPin(component1)) > 2
                            )
           Message(ERROR, "Components that have more than two pins are",
                                 "?component2");
EndRule
EndRuleDefine
```

In this example you can see that component1 is the list that is used in the Forech construct. It is also the variable name used inside the condition. component2 will be a subset of component1. It will contain those elements of component1 that satisfy the condition section of the foreach construct.

Saving Intermediate Results Within a Foreach construct

Often you would like to collect the intermediate results while doing computation inside a foreach construct. This is especially helpful for debugging rules as you develop them. The value of the variable used inside a foreach construct will change for each iteration of the loop. The append() predicate provides a convenient way to accumulate the results in one variable.

Suppose you want to find all cline segments over a shape that intersect with the shape boundary at the non-90 degree intersection angles of cline segments over a shape. The rule for this will be something like:

```
RuleDefine
Rule append_usage
           nets_clineseg := getClineSegsInArea(getPolygon(shape)) AND
           nets_clineseg_non90 :=
           foreach(nets_clineseg ,
valAngle := getIntersectAngle(shape, nets_clineseg) AND
valAngle /= 90 AND
           )
           Message(INFO, shape,"The none-90 degree cline segs",
                           "are","?nets_clineseg_non90");
EndRule
EndRuleDefine
```

Now if you want to find all the non-90 degree angles aswell, you can use append() in the above rule as illustarted below:

```
RuleDefine
Rule append_usage
           nets_clineseg := getClineSegsInArea(getPolygon(shape)) AND
           nets_clineseg_non90 :=
           foreach(nets_clineseg ,
valAngle := getIntersectAngle(shape, nets_clineseg) AND
valAngle /= 90 AND
                append(val_non90_angles , valAngle)
           )
           Message(INFO, shape,"The none-90 degree cline segs",
                           "are","?nets_clineseg_non90");
EndRule
EndRuleDefine
```

If we print `valAngle` in the message, we'll get the value for the last iteration. The append() predicate accumulates all non-90 degree angles in `val_non90_angles`. Append() differs from other predicates in some characteristics. They are -

- Append()creates its first argument. So it must be an identifier. Passing something else, e.g. a predicate call to append() will result in compilation error.
- As a corollary to 1, append can create a variable only once. Trying to use append twice on a variable will result in an error. The same variable cannot be used as the first argument of two different appends. Please note that iterating over the same append call is allowed using foreach, as in the above example. Actually that is what gives append the power!

If Construct

ARL provides an If conditional construct. Like the Foreach construct, If construct also returns a value which may be assigned to variables.

The syntax of the If construct is:

```
if(condition, expression1, expression 2)
```

If the condition evaluates to TRUE expression 1 is returned. Otherwise expression 2 is returned. The following example illustrates the use of If:

```
RuleDefine
      Rule If _prop
           valProp := if(hasProperty(component1, "EMC_COMP_TYPE"),
                      getPropertValue(component1, "EMC_COMP_TYPE"),
                      "NO_PROP"
                ) 
           Message(INFO, "Component ?component1 have comptype ?valProp);
      EndRule
EndRuleDefine
```

If the component has the `EMC_COMP_TYPE` property, valProp is assigned its value. Otherwise valProp gets the value NO_PROP.

expression1 and expression2 can contain assignments as well. In such cases the same set of variables should be bound in both expressions.

Exercise

Assuming component1 is a list of components and net1 is a list of nets in the design. Which of the following conditions are valid

1. component1 = concat(component2, net1)
2. val2 := concat("a", "b")
3. val3 := concat("a", 1)
4. component1 := component(design1) and

   net1 := net(design1) and

   val1 := name(component1) and

   val2 := name(component1) and

   val3 := concat(val1, val2)
5. component1 := net1
6. val1 := 2 and

   component1 := component(design1) and

   val1 == component1

Dissection of an existing rule

In this section we will discuss the implementation of some of the existing EMC rules. The source code of all the rules will be available in the customer installation. Understanding the existing rules will help the user to get a better insight into the ARL language and its power in specifying design rules. It will also help the user to make modifications in the rules so that they can be tailor-made to suit his/her EMC needs.

Rule critical_net_via_count

Let us first look at the simple routing rule critical_net_via_count. The rule checks that all critical nets have not more than EMC_VIA_COUNT vias on them.

EMC_VIA_COUNT is a parameter used in this rule which specifies the maximum number or vias allowed for each class.

```
#define EMC_VIA_COUNT "7 8 9 10 11"

RuleDefine
Rule critical_net_via_count

valClass := upperCase(getPropertyValue( net1, EMC_CRITICAL_NET)) AND
/* Get the net class into valClass. Only critical nets get selected */

valPosClass := upperCase(getAllSubstrings( EMC_NET_CLASSES, {""," "}, {" ", ""}))
AND
/* Get the list of all net classes in valPosClass */
valIndex := index(valPosClass, valClass) AND/* Get the index of the selected net’s
class in valPosClass */

valViaCntAll := atoi(getAllSubstrings( EMC_VIA_COUNT, {""," "}, {" ", ""})) AND
/* Get the list of all via counts specified in parameter EMC_VIA_COUNT */

valViaCnt := nth(valViaCntAll, valIndex) AND
/* Get the via count corresponding to the selected net’s class .
For example a CLASS3 net will have valViaCnt = 9*/

val1 := count( getViasOnNet( net1 ) )  AND
/* Get the via count for the net into val1 */

val1 >  valViaCnt
/* Compare if the via count val1 is greater than the maximum allowed number
valViaCnt */

Message( ERROR,
          net1,
          "Check the number of vias per net.","\nThe number of vias per net should
          be kept at a",
          "minimum in order to improve reliability. The maximum number of ",
          "vias allowed = ?valViaCnt, vias found on net ?net1 = ?val1 "
);

/* Report the violation. Highlight the net and report the number of vias found */
```

```
endRule
endRuleDefine
```

Rule conn_in_low_freq_regions

Now we will study the implementation of a more complex placement rule conn_in_low_freq_regions. The rule checks for 2 things.

- All connectors should be placed in the lowest frequency regions
- A lowest frequency region containing a connector should not have any higher frequency components placed in it.

```
RuleDefine
    Rule  conn_in_low_freq_regions
                     /* CHECK 1 */
     isEmcRegion(shape_emc_region)   AND
     /* Select emc regions only */

     valRegionClass := upperCase(getPropertyValue( shape_emc_region,
                    EMC_CRITICAL_REGION)) AND
     /* Get the EMC_CRITICAL_REGION property value for the selected region */

     valAllClasses := getAllSubstrings( EMC_COMP_CLASSES, {""," "}, {" ", ""})
     AND
     /* Get the list of all EMC region classes into valAllClasses */

     valLowestFreqClass :=  last(valAllClasses ) AND
     /* Get the lowest frequency class */

     valRegionClass /= valLowestFreqClass  AND
     /* The rule proceeds further only if the EMC region is not of lowest frequency
     */

     valLayer := if( getSubclass(shape_emc_region) == "BOTTOM_ROOM" ,
                          "BOTTOM", "TOP")  AND
     /* get the layer (TOP or BOTTOM) of the EMC region */
     components := getCompInArea(getPolygon(shape_emc_region)) AND
     /* get all the components in the area of the EMC region */

     components_region := foreach(components,
                     getLayer(components) == valLayer) AND
     /* Select only those components which are in the same layer as that of the
        EMC region */


     components_connector  := foreach(components_region,
                     isConnector(components_region))
     /* Find out all the connector components in the EMc region */

     Message(WARNING, shape_emc_region components_connector,
          "Checks for connectors in Low Frequency Region",
          "\nConnectors should be placed in the lowest frequency region.",
          "Connector(s)\n\t?components_connector",
          "lies in region ?shape_emc_region of frequeny class ?valRegionClass.",
          "This connector should be placed in a ?valLowestFreqClass region"
          );

/* Report violation. Highlight the EMC regions and offending connectors */
                         /* CHECK 2 */
     isEmcRegion(shape_emc_region)   AND
     /* Select emc regions only */

     valAllClasses := getAllSubstrings( EMC_COMP_CLASSES, {""," "}, {" ", ""}) AND
     /* Get the list of all EMC region classes into valAllClasses */

     valRegionClass := upperCase(getPropertyValue( shape_emc_region,
     EMC_CRITICAL_REGION)) AND
     /* Get the EMC_CRITICAL_REGION property value for the selected region */

     valRegionClass == last(valAllClasses) AND
     /* The rule proceeds further only if the EMC region of lowest frequency */

     valIndex := index(valAllClasses, valRegionClass) AND
     /* Get the index of the EMC region class in list valAllClasses */

     valLayer := if( getSubclass(shape_emc_region) == "BOTTOM_ROOM" ,
     "BOTTOM","TOP") AND
     /* get the layer (TOP or BOTTOM) of the EMC region */

     components := getCompInArea(getPolygon(shape_emc_region)) AND
     /* get all the components in the area of the EMC region */

     components_region := foreach(components,
                         getLayer(components) == valLayer) AND
     /* Select only those components which are in the same layer as that of the
        EMC region */

     components_connector  := foreach(components_region,
                         isConnector(components_region)) AND
     /* Find out all the connector components in the EMC region */

     components_high_freq := foreach(components_region,
          valCompClass := upperCase(getPropertyValue(components_region,
          EMC_CRITICAL_IC)) AND
          index(valAllClasses, upperCase(valCompClass))  < valIndex )

     /* Get a list of all higher frequency components in the EMC region. This is
        found out by comparing the indices of the component class and region class
     */


Message(WARNING, shape_emc_region components_high_freq,
          "Checks for Higher Frequency components in Low Frequency Region",
          "\nLowest frequency regions containing connectors should not contain",
          "any higher frequeny components.\n",
          "The lowest frequency region ?shape_emc_region  containing
           connector(s)",
          "\t?components_connector",
          "contain the following  higher frequency component(s):",
          "\t?components_high_freq"
);

/* Report violation. Highlights the lowest frequency EMC region and the higher
frequency components contained in it */
```

```
EndRule
EndRuleDefine
```

Laboratory Exercises

Basic Rules

Exercise 1

Write a rule to report all nets having EMC_CRITICAL_NET as CLASS1. The base object should be net.

Exercise 2

Keeping the same base objest, modify the rule in problem 1 so that it reports all nets having EMC_CRITICAL_IC as CLASS1 and connected to a connector.

Exercise 3

Write a rule to report the number of pins and number of vias on a net.

Exercise 4

Modify the rule in exercise 2 to list the connector names also.

Exercise 5

Write a rule which for each IC component lists all other IC components which are within a distance DIST and is connected to the component. (DIST is a parameter used in the rule).

Modifying existing EMControl rule (conn_in_low_freq_regions)

Some times the user may want to make minor changes in the rule logic to adapt it to suit his/her design guidelines. Here is an exercise which captures this.

Exercise 1

Modify the conn_in_low_freq_regions rule so that the rule reports a violation whenever a higher frequency component is found in a region. (eg: CLASS1 component in CLASS2 region should report an error)

Exercise 2

Modify the conn_in_low_freq_regions rule so that there is an exact match between component class and region class. For example the rule should give a violation if a non-CLASS1 component is found in a CLASS1 region.

Custom EMC rule writing

Exercise 1

Write a rule which checks for the capacitance per unit area between power and ground planes of a board. The rule should give a message if the capacitance exceeds the parameter LAYER_CAPACITANCE. Assume the dielectric constant of the dielectric material to be 4.5. The default value of LAYER_CAPACITANCE is 100 f Farad/square mil

B

EMControl Rules

Overview

This appendix describes the rules included with EMControl. These rules are grouped in the following categories:

- Placement Rules
- Bypass Rules
- Power and Ground Plane Rules
- DC Routing Rules
- Signal Routing Rules
- Signal Quality Rules

Rule descriptions in this appendix appear alphabetically grouped within the categories described above. Each rule description contains the following information:

- A brief description of the rule
- Properties used by the rule
- Variables used by the rule
- Information about data (such as properties and variables) required by the rule
- The default severity level of the rule

The EMC expert at your site can best determine when and where to use each rule to check your design.

The uncompiled, ASCII versions of the rules are contained in the following files:

- `emc_placement.arl`
- `emc_bypass.arl`

- `emc_pwr_gnd_dist.arl`
- `emc_dc_route.arl`
- `emc_sig_route.arl`
- `emc_sig_qual.arl`

The path locations for these files are provided in Chapter 2, "Setting Up the EMControl Environment." Information on editing these rule files and writing new rules is contained in Chapter 5, "Writing Rules."

Many rules in this appendix use variables. You can modify the default values associated with these variables by editing the `emc_param.par` file. For more information on variable defaults, parameterized variables, and the `emc_param.par` file, see Chapter 2, "Setting Up the EMControl Environment."

Many rules described in this appendix depend on certain properties being attached to objects in your design. For information about attaching properties to your design before running EMControl, see "EMControl Properties" on page 27.

Some of the Signal Routing and Signal Quality rules use data from DF/SigNoise. Before running these rules, you must properly configure EMControl and SigNoise to run together. See "Initializing EMControl" on page 24 and "Initializing SigNoise" on page 26.

Placement Rules

The following rules are included in the `emc_placement.rle` file:

- central_clock
- comp_not_conn_dist
- comp_to_conn_dist
- conn_in_low_freq_regions
- gnd_screw_between_clock_and_conn

central_clock

Checks whether a clock generator is located at the center of the region formed by the IC components that it drives.

The rule first identifies all clock nets. Clock nets are defined as nets connected to the clock generator at pins with PINUSE = OUT. It then identifies all IC components connected to the clock nets.

The sum of the distances from each individual IC component to the clock generator is calculated and compared with the sum of the distances from each IC component to all other IC components on the net and the clock generator. If the first sum is greater than the second sum, a violation is reported.

Properties Used

EMC_COMP_TYPE = CLOCK_GEN

PINUSE = OUT

Variables Used

None

Required Data

Attach the EMC_COMP_TYPE property set to CLOCK_GEN to all clock generators.

You do not have to attach the PINUSE property to the pins. EMControl reads the PINUSE property directly from the layout database. (PINUSE is a hidden property that is usually present.) You can, however, overwrite the automatic property setting by attaching PINUSE as you would any other property.

Default Severity

Info

comp_not_conn_dist

Checks for a minimum distance between an IO component (a connector) and the components not connected to it. This rule ensures that any component not connected to an offcard net is more than the COMP_CONN_MIN_DISTANCE from the IO component. It also ensures that components not connected to the IO component are more than the COMP_COMP_MIN_DISTANCE from any component connected to the IO component.

Properties Used

None

Variables Used

COMP_CONN_MIN_DISTANCE

COMP_COMP_MIN_DISTANCE

Required Data

You can edit the values for the EMControl variables before using this rule to check your design:

- COMP_CONN_MIN_DISTANCE specifies the permissible minimum distance between components that are not connected to the IO component and the IO component itself.
- COMP_COMP_MIN_DISTANCE specifies the minimum distance between components that are not connected to the IO component and components that are connected to it.

Default Severity

Warning

comp_to_conn_dist

Checks for a maximum distance between an IO component (a connector) and the components connected to it, excluding passive elements and discrete components.

This rule ensures that all components connected to an offcard net are within the distance specified by the EMC_COMP_CONN_DISTANCE of the IO component connected to the offcard net.

Properties Used

None

Variables Used

EMC_COMP_CONN_DISTANCE

Required Data

You can edit the EMC_COMP_CONN_DISTANCE variable to modify the value for the maximum distance between components and the IO component.

Default Severity

Warning

conn_in_low_freq_regions

Checks whether all connectors are placed in the lowest frequency region available on the board.

This rule also checks whether any higher frequency components (as identified by their EMC_CRITICAL_IC property value) are present in the lowest frequency regions that contain connectors.

Properties Used

EMC_CRITICAL_IC

EMC_CRITICAL_REGION

Variables Used

None

Required Data

Attach the EMC_CRITICAL_IC property to critical ICs.

Attach the EMC_CRITICAL_REGION property to critical frequency regions.

Default Severity

Error

gnd_screw_between_clock_and_conn

Checks whether a grounded screw exists between a clock generator and a connector.

The area searched for a ground screw is a rectangular area with a length equal to the distance from the center of the clock generator to the center of the connector and a width equal to the length multiplied by the GND_SCREW_SENS_DIST_RATIO.

Properties Used

EMC_COMP_TYPE = CLOCK_GEN

EMC_COMP_TYPE = GND_SCREW

Variables Used

GND_SCREW_SENS_DIST_RATIO

Required Data

Attach the EMC_COMP_TYPE property set to CLOCK_GEN to all clock generators.

Attach the EMC_COMP_TYPE property set to GND_SCREW to all ground screws.

If required, you can edit the value of the GND_SCREW_SENS_DIST_RATIO variable before using this rule to check your design. GND_SCREW_SENS_DIST_RATIO specifies the ratio between the length and width for the rectangular area being searched for a ground screw.

Default Severity

Error

Bypass Rules

The following rules are included in the `emc_bypass.rle` file:

- bypass_cap_type
- bypass_critical_IC
- bypass_drvr_rcvr_bidir
- bypass_fast_sw_trans
- critical_IC_3caps_C_2C_4C
- critical_IC_loop_area
- decouple_emc_regions
- fence_off_emc_regions

bypass_cap_type

Checks that all bypass capacitors are one of the types specified by ALL_BYPASS_CAP_TYPE.

Properties Used

EMC_CPOMP_TYPE = BYPASS_CAP

TOL

VALUE

Variables Used

ALL_BYPASS_CAP_TYPE

Required Data

Attach the EMC_COMP_TYPE property set to BYPASS_CAP to all bypass capacitors.

The TOL and VALUE properties identify the tolerance and capacitance values of the capacitor.

If required, you can edit the value of the ALL_BYPASS_CAP_TYPE variable before using this rule to check your design. ALL_BYPASS_CAP_TYPE specifies the allowed bypass capacitor types. Use the following syntax to specify a bypass capacitor type:

```
"<device type>:<cap value>:<tolerance>"
```

For example:

```
#define ALL_BYPASS_CAP_TYPE "CAP-1:82UF:5%" ,"CAP-2:0.01uf:5%"
, "CAP-3:0.01uf:3%" , "CAP-2:10uf:4%" "CAP-4:1.0uf:2%"
, "CAP-5:5uf:5%"
```

Default Severity

Warning

bypass_critical_IC

Checks bypassing of high-current, high-speed integrated circuits (ICs). It ensures that all critical ICs have connected to them at least the number of bypass capacitors specified by MIN_BYPASS_CAPS. The capacitors used must be one of the types specified by the variable CRITICAL_IC_BYP_CAP_TYPE.

Properties Used

EMC_CRITICAL_IC

EMC_COMP_TYPE

TOL

VALUE

Variables Used

BYPASS_CAP_SENS_DIST

CRITICAL_IC_BYPASS_CAP_TYPE

MIN_BYPASS_CAPS

Required Data

Attach the EMC_CRITICAL_IC property to all critical ICs.

Attach the EMC_COMP_TYPE property set to BYPASS_CAP to all bypass capacitors.

The TOL and VALUE properties identify the tolerance and capacitance values of the capacitor.

If required, you can edit the default values for EMControl variables before using this rule to check your design:

- BYP_CAP_SENS_DIST defines the bypass capacitor sensitive distance. (This variable is parameterized.)
- MIN_BYPASS_CAPS defines the minimum number of bypass capacitors that must be associated with a critical IC. (This variable is parameterized.)
- CRITICAL_IC_BYP_CAP_TYPE identifies the bypass capacitor types that are allowed for a critical IC.

This variable is parameterized. Use the following syntax to specify the bypass capacitor type for each class:

"<*device type*>:<*cap value*>:<*tolerance*>"

For example:

```
#define CRITICAL_IC_BYP_CAP_TYPE "CAP-1:82UF:5%!CAP-3:0.01uf:3%"
, "CAP-2:10uf:4%" , "CAP-4:1.0uf:2%!CAP-5:5uf:5%" , "CAP-5:5uf:5%"
, "CAP-5:5uf:5%"
```

Default Severity

Error

bypass_drvr_rcvr_bidir

Checks that all line drivers, line receivers, and bidirectional transceivers are bypassed. These components must have their bypass capacitors at a specified distance from the power pin. The rule identifies the power pin as having the PINUSE="POWER" property attached.

Properties Used

EMC_COMP_TYPE = LINE_DRIVER

EMC_COMP_TYPE = LINE_RECEIVER

EMC_COMP_TYPE = BIDIR_TRANS

EMC_COMP_TYPE = BYPASS_CAP

PINUSE = POWER

Variables Used

BYP_CAP_SENS_DIST

EMC_BYPASS_CAP_PWR_PIN_DIST

Required Data

Attach the EMC_COMP_TYPE property set to LINE_DRIVER to all line drivers.

Attach the EMC_COMP_TYPE property set to LINE_RECEIVER to all line receivers.

Attach the EMC_COMP_TYPE property set to BIDIR_TRANS to all bidirectional transceivers.

Attach the EMC_COMP_TYPE property set to BYPASS_CAP to all bypass capacitors.

You do not have to attach the PINUSE property to the pins. EMControl reads the PINUSE property directly from the layout database. (PINUSE is a hidden property that is usually present.) You can, however, overwrite the automatic property setting by attaching PINUSE as you would any other property.

If required, you can edit the default values for EMControl variables before using this rule to check your design:

- BYP_CAP_SENS_DIST specifies the bypass capacitor sensitive distance. (This variable is parameterized.)

  The rule selects the maximum value of this variable.
- EMC_BYPASS_CAP_PWR_PIN_DIST specifies the maximum distance between the power pin and the bypass capacitors.

Default Severity

Error

bypass_fast_sw_trans

Checks that all fast-switching transistors are bypassed properly.

Properties Used

EMC_COMP_TYPE = FAST_SWITCH_TRANSISTOR

EMC_COMP_TYPE = BYPASS_CAP

Variables Used

BYP_CAP_SENS_DIST

Required Data

Attach the EMC_COMP_TYPE property set to FAST_SWITCH_TRANSISTOR to all fast switching transistors.

Attach the EMC_COMP_TYPE property set to BYPASS_CAP to all bypass capacitors.

If required, you can edit the value of the BYP_CAP_SENS_DIST variable before using this rule to check your design. BYP_CAP_SENS_DIST defines the bypass capacitor sensitive distance. (This variable is parameterized.)

The rule selects the maximum value of this variable.

Default Severity

Error

critical_IC_3caps_C_2C_4C

Checks the distances between bypass capacitors and critical IC power pins.

This rule ensures that

- Three bypass capacitors are used for each critical IC power pin.
- The three capacitors have values in the ratio of 1:2:4.
- The largest capacitor is farthest from the pin within a distance of CAP1_DIST_FROM_PWR_PIN.
- The second largest capacitor is within a distance of CAP2_DIST_FROM_PWR_PIN.
- The smallest capacitor is closest within a distance of CAP3_DIST_FROM_PWR_PIN.

The rule identifies the power pin by the PINUSE="POWER" property on the pin.

Properties Used

EMC_CRITICAL_IC

EMC_COMP_TYPE = BYPASS_CAP

VALUE

PINUSE = POWER

Variables Used

CAP1_DIST_FROM_PWR_PIN

CAP2_DIST_FROM_PWR_PIN

CAP3_DIST_FROM_PWR_PIN

Required Data

Attach the EMC_COMP_TYPE property set to BYPASS_CAP to all bypass capacitors.

Attach the EMC_CRITICAL_IC property to all critical ICs.

The VALUE property identifies the capacitance value of the capacitor.

You do not have to attach the PINUSE property to the pins. EMControl reads the PINUSE property directly from the layout database. (PINUSE is a hidden property that is usually present.) You can, however, overwrite the automatic property setting by attaching PINUSE as you would any other property.

If required, you can edit the default values for EMControl variables before using this rule to check your design:

- CAP1_DIST_FROM_PWR_PIN specifies the distance of capacitor 1 from the power pin, for example, 300 mils. (This variable is parameterized.)
- CAP2_DIST_FROM_PWR_PIN specifies the distance of capacitor 2 from the power pin, for example, 200 mils. (This variable is parameterized.)
- CAP3_DIST_FROM_PWR_PIN specifies the distance of capacitor 3 from the power pin, for example, 100 mils. (This variable is parameterized.)

Default Severity

Error

critical_IC_loop_area

Calculates the power and ground loop area and compares it with the area calculation based on the geometry of the package. The rule handles both routed power and ground (double-sided PCB) and unrouted power and ground (multilayered PCB) cases.

The rule works on all ICs with the property EMC_CRITICAL_IC. For every power pin of the IC, the closest ground pin and the closest capacitor are considered for area calculations.

`critical_IC_loop_area` identifies the power and ground pins as the pins having the properties PINUSE = “POWER” and PINUSE = “GROUND”, respectively.

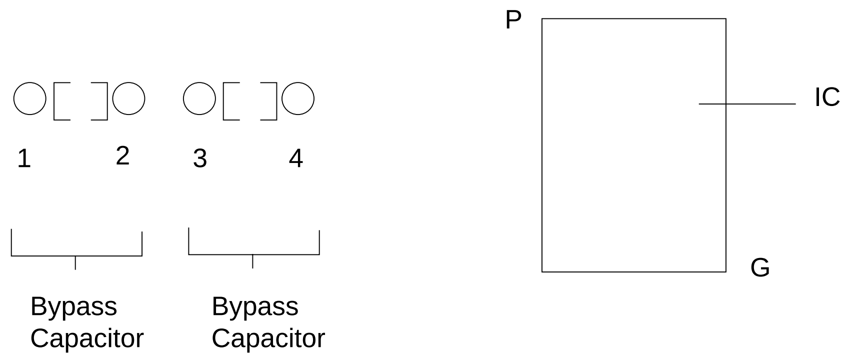
For an IC with the property EMC_CRITICAL_IC, the constraint area is calculated as described in the following example.

In <Hot>Figure 2-1 , P identifies the location on the IC of the power pin and G identifies the ground pin. P-G is taken to be `len1`, the direct distance between the power pin and the ground pin. Using `len1` as the hypotenuse, a right-angled isosceles triangle is constructed. The area of the triangle is calculated to be `Tref`.

The actual loop area is calculated differently for routed and unrouted cases:

- **Unrouted cases** — No routed connections between power and ground pins on the IC and the bypass capacitor.

Figure 2-1 Example for Calculating Loop Area



A triangle is constructed with the following vertices:

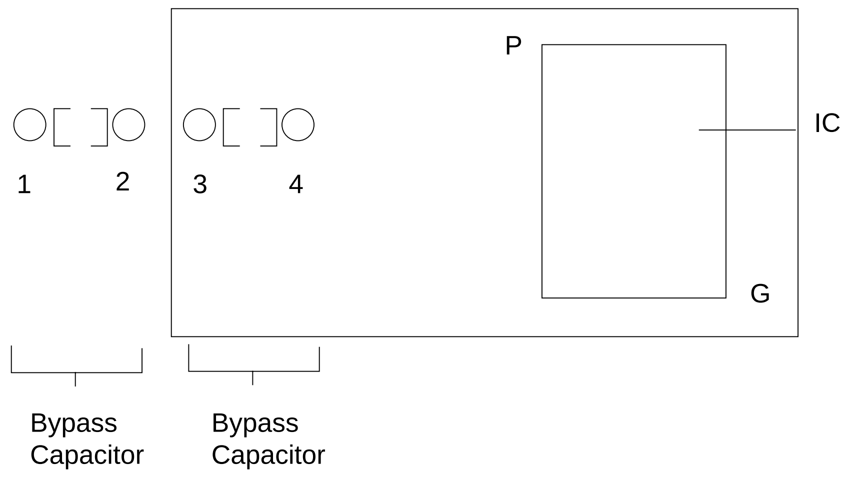
- ❑ The power pin on the IC (P)
- ❑ The ground pin on the IC (G)
- ❑ The capacitor pin that is most distant from the power pin (3)

In the above example, the triangle is P-G-3. The area of this triangle is calculated as `Tact`.

- ■ **Routed cases** — The rule identifies the shortest etch paths from the power and ground pins of the IC to the bypass capacitor.

  The etch that connects the power pin to the capacitor is called `pwr_etch`. The etch that connects the ground pin to the capacitor is called `gnd_etch`.

Figure 2-2 Example of Routed Case



In <Hot>Figure 2-2 , the minimum bounding box is shown as a dotted line. It encloses:

- ❑ The power pin on the IC (P)
- ❑ The ground pin on the IC (G)
- ❑ The bypass capacitor (capacitor with pins 3, 4)
- ❑ The pwr_etch (etch 3-P or 4-P)
- ❑ The gnd_etch (etch 4-G or 3-G)

The area of the minimum bounding box is calculated as `Tact`.

Two areas have now been determined: `Tact` and `Tref`. The ratio of these areas is calculated. A violation is reported if the ratio exceeds the value specified by the LOOP_AREA_COEFFICIENT variable.

Properties Used

EMC_COMP_TYPE = BYPASS_CAP

EMC_CRITICAL_IC

PINUSE = POWER

PINUSE = GROUND

Variables Used

LOOP_AREA_COOEFFICIENT

BYP_CAP_SENS_DIST

Required Data

Attach the EMC_COMP_TYPE property set to BYPASS_CAP to all bypass capacitors.

Attach the EMC_CRITICAL_IC property to all critical ICs.

You do not have to attach the PINUSE property to the pins. EMControl reads the PINUSE property directly from the layout database. (PINUSE is a hidden property that is usually present.) You can, however, overwrite the automatic property setting by attaching PINUSE as you would any other property.

If required, you can edit the default values for EMControl variables before using this rule to check your design:

- LOOP_AREA_COEFFICIENT defines the maximum allowable ratio of the actual area (calculated as `Tact`) and the reference area (calculated as `Tref`). (This variable is parameterized.)
- BYP_CAP_SENS_DIST defines the bypass capacitor sensitive distance. (This variable is parameterized.)

Default Severity

Error

decouple_emc_regions

Checks the boundary between adjacent EMC regions having different EMC frequency classifications for a sufficient number of bypass capacitors. An EMC region is identified by the EMC_CRITICAL_REGION property value attached to it. Bypass capacitors are identified by the property EMC_COMP_TYPE = BYPASS_CAP.

Areas of the board without an assigned EMC_CRITICAL_REGION property are considered to be part of a room having the lowest assigned frequency class. Rooms are not actually merged. When two rooms overlap, the room with the higher frequency class takes precedence.

The "adjacency" of two rooms and their common boundary is defined as follows:

- If the edges of a room and its neighbor are separated by more than the distance specified by BYP_CAP_SENS_DIST, they will not be considered adjacent.

  The area between them will be considered as belonging to the default room of the lowest frequency.

- If the edges of the two rooms are within the distance specified by BYP_CAP_SENS_DIST, the edge of the region corresponding to the higher frequency will be taken as the center of the boundary region where bypass capacitors are to be checked.

  No bypass checking will be done for the lower frequency region.

- A TOP EMC region considers a TOP or BOTH region as its neighbor.

  A BOTTOM region considers only BOTTOM rooms as neighbors.

  Because the edge of a room can have more than one room adjacent to it, the checking along each edge is performed in segments.

Properties Used

EMC_COMP_TYPE = BYPASS_CAP

EMC_CRITICAL_REGION

Variables Used

BYP_CAP_SENS_DIST

BYP_CAP_SEP_DIST

Required Data

Attach the EMC_COMP_TYPE property set to BYPASS_CAP to all bypass capacitors.

Attach the EMC_CRITICAL_REGION property to all critical regions.

If required, you can edit the default values for EMControl variables before using this rule to check your design:

- BYP_CAP_SENS_DIST defines the bypass capacitor sensitive distance. (This variable is parameterized.)
- BYP_CAP_SEP_DIST defines the search window for boundaries. (This variable is parameterized.)

  The BYP_CAP_SEP_DIST variable has one less value for a board than the number of critical frequency regions specified by EMC_CRITICAL_REGION.

Default Severity

Info

fence_off_emc_regions

Checks for the presence of a fence, or boundary area, along the boundaries between EMC regions having different EMC_CRITICAL_REGION property values. This rule also checks for the percentage overlap of the fence and the boundary.

Fences are identified by the property EMC_COMP_TYPE = FENCE. A TOP region checks for fences on the TOP layer only. A BOTH region checks for fences on the TOP and BOTTOM layers. A BOTTOM region checks for fences on the BOTTOM layer only.

Properties Used

EMC_COMP_TYPE = FENCE

EMC_CRITICAL_REGION

Variables Used

FENCE_BOUNDARY_DIST

FENCE_BOUNDARY_RATIO

Required Data

Attach the EMC_COMP_TYPE property set to FENCE to all fences.

Attach the EMC_CRITICAL_REGION property to critical frequency regions.

If required, you can edit the default values for EMControl variables before using this rule to check your design:

- FENCE_BOUNDARY_DIST specifies the sensitive distance from the boundary within which to search for fences.
- FENCE_BOUNDARY_RATIO specifies the minimum ratio of overlap required between the fences and the boundary segments.

Default Severity

Info

Power and Ground Plane Rules

The following rules are included in the `emc_pwr_gnd_dist.rle` file:

- gnd_under_clock
- pwr_gnd_plane_separation

gnd_under_clock

Checks for a symmetrically placed ground shape under each clock generator. Ground shapes should have the same subclass as that of the clock generator footprint (that is, either TOP or BOTTOM depending on where the clock generator is placed).

The rule also reports errors if the ratio of overlap between the ground shapes and package geometry area of the clock generator is less than 0.5 or if there is not at least one ground shape under each quadrant of the package geometry.

Properties Used

EMC_COMP_TYPE = CLOCK_GEN

VOLTAGE

Variables Used

None

Required Data

Attach the EMC_COMP_TYPE property set to CLOCK_GEN to all clock generators.

All ground shapes must be part of a net with the property VOLTAGE = 0.

Default Severity

Warning

pwr_gnd_plane_separation

Checks for a separation between power and ground planes. The rule checks that the distance between a power plane and a ground plane (on the Z axis) is not less than the value specified by MIN_PWR_GND_SEPARATION and not more than the value specified by MAX_PWR_GND_SEPARATION.

Properties Used

None

Variables Used

POWER_PLANE_NAME

GROUND_PLANE_NAME

MIN_POWER_GND_SEPARATION

MAX_POWER_GND_SEPARATION

Required Data

If required, you can edit the values for EMControl variables before using this rule to check your design:

- POWER_PLANE_NAME specifies the default name of the power plane.

  For example:

  ```
  #define POWER_PLANE_NAME "VCC"
  ```
- GROUND_PLANE_NAME specifies the default name of the ground plane.

  For example:

  ```
  #define GROUND_PLANE_NAME "GND"
  ```
- MIN_PWR_GND_SEPARATION identifies the minimum power to ground separation distance on the Z axis.
- MAX_PWR_GND_SEPARATION identifies the maximum power to ground separation distance on the Z axis.

Default Severity

Error

DC Routing Rules

The following rules are included in the `emc_dc_route.rle` file:

- bypass_pwr_trace
- filters_to_clean_ground
- max_pwr_gnd_resistance
- pwr_gnd_trace_width

bypass_pwr_trace

Checks that power traces are bypassed to ground. The rule operates on the power trace segments whose length is more than the value specified by POWER_TRACE_BYPASS_DIST. It counts the number of bypass capacitors on these segments and compares this number with the number of bypass capacitors required on the segment. The number of required bypass capacitors is determined by assuming that bypass capacitors must be equally spaced and within the distance specified by POWER_TRACE_SENS_DIST from the power trace.

For example, if POWER_TRACE_BYPASS_DIST has a value of 500 and the length of a segment of the power trace is 1100 units, the number of bypass capacitors required for the segment is two.

Properties Used

EMC_COMP_TYPE = BYPASS_CAP

VOLTAGE

Variables Used

POWER_TRACE_SENS_DIST

POWER_TRACE_BYPASS_DIST

Required Data

Attach the EMC_COMP_TYPE property set to BYPASS_CAP to all bypass capacitors.

Attach the VOLTAGE property to all DC nets. The most common nets requiring attachment of this property are power and ground. Ground nets must have a VOLTAGE value of zero. Power nets are identified by a nonzero VOLTAGE value.

If required, you can edit the default values for EMControl variables before using this rule to check your design:

- POWER_TRACE_SENS_DIST defines the distance between bypass capacitors and the power trace.
- POWER_TRACE_BYPASS_DIST specifies the length of the power trace for which bypass capacitors should exist.

Default Severity

Warning

filters_to_clean_ground

Checks whether the IO filter components are connected to a clean ground plane. A clean section of the ground plane is defined as a separate shape on the ground plane surrounded by a moat and having only IO and filter components connected to it.

Filter components are identified by the property EMC_COMP_TYPE = FILTER. Ground shapes are on ETCH subclass with a layer type of plane. They are part of a net with the property VOLTAGE = 0.

For each filter connected to an IO component and having more than two pins, at least one pin must connect to a clean ground shape. An error is reported if there is no connection to clean ground.

Properties Used

EMC_COMP_TYPE = FILTER

VOLTAGE

Variables Used

None

Required Data

Attach the EMC_COMP_TYPE property set to FILTER to all filters.

Attach the VOLTAGE property set to zero to all ground shapes.

Default Severity

Error

max_pwr_gnd_resistance

Checks the resistance between the supply pin of an IC and the voltage source, using the maximum permissible value specified by the PIN_PIN_RESISTANCE variable or the PIN_PLANE_RESISTANCE variable, as described below.

Properties Used

VOLTAGE_SOURCE_PIN

VOLTAGE

Variables Used

PIN_PIN_RESISTANCE

PIN_PLANE_RESISTANCE

Required Data

When the design has no dedicated plane layers, attach the VOLTAGE_SOURCE_PIN property to a voltage source pin, such as a connector pin. The VOLTAGE_SOURCE_PIN property is a flag that identifies the voltage source.

Assign the VOLTAGE property to all nonsignal nets. The most common nets requiring attachment of this property are power and ground. Ground nets must have a VOLTAGE value of zero. Power nets are identified by a nonzero VOLTAGE value.

If required, you can edit the default values for EMControl variables before using this rule to check your design:

- PIN_PIN_RESISTANCE specifies the maximum pin-to-pin resistance.

  The PIN_PIN_RESISTANCE variable is used for designs in which no dedicated plane layers exist and one of the pins has the VOLTAGE_SOURCE_PIN property attached.

- PIN_PLANE_RESISTANCE specifies the maximum pin-to-plane resistance.

  The PIN_PLANE_RESISTANCE variable is used for designs that have dedicated plane layers.

Default Severity

Error

pwr_gnd_trace_width

Checks the power and ground trace widths. The rule verifies that the power and ground traces are at least three times the nominal line width or the width defined by GND_PWR_TRACE_WIDTH, whichever is greater. This increases signal integrity and minimizes simultaneous switching noise. EMControl uses the VOLTAGE property to identify the power and ground traces.

Properties Used

VOLTAGE

Variables Used

GND_PWR_TRACE_WIDTH

Required Data

Attach the VOLTAGE property to power and ground. Ground nets must have a VOLTAGE value of zero. Power nets are identified by a nonzero VOLTAGE value.

You can edit the GND_PWR_TRACE_WIDTH variable to specify the minimum power and ground trace width to use.

Default Severity

Error

Signal Routing Rules

The following rules are included in the `emc_sig_route.rle` file:

- critical_net_card_edge_dist
- critical_net_exp_length
- critical_net_man_ratio
- critical_net_via_count
- critical_net_via_pin_ratio
- filtered_IO_signals
- max_critical_net_xtalk
- nets_over_clean_gnd
- no_critical_net_thru_IO_comps
- shield_clock_nets

- return_path_near_signal_via

Some of the signal routing rules use data produced by DF/SigNoise. Before running these rules, you must properly configure EMControl and SigNoise to run together. See “Initializing EMControl” on page 24 and “Initializing SigNoise” on page 26.

critical_net_card_edge_dist

Checks the distance from the critical net to the card edge. The rule verifies that the critical nets do not come within the distance specified by EXT_NET_EDGE_CRITICAL_DIST of the reference plane copper edge. If critical nets are buried, they must not come within the distance specified by INT_NET_EDGE_CRITICAL_DIST of the reference plane copper edge.

Properties Used

EMC_CRITICAL_NET

Variables Used

EXT_NET_EDGE_CRITICAL_DIST

INT_NET_EDGE_CRITICAL_DIST

Required Data

Attach the EMC_CRITICAL_NET property to all nets considered critical from the EMC standpoint.

If required, you can edit the default values for EMControl variables before using this rule to check your design:

- EXT_NET_EDGE_CRITICAL_DIST specifies the minimum distance between critical nets on external layers and copper edge. (This variable is parameterized.)
- INT_NET_EDGE_CRITICAL_DIST specifies the minimum distance between critical nets on internal layers and copper edge. (This variable is parameterized.)

Default Severity

Error

critical_net_exp_length

Checks the exposed length of critical nets. This value must not exceed the value specified by EMC_CRITICAL_EXPOSED_LEN.

Properties Used

EMC_CRITICAL_NET

Variables Used

EMC_CRITICAL_EXPOSED_LEN

Required Data

Attach the EMC_CRITICAL_NET property to all nets considered critical from the EMC standpoint.

If required, you can edit the default values for EMControl variables before using this rule to check your design:

EMC_CRITICAL_EXPOSED_LEN defines the maximum permissible exposed length of a critical net. (This variable is parameterized.)

Default Severity

Warning

critical_net_man_ratio

Checks the length of all critical nets. In order to minimize radiation, critical nets should be routed in inner layers and they should have minimum exposed etch length. The length of critical nets must not exceed the percentage of their manhattan length specified by CRITICAL_TO_MHATTAN_LEN_RATIO.

Properties Used

EMC_CRITICAL_NET

Variables Used

CRITICAL_TO_MHATTAN_LEN_RATIO

Required Data

Attach the EMC_CRITICAL_NET property to all nets considered critical from the EMC standpoint.

If required, you can edit the default values for EMControl variables before using this rule to check your design:

CRITICAL_TO_MHATTAN_LEN_RATIO specifies the ratio of the critical net length to the manhattan length. (This variable is parameterized.)

Default Severity

Error

critical_net_via_count

Checks the number of vias for each critical net. The number should be kept to a minimum to improve reliability, and should not exceed the value specified by EMC_VIA_COUNT.

Properties Used

EMC_CRITICAL_NET

Variables Used

EMC_VIA_COUNT

Required Data

Attach the EMC_CRITICAL_NET property to all nets considered critical from the EMC standpoint.

If required, you can edit the EMC_VIA_COUNT variable to change the default definition for the maximum number of vias allowed for each critical net. (This variable is parameterized.)

Default Severity

Error

critical_net_via_pin_ratio

Checks the via-to-pin ratio on all critical nets. Restricting the number of vias allowed for each net can help enforce EMC.

The rule calculates the ratio of vias to pins and determines whether the ratio exceeds the value specified by MAX_VIA_PIN_RATIO. When the maximum allowed ratio is exceeded, the rule records a violation for the net. The advisor message for a rule violation reports the PINUSE property value for each pin on the net.

Properties Used

EMC_CRITICAL_NET

Variables Used

MAX_VIA_PIN_RATIO

Required Data

Attach the EMC_CRITICAL_NET property to all nets considered critical from the EMC standpoint.

If required, you can edit the MAX_VIA_PIN_RATIO variable to change the default definition for the maximum via-to-pin ratio. (This variable is parameterized.)

Default Severity

Error

filtered_IO_signals

Checks that all signals running to an IO connector are filtered.

Properties Used

EMC_COMP_TYPE = FILTER

Variables Used

None

Required Data

Attach the EMC_COMP_TYPE property set to FILTER to all filter components.

Default Severity

Warning

max_critical_net_xtalk

Checks the coupling from critical nets to neighboring IO nets, and compares the crosstalk with the maximum value specified by the MAX_PEAK_XTALK variable.

This rule requires Crosstalk table to be generated before executing this rule. Note that Geometry window should be equal to XTALK_WINDOW to get desired results. The Geometry window can be set through the Signal Analysis Crosstalk Table dialog box, *Preferences > InterconnectModels*.

For more information, see topic *Generating a Crosstalk Table* in the *SPECCTRAQuest Simulation and Analysis Reference Guide*.

Properties Used

EMC_CRITICAL_NET

Variables Used

XTALK_WINDOW

MAX_PEAK_XTALK

Required Data

Attach the EMC_CRITICAL_NET property to all nets considered critical from the EMC standpoint.

If required, you can edit the default values for EMControl variables before using this rule to check your design:

- XTALK_WINDOW specifies how far the crosstalk algorithms look for neighboring nets to determine the crosstalk value.
- MAX_PEAK_XTALK specifies the maximum crosstalk in mV that can come from any individual EMC_CRITICAL_NET neighbor before a violation is reported.

  The rule uses the larger value of either forward or backward crosstalk to determine whether a violation exists. (This variable is parameterized.)

Default Severity

Error

nets_over_clean_gnd

Checks the routing of nets over clean ground shapes. All nets routed over a clean ground shape should cross the moat (the void surrounding the clean ground shape) at right angles. The nets should have a minimum etch path within the clean ground shape.

This rule also checks whether any components other than filters or connectors are present above the clean ground shape.

Clean ground shapes are metallic islands residing on ground planes. They are surrounded by a moat (a void element).

Properties Used

EMC_COMP_TYPE = FILTER

VOLTAGE

Variables Used

CRITICAL_TO_MHATTAN_LEN_RATIO

Required Data

Attach the EMC_COMP_TYPE property set to FILTER to all filter components.

Attach the VOLTAGE property set to zero to all ground nets.

If required, you can edit the default value for the CRITICAL_TO_MHATTAN_LEN_RATIO variable. CRITICAL_TO_MHATTAN_LEN_RATIO defines the ratio of the critical net length to the manhattan length.

Default Severity

Error

no_critical_net_thru_IO_comps

Checks for the routing of critical nets through IO devices. Critical nets must not be routed through connector footprints.

Properties Used

EMC_CRITICAL_NET

Variables Used

None

Required Data

Attach the EMC_CRITICAL_NET property to all nets considered critical from the EMC standpoint.

Default Severity

Error

shield_clock_nets

Checks whether grounded guard traces exist for all clock nets and that they lie parallel to the clock nets. Clock nets are identified as nets connected to the output pins of a component with the property EMC_COMP_TYPE = CLOCK_GEN. Output pins have the property PINUSE = OUT.

Guard traces should have the following characteristics:

- They should connect to ground planes through vias at intervals equal to lambda/ GUARD_TRACE_VIAS_PER_LAMBDA where lambda is the wavelength associated with the clock net.
- They should be in the same layer as the clock cline.
- They should exist on both sides of the clock net within the distance specified by GUARDING_DISTANCE.

Properties Used

EMC_COMP_TYPE = CLOCK_GEN

PINUSE

VOLTAGE

Variables Used

GUARD_TRACE_VIAS_PER_LAMBDA

GUARDING_DISTANCE

Required Data

Attach the EMC_COMP_TYPE property set to CLOCK_GEN to all clock generators.

You do not have to attach the PINUSE property to the pins. EMControl reads the PINUSE property directly from the layout database. (PINUSE is a hidden property that is usually present.) You can, however, overwrite the automatic property setting by attaching PINUSE as you would any other property.

If required, you can edit the default values for EMControl variables before using this rule to check your design:

- GUARD_TRACE_VIAS_PER_LAMBDA defines the number of vias found on a guard trace for each lambda length, where lambda is the wavelength for the net.
- GUARDING_DISTANCE defines the maximum distance from the clock net to the guard traces.

Default Severity

Error

return_path_near_signal_via

Identifies critical signals that jump the reference planes. It also identifies the vias (jumping vias) at which signals jump their reference planes and ensures a return path close to the jumping vias. A return path in close proximity results in small loop area, hence lesser EMI.

Specify the maximum distance between jumping via and the return capacitor or via with local parameter MAX_RETURN_VIA_DIST.

The capacitor should be of the same or better type (EMC_CRITICAL_IC) than the class of the jumping signal (EMC_CRITICAL_NET).

Note: In practice, a capacitor can be connected to reference planes through vias. These are the vias that are tested for proximity to the jumping via. The distance with the capacitor is not considered. The lead-traces of the capacitor are also not checked for their length.

The property VOLTAGE attached to the nets of the plane is used to determine the type of interconnect required. Absence of this property results in the inability of the rule to determine whether a capacitor or a via should be the interconnect between the planes. In such a case, both capacitor and via are considered valid interconnects. In the absence of both, the rule prescribes putting an `interconnect`.

Caution

Attaching the EMC_CRITICAL_IC property to a capacitor causes the capacitor to be treated as an IC in some rules. For more information on these rules, see Properties Used by the System-Supplied EMControl Rules table on page 28

Properties Used

EMC_CRITICAL_IC

EMC_CRITICAL_NET

EMC_COMP_TYPE

Variables Used

MAX_RETURN_VIA_DIST

Required Data

If required, you can edit the MAX_RETURN_VIA_DIST parameter to change its default value.

Attach EMC_CRITICAL_IC property to the capacitor and EMC_CRITICAL_NET property to the net.

Attach property EMC_COMP_TYPE with value BYPASS_CAP to the capacitor.

It is recommended that you attach property VOLTAGE to the nets of the plane.

Default Severity

Warning

Signal Quality Rules

The following rules are included in the `emc_sig_qual.rle` file:

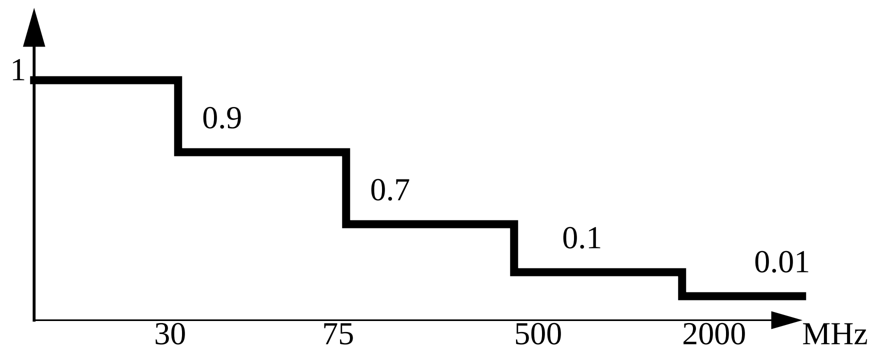
- clock_spectral_content
- critical_net_ringing
- critical_net_termination
- single_diff_mode_EMI
- sum_diff_mode_EMI

The signal quality rules use data produced by DF/SigNoise. Before running these rules, you must properly configure EMControl and SigNoise to run together. See “Initializing EMControl” on page 24 and “Initializing SigNoise” on page 26

clock_spectral_content

Checks the frequency content on clock nets and compares the spectrum with the MAX_CLOCK_SPECTRAL_CONTENT variable. MAX_CLOCK_SPECTRAL_CONTENT defines a staircase limit for the spectrum (<Hot>Figure B-3).

Figure B-3 MAX_CLOCK_SPECTRAL_CONTENT



When you execute the clock_spectral_content rule, EMControl starts a DF/SigNoise pulse simulation for each clock generator output net in the design and supplies SigNoise with the pulse frequency and duty cycle for the clock signal stimuli as specified by the PULSE_DUTY_CYCLE variable. The number of cycles simulated is determined by the MAX_SIMULATION_CYCLE variable.

EMControl reports a warning message for each net that violates the limits defined by the MAX_CLOCK_SPECTRAL_CONTENT variable. For example, <Hot>Figure B-3 defines the following limit:

```
MAX_CLOCK_SPECTRAL_CONTENT =  "0MHz:1V 30MHz:0.9V 75MHZ:0.7V 500MHz:0.1V
2GHz:10mV"
```

This limit specifies that the strength of all clock signals must be:

- Less than 1.0 V from DC to 30 MHz
- Less than 0.9 V from 30 MHz to 75 MHz
- Less than 0.7 V from 75 MHz to 500 MHz
- Less than 0.1 V from 500 MHz to 2 GHz
- Less than 0.01 V for frequency higher than 2 GHz

When you select a warning message in this list, EMControl does the following:

- Highlights the net in the design.

- Opens the DF/SigNoise waveform window (SigWave) and displays the waveform for the first load on the selected net.

  SigWave performs a Fast Fourier Transform (FFT) on the waveform and displays the results.

Selecting a different clock signal in the message list repeats the process for the selected net. Selecting another load from the SigWave pop-up menu updates the waveform and spectrum displays.

Properties Used

EMC_CRITICAL_NET

EMC_COMP_TYPE = CLOCK_GEN

PINUSE

Variables Used

PULSE_DUTY_CYCLE

MAX_CLOCK_SPECTRAL_CONTENT

MAX_SIMULATION_CYCLE

Required Data

Attach the EMC_CRITICAL_NET property to all nets considered critical from the EMC standpoint.

Attach the EMC_COMP_TYPE property set to CLOCK_GEN to all clock generators.

You do not have to attach the PINUSE property to the pins. EMControl reads the PINUSE property directly from the layout database. (PINUSE is a hidden property that is usually present.) You can, however, overwrite the automatic property setting by attaching PINUSE as you would any other property.

If required, you can edit the default values for EMControl variables before using this rule to check your design:

- PULSE_DUTY_CYCLE defines the pulse frequency and duty cycle for the respective EMC_CRITICAL_NET properties.

- MAX_CLOCK_SPECTRAL_CONTENT defines a staircase limit for the frequency content of critical clock nets.
- MAX_SIMULATION_CYCLE defines the number of cycles SigNoise simulates.

Default Severity

Warning

critical_net_ringing

Checks the ringing of critical nets by verifying that overshoot and undershoot do not exceed a specified percentage of the voltage swing.

This rule requires that DF/SigNoise be run in the background. Before you run this rule, you must properly configure EMControl and SigNoise to run together. See “Initializing EMControl” on page 24 and “Initializing SigNoise” on page 26

For each net with the EMC_CRITICAL_NET property, the critical_net_ringing rule verifies that the maximum overshoot and undershoot obtained by SigNoise simulation is less than the value of the ratio defined by MAX_OVER_UNDERSHOOT multiplied by the smallest voltage swing because of driver pins on the net.

For example, if the value of MAX_OVER_UNDERSHOOT is 0.15 for a net that has a 0 V to 5 V swing, the maximum amount of overshoot or undershoot will be determined as follows:

```
    0.15 * 5V = 0.75V
```

In this example, any value greater than 0.75 V is considered to be a violation.

Properties Used

EMC_CRITICAL_NET

Variables Used

MAX_OVER_UNDERSHOOT

Required Data

Attach the EMC_CRITICAL_NET property to all nets considered critical from the EMC standpoint.

If required, you can edit the default value for the MAX_OVER_UNDERSHOOT variable before using this rule to check your design. MAX_OVER_UNDERSHOOT specifies the ratio used to calculate the maximum permitted overshoot and undershoot. (This variable is parameterized.)

Default Severity

Error

critical_net_termination

Checks for the termination of critical nets. Critical signals must be terminated when the driver output rise or fall time is less than twice the propagation delay.

This rule requires data produced by DF/SigNoise. Before running this rule, you must properly configure EMControl and SigNoise to run together. See “Initializing EMControl” on page 24 and “Initializing SigNoise” on page 26

Properties Used

EMC_CRITICAL_NET

TERMINATOR_PACK

Variables Used

None

Required Data

Attach the EMC_CRITICAL_NET property to all nets considered critical from the EMC standpoint.

Attach the TERMINATOR_PACK property to all components that are terminator packs (that is, they contain terminator resistors).

Default Severity

Error

single_diff_mode_EMI

Checks for excessive differential mode EMI produced by critical nets routed on external layers.

This rule requires that DF/SigNoise be run in the background. Before you run this rule, you must properly configure EMControl and SigNoise to run together.See “Initializing EMControl” on page 24 and “Initializing SigNoise” on page 26

When you execute the single_diff_mode_EMI rule, EMControl starts a DF/SigNoise pulse simulation for each critical net in the design and supplies SigNoise with the pulse frequency and duty cycle for the clock signal stimuli as specified by the PULSE_DUTY_CYCLE variable. The number of cycles simulated is determined by the MAX_SIMULATION_CYCLE variable.

Each simulation produces the differential mode current at the driver pin for a critical net. This current is assumed to be flowing for the length of the trace. From this information, the differential mode EMI for the net is estimated and compared against the NET_EMI_REGULATION value for the net.

EMControl reports an error message in the EMC Advisor window for each net that violates the limit defined by the NET_EMI_REGULATION variable. Each message includes one of the following recommendations for removing the violation:

- Use a series terminator to reduce the high frequency content
- Reduce the exposed length on the net

When you select a message from the list in the EMC Advisor window, EMControl does the following:

- Highlights the net in the design
- Opens the DF/SigNoise waveform window (SigWave) and displays the emission level for the net

Properties Used

EMC_CRITICAL_NET

PINUSE

Variables Used

NET_EMI_REGULATION

PULSE_DUTY_CYCLE

MAX_SIMULATION_CYCLE

Required Data

Attach the EMC_CRITICAL_NET property to all nets considered critical from the EMC standpoint.

You do not have to attach the PINUSE property to the pins. EMControl reads the PINUSE property directly from the layout database. (PINUSE is a hidden property that is usually present.) You can, however, overwrite the automatic property setting by attaching PINUSE as you would any other property.

If required, you can edit the default values for EMControl variables before using this rule to check your design:

- NET_EMI_REGULATION defines the constraint set for EMI emissions against which nets are measured.
- PULSE_DUTY_CYCLE defines the pulse frequency and duty cycle for the respective EMC_CRITICAL_NET properties.
- MAX_SIMULATION_CYCLE defines the number of cycles SigNoise simulates.

Default Severity

Error

sum_diff_mode_EMI

Checks an entire design for total EMI emissions.

The sum_diff_mode_EMI rule does the following:

- Calculates the differential mode EMI for each critical net as described for the single_diff_mode_EMI rule
- Sums the EMI values for individual nets
- Compares the total EMI to the appropriate BOARD_EMI_REGULATION value
- Reports an error message in the EMC Advisor window if the limit is exceeded

This rule requires that DF/SigNoise be run in the background. Before you run this rule, you must properly configure EMControl and SigNoise to run together.See "Initializing EMControl" on page 24 and "Initializing SigNoise" on page 26

Properties Used

EMC_CRITICAL_NET

PINUSE

Variables Used

BOARD_EMI_REGULATION

PULSE_DUTY_CYCLE

MAX_SIMULATION_CYCLE

Required Data

Attach the EMC_CRITICAL_NET property to all nets considered critical from the EMC standpoint.

You do not have to attach the PINUSE property to the pins. EMControl reads the PINUSE property directly from the layout database. (PINUSE is a hidden property that is usually present.) You can, however, overwrite the automatic property setting by attaching PINUSE as you would any other property.

If required, you can edit the default values for EMControl variables before using this rule to check your design:

- BOARD_EMI_REGULATION defines the constraint set for EMI emissions against which boards are measured.
- PULSE_DUTY_CYCLE defines the pulse frequency and duty cycle for the respective EMC_CRITICAL_NET properties.
- MAX_SIMULATION_CYCLE defines the number of cycles SigNoise simulates.

Default Severity

Error

C

EMControl Predicates

Overview

Predicates are the routines supplied by Cadence to perform the various tasks required by the EMControl rules. You can use the Cadence predicates to write your own rules. Predicates call functions that are written in SKILL or the C language.

This appendix describes the physical environment objects and predicates supplied with EMControl.

Physical Environment Objects and Predicates

Table 3-1 lists the primary objects that exist in the layout physical environment that can be referenced by the EMControl predicates.

Table 3-1 Objects Accessed by EMControl Predicates

Object	Description
design	Specifies the complete design or a user-specified portion of the design
component	Specifies an instance of a physical component in the design
via	Defines an opening in a dielectric layer that connects adjacent conductor layers
net	Specifies a signal corresponding to a logical net in the design
pin	Specifies an interface terminal of a component
shape	Specifies a shape, rectangle, or filled rectangle in the design
polygon	Specifies a polygon in the layout x-y plane

The following sections describe the predicates that allow you to access information in the layout database. Each section lists and describes the predicates that apply to one of the object categories described in Table B-1. Within each section, predicates appear in alphabetical order.

A call to a predicate takes one or more arguments and returns a list comprised of one or more values.

General-Purpose Predicates

This section describes predicates that you can apply to values in the layout database.

append

`list` `append(list1, list2)`

Appends `list2` to `list1`, modifying `list1`. The lists can be object lists or value lists.

atof

`value` `atof(string)`

Converts the number `string` to a real number.

atoi

`value` `atoi(string)`

Converts the number `string` to an integer number.

compareProfile

`value` `compareProfile(`*`valProfile, valLimit, valFlag, valLength`*`)`

Compares a list of frequency and value pairs with a predefined profile limit. It returns `true` if it exceeds the limit.

- valProfile is the value returned from the `getSpectralContent` predicate.
- valLimit is a user-defined parameter.
- The value of valFlag varies depending on the format in which you specify valLimit:
  - When valLimit is given in dBuV format, valFlag = 1. For example:

```
#define valLimit “30MHz:49.5dBuV 88MHz:54.0dBuV 216MHz:56.5dBuV
960MHz:60dBuV”
```

  - When valLimit is given in V format, valFlag = 0 if. For example:

```
#define valLimit "0MHz:5.0V 30MHz:0.9V 75MHz:0.7V   500MHz:0.1V"
```

- The value of Length varies depending on the rule being used:
  - When used in the spectral_content rule, valLength = 0.0
  - When used in the single_diff_mode_EMI rule, valLength is the exposed trace length
  - When used in the sum_diff_mode_EMI rule, valLength < 0.0

component

`value` `component(value)`

Converts the component ID `value` to the component itself.

convertUnits

`value` `convertUnits(value, unit)`

Converts *value* into the units specified by `unit`.

count

`value` `count(list)`

Returns the number of elements in the list. The list can be an object list or a value list.

float

`value` `float(value)`

Converts the integer `value` to a real number.

format

`value` `format(`*`value, value`*`)`

Formats a string (the first `value`) such that its length is the integer specified by the second `value`. When the integer is less than the string length, the string is truncated at the trailing end. Otherwise, extra blank characters are inserted.

getAllSubstrings

`value` `getAllSubstrings(string1, startToken, endToken)`

Returns all substrings of `string1` that occur between `startToken` and `endToken`. The returned string does not include `startToken` and `endToken`.

For example:

```
getAllSubstrings("VCC=VEE;VDD=VSS;","=",";")
```

returns a list containing VEE and VSS.

getDistance

```
value  getDistance(value ,value ,value ,value)
```

Finds the distance between two points, (x1,y1) and (x2,y2), where x1 and y1 are the first and second `values` and x2 and y2 are the third and fourth `values` three and four.

getListSumMerge

```
value  getListSumMerge(value)
```

Merges two pairs of numbers returned by the `getSpectralContent` predicate. The `getListSumMerge` predicate is used to compute the total emissions from a number of nets.

getNthListOfSkillList

```
value  getNthListOfSkillList(index, list)
```

Returns the element in the specified SKILL `list` that is pointed to by `index`.

getShapeOutline

```
value getShapeOutline("class_name/subclass_name")
```

Returns the outline of the shape specified by the `class_name` and `subclass_name`.

getValueOneOfList

```
value  getValueOneOfList(ARL_optional_list)
```

Returns the list of values that are passed as an optional list of values to `getValueOneOfList`. This predicate is for printing an optional list specified by the user in the rule message field.

For example:

```
val1 = getValueOneOfList( {"A", "B", "C", "D" } )
```

In the above example, `val1` is the list containing the values:
`“A”, “B”, “C”, “D”.`

index

`*value*  index(list, value)`

Returns the index to the first occurrence of `value` in the `list`.

integer

`*value*  integer(realValue)`

Returns the integer value of the `realValue`.

isNull

`*value*  isNull(object)`

Determines whether the value of `object` is null. If so, `isNull` returns *t*. Otherwise, it returns nil.

isTrue

`*value*  isTrue(boolean)`

Returns *t* if the `boolean` value is true. Returns nil if the `boolean` value is false.

matchCapType

`*value*  matchCapType(captype1, captype2)`

Compares the capacitor types of two bypass capacitors. Returns *t* if they match. Otherwise, it returns nil.

The capacitor type for bypass capacitors must be in the following format:

```
<device type>:<cap value>:<tolerance>
```

max

`*value*  max(list)`

Returns the maximum value of the `list`.

min

`value` `min(list)`

Returns the minimum value of the `list`.

nth

`value` `nth(list, integerValue)`

Returns the element in the `list` at the position indexed by `integerValue.`

ntoa

`value` `ntoa(`*`value`*`)`

Converts the numerical argument into a string.

skipWhiteSpaces

`value` `skipWhiteSpaces(string)`

Strips spaces, tabs, and linefeeds from the specified *string*, unless they are enclosed within single quotation marks (for example, ‘`\t`’ is not stripped). Returns the resulting string.

unique

`value` `unique(list)`

Returns a list of the elements in the input *list*. Multiple occurrences of any element are filtered out of the returned list. For example, for the following input:

```
unique(2, 3, 3, 4, 6, 2, 3)
```

`unique` returns the following list:

```
(2, 3, 4, 6)
```

upperCase

`value` `upperCase(string)`

Converts the input *string* to a string that has all uppercase letters.

Design Predicates

This section describes predicates that you can apply to the design object.

component

`component` `component(design)`

Returns all the component instances in `design`.

getDiffPSegs

`net` `getDiffPSegs(design)`

Returns the trace segment for each differential pair that has the maximum length in the `design`. The trace segment is part of the differential pair net. Differential pair nets are nets with the same DIFFERENTIAL_PAIR property assignment.

getGroundNet

`net` `getGroundNet(design)`

Returns the ground net in `design`.

getLayerSeparation

`value` `getLayerSeparation(design, "layer1", "layer2")`

Returns the vertical layer separation distance between `layer1` and `layer2` of `design`. `layer1` and `layer2` are the names of the layers. Layer names must be enclosed in quotation marks.

getNominalNetWidth

`value` `getNominalNetWidth(design)`

Returns the nominal etch width of the nets in `design`. The value is returned in user units.

getPowerNet

`net` `getPowerNet(design)`

Returns the power net in `design`.

name

`value` `name(`*`design`*`)`

Returns the design (board) name

net

net `net(design)`

Returns all the nets in `design`.

pin

pin `pin(design)`

Returns all the component pins in `design`.

shape

shape `shape(`*design*`)`

Returns all the shape objects in `design`.

via

via `via(design)`

Returns all the vias in `design`.

Component Predicates

This section describes predicates that you can apply to an object that is a component.

getCenter

value `getCenter(`*component*`)`

Returns the center of the component as a list of two numbers representing the coordinate.

getComp2CompDistance

value `getComp2CompDistance(component1, component2)`

Returns the distance between `component1` and `component2`.

getCompC1eqTwiceC2eqTwiceC3

value `getCompC1eqTwiceC2eqTwiceC3(component, distance,` *propertyValue*`)`

Returns all the components, in the area represented by the bounding box, that have the value for the EMC_COMP_TYPE property specified by `propertyValue` (see Table 2-1 on page 28 for legal values for EMC_COMP_TYPE).

getCompC1eqTwiceC2eqTwiceC3 determines the bounding box area as follows:

```
((x1-distance, y1-distance) (x2+distance, y2+distance))
```

where ((x1,y1) (x2, y2)) specifies the diagonal coordinates of the bounding box of the component.

If the value of *propertyValue* is *ALL*, this predicate returns all of the components in the calculated area.

getCompC1eqTwiceC2eqTwiceC3 is used to find the bypass capacitors for a given component. You should use the property EMC_COMP_TYPE with an assigned value of BYPASS_CAP with this predicate.

Once all the bypass capacitors for a component are found, the predicate gets the value for the VALUE property on the capacitors, and looks for all the capacitors that satisfy the following equation:

```
1st cap = twice (2nd cap), 2nd Cap  = twice (3rd cap)
```

getCompC1eqTwiceC2eqTwiceC3 returns the list of the capacitors that match the equation in a fixed order:

- The first element of the list is the C1 capacitors.
- The second element is the C2 capacitors.
- The third element is the C3 capacitors.

For example, in a case where a component has five bypass capacitors with the following values:

- C1 = 0.01UF
- C2 = 0.03UF
- C3 = 0.02UF
- C4 = 0.01UF
- C5 = 0.04UF

getCompC1eqTwiceC2eqTwiceC3 finds all the capacitors that satisfy the equation described above. For the example, these are:

- 1st Cap = C5

- 2nd Cap = C3
- 3rd Cap = C1, C4

For the example, `getCompC1eqTwiceC2eqTwiceC3` returns the following list:

```
(C5, C3, (C1, C4))
```

getCompInArea

`component` `getCompInArea(component, distance,` *`propertyValue`*`)`

Returns all the components with the EMC_COMP_TYPE definition specified by `propertyValue` in the area represented by the bounding box (see Table 2-1 on page 28 for legal values for EMC_COMP_TYPE). `getCompInArea` determines the area as follows:

```
((x1-distance, y1-distance) (x2+distance, y2+distance))
```

where ((x1,y1) (x2, y2)) specifies the diagonal coordinates of the bounding box of the component.

If the value of *propertyValue* is *ALL*, *getCompInArea* returns all of the components in the calculated area.

getComponentType

`value` `getComponentType(component)`

Returns a string that provides the component type of `component`. The component type is returned in the following format:

```
<device type>:<value>:<tolerance>
```

`getComponentType` is usually used to get the bypass capacitor type.

getCompsConnToComp

`component` `getCompsConnToComp(component, bypassDiscreteFlag)`

Returns all the components connected to `component`. If `bypassDiscreteFlag` is `t`, `getCompsConnToComp` ignores the discrete components connected to `component` and scans further until it finds a nondiscrete component. If *bypassDiscreteFlag* is false, `getCompsConnToComp` stops scanning at any discrete component.

A component is considered to be connected to another component if the connecting net is a signal net. All nonsignal nets must have the VOLTAGE property assigned to them.

getDesign

design `getDesign(component)`

Returns the design of `component`.

getLayer

value `getLayer(component)`

Returns TOP or BOTTOM depending on the location of the component on the board.

getNetsConnToComp

net `getNetsConnToComp(component)`

Returns all nets connected to `component`.

getNetsInArea

net `getNetsInArea(component)`

Returns all nets in the bounding box of component.

getPin

pin `getPin(component)`

Returns all pins of `component`.

getPolygon

polygon `getPolygon(`*component*`)`

Returns a polygon representing the package bounding box of the component.

getPropertyValue

value `getPropertyValue(component, propertyName)`

Returns the value of `propertyName` attached to `component`. Returns nil if `propertyName` is not found.

hasProperty

component `hasProperty(component, propertyName)`

Returns *t* if the property specified by `propertyName` is attached to the `component`. Otherwise, it returns nil.

isCompClass

`value` `isCompClass(component,` *`compClass`*`)`

Returns *t* if `component` has the component class specified by `compClass`. Legal values for `compClass` are `IO`, `IC`, or `DISCRETE`.

isConnectedToNet

`component` `isConnectedToNet(component, net)`

Returns *t* if `component` is connected to `net`. Otherwise, it returns nil. A connector has a component class of `IO`.

isConnector

`component` `isConnector(component)`

Returns *t* if *component* is a connector. Otherwise, it returns nil. A connector has a component class of `IO`.

isSeriesResistor

`component` `isSeriesResistor(component)`

Checks whether `component` is a series resistor. Returns *t* if the `component` is a series resistor. Otherwise, it returns nil.

A component is identified as a series resistor by the following:

- It has the TERMINATOR_PACK property attached.
- The nets connected to the `component` do not have the VOLTAGE property attached.

name

`value` `name(component)`

Returns the reference designator of `component`.

Net Predicates

This section describes predicates that you can apply to an object that is a signal (net).

getClineLength

`*value*  getClineLength(*net*)`

Returns the length of the cline.

getClineOfSeg

`*net*  getClineOfSeg(*net*)`

Returns the parent cline of the cline segment.

getCompInArea

`*component*  getCompInArea(netSeg, distance, *propertyValue*)`

Returns all the components with the EMC_COMP_TYPE definition specified by `propertyValue` in the area represented by the bounding box. (see Table 2-1 on page 28 for legal values for EMC_COMP_TYPE.)

`The getCompInArea` predicate determines the area as follows:

`((x1-*distance*, y1-*distance*) (x2+*distance*, y2+*distance*))`

where ((x1, y1) (x2, y2)) specifies the diagonal coordinates of the bounding box of the net segment.

If the value of *propertyValue* is *ALL*, *getCompInArea* returns all of the components in the calculated area.

Note: The `netSeg` argument must represent a trace segment and not the complete net.

getCompsConnToNet

`*component*  getCompsConnToNet(net)`

Returns all the components connected to `net`.

getDelayTimeUnits

`*value*  getDelayTimeUnits(net)`

Returns the delay time units for `net`.

getDesign

`*design*  getDesign(net)`

Returns the design of `net`.

getEtchLength

`*value*  getEtchLength(*net, shape*)`

Returns the etch length of `net` over `shape`.

getManhattanLen

`*value*  getManhattanLen(net)`

Returns the manhattan length of `net`.

getManhattanLength

`*value*  getManhattanLength(*net, shape*)`

Returns the manhattan length of `net` over `shape`.

getMaxOverUnderShoot

`*value*  getMaxOverUnderShoot(net*1*)`

Returns the maximum undershoot and overshoot values for the net specified by `net1`.

getMaxPropDelay

`*value*  getMaxPropDelay(net, units)`

Returns the maximum propagation delay for `net` in the units specified by `units`.

getMaxRiseTime

`*value*  getMaxRiseTime(net, units)`

Returns the slew rate of the fastest signal that passes through the net in the units specified by `units`.

getMinDistance

value `getMinDistance(net, lineSegment)`

Returns the minimum distance between `net` and `lineSegment`. `lineSegment` can be any other net or the design outline.

getMinNetWidth

value `getMinNetWidth(net)`

Returns the minimum width of `net` in user units.

getMinWaveLength

value `getMinWaveLength(`*net*`)`

Returns the minimum wavelength of the signal on the net.

getNetCapacitance

value `getNetCapacitance(net, units)`

Returns the net capacitance for `net` in the units specified by `units`.

getNetLenOnLayer

value `getNetLenOnLayer(net, layerName)`

Returns the physical length of `net` on the layer specified by `layerName`.

getNetOfSeg

net `getNetOfSeg(net1)`

Returns the parent net of the net segment `net1`.

getNetOnLayer

value `getNetOnLayer(net, layerName)`

Returns all the clines of `net` on the layer specified by `layerName`.

getNetSegOnLayer

net `getNetSegOnLayer(net, layerName)`

Returns all trace segments of the parent `net` on the layer specified by `layerName`.

getNetShape

`value` `getNetShape(net)`

Returns the shape that is a part of `net`. Normally, a supply net has a shape as part of the net.

getNetsInArea

`net` `getNetsInArea(net, distance)`

Returns all nets found within the specified `distance` of `net`.

getNetSourceImpedance

`value` `getNetSourceImpedance(net, resUnit)`

Returns the source impedance for `net` in the units specified by `resUnit`.

getParallelNetsToSeg

`net` `getParallelNetsToSeg(net1, distance)`

Returns all net segments that are parallel to the net segment `net1` and are within the specified `distance` from `net1`. A net segment is considered to be parallel to another if the segments are parallel for at least 85 percent of the segment length of `net1`.

getParallelTraces

`net` `getParallelTraces(`*`net, value, value`*`)`

Returns all cline segments that are parallel to the given cline segment (the first argument, *net*) and are located within a specified distance (the second argument, *value*) in the specified direction (the third argument, `value`). The direction can be either UL (for upper left) or LR (for lower right).

getPinsOnNet

`pin` `getPinsOnNet(net)`

Returns all pins on `net`.

getPropertyValue

`*value* getPropertyValue(net, propertyName)`

Returns the value of `propertyName` attached to the `net`. Returns nil if `propertyName` is not found.

getSegLength

`*value* getSegLength(netSeg)`

Returns the length, in user units, of the line segment specified by `netSeg`. The `netSeg` argument must represent a trace segment and not the complete net.

getTraceLengthInWindow

`*value* getTraceLengthInWindow(*net, net, value*)`

Returns the length of cline (the first argument, `net`) enclosed in a rectangular window. The rectangle has a specified length (the second argument, `net`) and a height two times the specified `value` (the third argument). The length bisects the window.

For example, specify c1, c2, and d as the arguments. The length of cline is c1. It is enclosed in a rectangular window that has c2 as its length. c2 also bisects the window. The height of the window is two times d.

getViasOnCline

`*via* getViasOnCline(*net*)`

Returns the vias on the cline.

getViasOnNet

`*via* getViasOnNet(net)`

Returns all vias on `net`.

getXtalkBetweenNets

`*value* getXtalkBetweenNets(net1, net2)`

Returns the crosstalk contribution in mV on `net2` caused by `net1.`

hasProperty

net `hasProperty(net, propertyName)`

Returns the net if the property specified by `propertyName` is attached to `net`. Otherwise, it returns nil.

isNetTerminated

value `isNetTerminated(net)`

Returns `t` if `net` is terminated.

name

value `name(net)`

Returns the name of `net`.

Pin Predicates

This section describes predicates that you can apply to an object that is a pin.

getActualArea

value `getActualArea(pin, propertyValue, `*`sens_distance`*`)`

Returns the `actual area` for the IC with respect to the power `pin` and the nearest bypass capacitor of the IC. The nearest bypass capacitor is identified as the nearest component within the distance specified by `sens_distance` that has the EMC_COMP_TYPE property value specified by `propertyValue.` (see Table 2-1 on page 28 for legal values for EMC_COMP_TYPE.)

The actual area is calculated differently for routed and unrouted cases:

Unrouted Cases:

For these cases, the nearest bypass capacitor is either not connected to the power pin and ground pin or the connection is partial. A triangle is constructed, with the vertices being

- The power pin
- The nearest ground pin on the IC
- The bypass capacitor pin most distant from the power pin

The area of this triangle defines the actual area.

Routed Cases:

For these cases, the bypass capacitor is connected to the power and ground pins by etch. The etch that connects the power pin to the capacitor is called `pwr_etch`. Similarly, the etch that connects the ground pin to the capacitor is called `gnd_etch`.

A minimum bounding box is constructed. The box contains

- The power pin of the IC
- The ground pin of the IC
- The package geometry of the bypass capacitor
- The `pwr_etch`
- The `gnd_etch`

The calculated area of this bounding box defines the actual area. See **** for a description of the rule `check_loop_area`, which uses this predicate.

getAllegroPinUse

```
value  getAllegroPinUse(pin)
```

Returns the PINUSE of the specified `pin`. `getAllegroPinUse` extracts the PINUSE from the layout database. If you have explicitly attached the PINUSE property to a pin, the value of the PINUSE property overrides the PINUSE attached to the component definition.

getCompInArea

```
component  getCompInArea(pin, distance, propertyValue)
```

Returns all components with the EMC_COMP_TYPE definition specified by `propertyValue` in the area represented by the bounding box. (see Table 2-1 on page 28 for legal values for EMC_COMP_TYPE.) `getCompInArea` determines the area as follows:

```
((x1-distance, y1-distance) (x2+distance, y2+distance))
```

where ((x1, y1) (x2, y2)) specifies the diagonal coordinates of the bounding box of the pin.

If the value of *propertyValue* is *ALL*, *getCompInArea* returns all of the components in the calculated area.

getComponent

`component` `getComponent(pin)`

Returns the component of which the specified `pin` is a part.

getDistance

`value` `getDistance(pin, component)`

Returns the distance from the specified `pin` to the `component`.

getDistance

`value` `getDistance(pin1, pin2)`

Returns the pin-to-pin distance.

getLoopDistance

`value` `getLoopDistance(pin, byp_cap_prop,` *`sens_distance`*`)`

Returns the loop distance from the pin to the nearest bypass capacitor to the ground pin, then back to the pin.

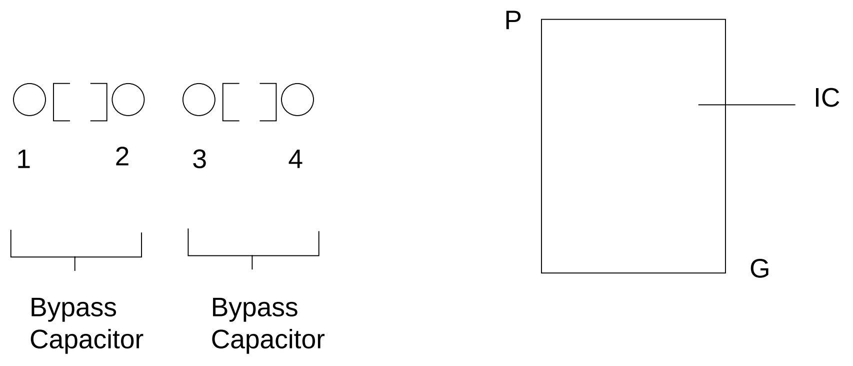
The bypass capacitors must have the EMC_COMP_TYPE property definition specified by `byp_cap_prop` (for example, BYPASS_CAP), and they must be within the distance specified by `sens_distance` from the component boundary.

For an IC marked EMC_CRITICAL_IC, the loop distance is calculated as follows:

1. From the power pin of the IC to the nearest bypass capacitor pin
2. From that bypass capacitor pin to the other (second) bypass capacitor pin for that capacitor
3. From that second bypass capacitor pin to the nearest ground pin on the same IC
4. From the nearest ground pin on the same IC back to the original power pin

For example, in the following figure, P identifies the location on the IC of the power pin and G identifies the ground pin. Pin 3 is assumed to be attached to the ground plane. The pin-to-pin connections that the rule needs to calculate are P-4, 4-3, 3-G, and G-P.
Note that 1-2 is ignored because, even if it is connected to power and ground, the check is always made on the closest bypass capacitor.

Example of Checking Power-Ground Loop Distance



In the cases of P-4 and 3-G, real etch might or might not exist. In such cases, the real etch length is used for the loop distance calculations; otherwise, manhattan length is used. In the case of
4-3 and G-P, no real etch can exist in the layout, and the rule always uses pin-center to pin-center distances.

getMinRiseFallTime

`value` `getMinRiseFallTime(pin, units)`

Returns the minimum rise time or fall time, whichever is higher, of `pin` in the specified `units`.

getNetName

`value` `getNetName(pin)`

Returns the name of the net connected to the specified `pin`.

getNetOfPin

`net` `getNetOfPin(`*`pin`*`)`

Returns the net *pin* is a part of.

getPin2PinResistance

`value` `getPin2PinResistance(Pin1, Pin2)`

Returns the resistance between `Pin1` and `Pin2`.

getPin2PlaneResistance

value `getPin2PlaneResistance(pin, planeShape)`

Returns the resistance between `pin` and the supply plane specified by `planeShape`.

getPropertyValue

value `getPropertyValue(pin, propertyName)`

Returns the value of `propertyName` attached to `pin`. Returns nil if `propertyName` is not found.

getReferenceArea

value `getReferenceArea(pin)`

Returns the reference area of the IC with respect to the specified power `pin`. The reference area is calculated as follows:

- The nearest ground pin for this power pin on the IC is identified.
- C is taken to specify the distance between this ground pin and the power pin.
- A right-angled isosceles triangle is constructed, with C as the hypotenuse.
- The area of this triangle defines the reference area.

See **** for a description of the rule `check_loop_area`, which uses this predicate.

getShapesConnToPin

shape `getShapesConnToPin(`*pin*`)`

Returns all shapes physically connected to `pin` through thermal-relief clines.

getSpectralContent

value `getSpectralContent(`*pinDriver, val_lengthl, valDuty, valCycle, valClassNum*`)`

Performs pulse simulations for all the drivers on the xnet that connect to `pinDriver`. Returns the result of the simulation.

- `pinDriver` is a driver pin
- `valLengthl` is the trace length in meters:

- ❑ When `valLengthl` = 0, the waveform at the driver pin is transformed to the frequency domain by means of a DFT.
- ❑ When `valLengthl` > 0, the emission level caused by the net is computed using a closed-form equation and a list of frequencies.

- `valDuty` is a parameter containing the clock frequency and duty cycles.

  For example, *valDuty* could be the EMControl variable PULSE_DUTY_CYCLE. For example:

  ```
  #define PULSE_DUTY_CYCLE  “300MHz:50%, 75MHz:50%, 50MHz:50%”
  ```

- `valCycle` is the maximum number of cycles the circuit simulator would simulate.
- `valClassNum` is the class of the net (for example, CLASS1).

getVoltageSwing

`*value*  getVoltageSwing(pin)`

Returns the voltage swing for `pin`.

hasProperty

`*pin*  hasProperty(pin, propertyName)`

Returns *t* if the property specified by `propertyName` is attached to `pin`. Otherwise, it returns nil.

isLoopRouted

`*value*  isLoopRouted(pin, byp_cap_prop, *sens_distance*)`

Returns *t* if the loop is routed. Otherwise, it returns nil. A loop is considered to be routed if there is real etch between the power pin and the bypass capacitor pin or between the bypass capacitor pin and the ground pin.

matchObjectByPropertyValue

`*pin*  matchObjectByPropertyValue(pin, *propertyName*, *propertyValue*)`

Returns *t* if `pin` has the specified `propertyName` attached, with the value specified by `propertyValue`. Otherwise, it returns nil.

name

value `name(pin)`

Returns the name of the specified `pin`

Via Predicates

This section describes predicates that you can apply to an object that is a via.

getShapesConnToVia

shape `getShapesConnToVia(`*via*`)`

Returns all shapes physically connected to *via* through thermal-relief clines.

getViaLayer

value `getViaLayer(via)`

Returns layers that are connected by `via`.

name

value `name(via)`

Returns the coordinates of `via`.

Shape Predicates

This section describes predicates that you can apply to an object that is a shape.

getClass

value `getClass(`*shape*`)`

Returns the class of the shape object.

getIntersectAngle

value `getIntersectAngle(`*shape, net*`)`

Returns the list of angles of intersection of the shape with a cline segment.

getNet

net `getNet(`*shape*`)`

Returns the net of which `shape` is a part.

getPolygon

polygon `getPolygon(`*shape*`)`

Returns a polygon representing the geometry of the shape.

getPropertyValue

value `getPropertyValue(`*shape , value*`)`

Returns the `value` of the property of `shape`.

getShapeSegments

shape `getShapeSegments(`*shape*`)`

Returns the list of boundary segments that make up `shape`.

getSubClass

value `getSubClass(`*shape*`)`

Returns the subclass of `shape`.

getUnbypassedRegionSegments

shape `getUnbypassedRegionSegments(`*shape, value, value, value, value*`)`

Returns the boundary segments of an EMC region that do not have enough decoupling capacitors. A decoupling capacitor is identified by having the property EMC_COMP_TYPE = BYPASS_CAP attached.

- The second argument is the sensitive distance.
- The third argument is the maximum separation allowed between decoupling capacitors along the boundary.
- The fourth argument is the bypass capacitor property value (that is, BYPASS_CAP).
- The fifth argument is the critical classes of regions defined (for example, “CLASS1 CLASS2 CLASS3 CLASS4 CLASS5”).

getUnfencedRegionSegments

shape `getUnfencedRegionSegments(`*shape, value, value, value, value*`)`

Returns the boundary segments of an EMC region that are not fenced properly.

- The second argument is the fence-sensitive distance.
- The third argument is the minimum overlap required between a fence and the boundary segment of the room.
- This value is given as a fraction.
- The fourth argument is the fence component type (that is, EMC_COMP_TYPE = FENCE).
- The fifth argument is the critical classes of regions defined (for example, “CLASS1 CLASS2 CLASS3 CLASS4 CLASS5”).

hasProperty

shape `hasProperty(`*shape , value*`)`

Returns `shape` if the object has the property given by *value*. Otherwise, it returns nil.

isCleanGround

shape `isCleanGround(`*shape*`)`

Returns `shape` if the object is a clean ground shape. Otherwise, it returns nil. A clean ground shape is a plane ground shape enclosed by a void.

isEmcRegion

shape `isEmcRegion(`*shape*`)`

Returns `shape` if the object is an EMC region. Otherwise, it returns nil. EMC regions reside in BOARD_GEOMETRY/ [TOP/BOTTOM/BOTH]_ROOM and have segments parallel to the x or y axis of the layout plane.

isPlaneGroundShape

shape `isPlaneGroundShape(`*shape*`)`

Returns `shape` if the object is a plane ground shape. Otherwise, it returns nil. Plane ground shapes lie in ETCH class with a PLANE subclass type. They are part of a net with VOLTAGE = 0.

isRectangle

`*shape*  isRectangle(*shape*)`

Returns `shape` if the object is an Allegro rectangle element. Otherwise, it returns nil.

name

`*value*  name(*shape*)`

Returns the name of the shape.

Polygon Predicates

This section describes predicates that you can apply to an object that is a polygon.

getArea

`*value*  getArea(*polygon*)`

Returns the area of the polygon.

getCenter

`*value*  getCenter(*polygon*)`

Returns the center of the bounding box of the polygon.

getClineSegsInArea

`*net*  getClineSegsInArea(*polygon*)`

Returns all cline segments that lie fully or partially inside the polygon.

getCompInArea

`*component*  getCompInArea(*polygon*)`

Returns all components that lie fully or partially inside the polygon.

getIntersectionPolygon

`*polygon*  getIntersectionPolygon(*polygon* , *polygon*)`

Returns the intersection of the polygons.

getLength

`*value*  getLength(*polygon*)`

Returns the length of the polygon bounding box.

getRectangle

`*polygon*  getRectangle(*value* ,*value*, *value* ,*value* ,*value*)`

Returns a rectangular polygon with the dimensions specified by the arguments.:

- The first argument is x1.
- The second argument is y1.
- The third argument is x2.
- The fourth argument is y2.
- The fifth argument is b.

The `getRectangle` predicate constructs a rectangle whose center line runs from point (x1, y1) to point (x2, y2) and whose width equals b.

getShapesInArea

`*shape*  getShapesInArea(*polygon*,*value*,*value*)`

Returns all shapes in the area defined by the polygon that have a class matching the second argument and a subclass matching the third argument.

getUnionPolygon

`*polygon*  getUnionPolygon(*polygon* , *polygon*)`

Returns the union of the polygons.

name

`*value*  name(*polygon*)`

Returns the name of the polygon as poly:Bbox.

splitPolygon

`*polygon*  splitPolygon(*polygon*)`

Returns a list of four polygons representing the four quadrants of the bounding box of the polygon.

Index