cadence®

# Allegro Design Entry HDL Rules Checker User Guide

**Product Version 23.1**
**September 2023**

Cadence Design Systems, Inc. (Cadence), 2655 Seely Ave., San Jose, CA 95134, USA.

# Contents

# Preface

This preface describes the following:



## About This Guide

This user guide shows you how to use the Allegro Design Entry HDL Rules Checker to check Allegro Design Entry HDL schematics for design rule violations. It discusses various features of Allegro Design Entry HDL Rules Checker in batch mode and in interactive mode.

The user guide explains the necessary concepts and procedures required to use the rules of the Allegro Design Entry HDL Rules Checker (Rules Checker) tool in various environments. It also describes the Advanced Rule Language (ARL) in which the rules are written. This helps you develop your own new rules.

This guide assumes you are familiar with the Allegro Design Entry HDL schematic editor. It also assumes that you are familiar with the user interface of Cadence tools and basic Window NT tools.

## How to Use This Guide

The user guide begins with a brief introduction of Rules Checker 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

- describe all the concepts used in this tool.
- explain the various modes in which Rules Checker is used.
- help you write new rules.

If you are a new user and do not have any prior working experience with Rules Checker, start your learning process with the first chapter and continue exploring the guide in sequence. If you are using the user guide as a reference, you may directly reference any chapter corresponding to a particular topic. For details, see the Brief Outline of Different Chapters section.

# Brief Outline of Different Chapters

This guide is organized into seven chapters.

1. Chapter 1: Introduction to Allegro Design Entry HDL Rules Checker

   This chapter gives a brief introduction to the Allegro Design Entry HDL Rules Checker. It describes the two modes, interactive and batch, in which the tool can be used.

2. Chapter 2: Setting Up Allegro Design Entry HDL Rules Checker

   This chapter describes the Allegro Design Entry HDL Rules Checker environment and explains how you can go about changing default settings.

3. Chapter 3: Customizing Allegro Design Entry HDL Rules Checker

   This chapter discusses how you can customize Allegro Design Entry HDL Rules Checker. You can customize the tool for your entire group (all users who use a particular Rules Checker licence), or you can customize it locally.

4. Chapter 4: Running Allegro Design Entry HDL Rules Checker in Batch Mode

   This chapter describes the various procedures that you can perform in batch mode, and how the `cp.dat` file is to be used to perform various tasks.

5. Chapter 5: Running Allegro Design Entry HDL Rules Checker in Interactive Mode

   This chapter describes the various procedures that you can perform in the interactive mode, i.e., by using the UI of Allegro Design Entry HDL Rules Checker.

6. Chapter 6: Creating Rules

   This chapter discusses how you can create new rules in Allegro Design Entry HDL Rules Checker.

7. Chapter 7: Using Advanced Rule Language (ARL)

   This chapter describes the Advanced Rule Language (ARL), the language in which the rules are written. It discusses various features and constructs of the language you can use to write new rules.

# Typographic and Syntax Conventions

This list describes the syntax conventions used for tools used in the Allegro Design Entry HDL Rules Checker 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.<br>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.<br>*argument*`...: specify at least one argument, but more are possible`<br>`[`*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 Allegro Design Entry HDL Rules Checker

Allegro Design Entry HDL Rules Checker lets you check for violations of design-related rules in a design as well as obtain information about various aspects of your design. It includes fifteen sets of rules and design checks that let you know early in the design cycle if you are violating design rules.

Before expanding your design, you can check rules in the graphical and body environments. These rules check for proper placement of elements on the drawing, consistency between the logic and body drawing, properties and property values, unconnected elements, and invalid names.

After you expand (and for some rules, package) your design, you can check rules in logical and physical environments. These rules check for fan-in and fan-out errors, load errors, required properties and property values, unconnected elements, naming requirements, power requirements, cost requirements, and so on.

The following table briefly describes the four Rules Checker environments:

| Environment | Dependencies | Comments |
|---|---|---|
| Body | | Checks bodies |
| Graphical | | Checks unexpanded designs page by page |
| Logical | expanded design | Automatically expands design when checking rules in Logical environment |
| Physical | packaged design | You must run Packager-XL on the design before checking rules in Physical environment. |

**Note:** The Rules Checker outputs a marker file (`.mkr`) that can be loaded into the Markers tool. This file contains information about objects associated with rule violations found in your

design. This helps you locate and fix violations. The Markers tool can be invoked from the Tools section in Allegro Design Entry HDL.

# Batch and Interactive Modes

You can run the Rules Checker in either batch mode or interactive mode.

### Batch Mode

In batch mode, you run the Rules Checker by specifying the project file on the command line:

```
checkplus -proj <project file>
```

You also provide the Rules Checker with a file (`cp.dat`) in the run directory that includes information on which rules to run on the design. After the run, you can use the message file (`cp.msg`), created in the run directory, to view the listing of messages in your design.

To locate violations in your schematic

➤ Load the marker file (`cp.mkr`) created in the run directory into the Markers tool in Allegro Design Entry HDL.

For more information on the batch mode, see Chapter 4, “Running Allegro Design Entry HDL Rules Checker in Batch Mode”

### Interactive Mode

In interactive mode, you use the Rules Checker graphical user interface (GUI), which you can access from the Board Design flow from the Tools section. It can also be invoked stand-alone by specifying the project file on command line, namely

```
checkplusui -proj <project file>
```

When using the GUI, you choose rules and check the design by performing point-and-click operations.

For more information on the interactive mode, see Chapter 5, “Running Allegro Design Entry HDL Rules Checker in Interactive Mode”

# Rule Files

The Rules Checker checks rules from fifteen default files, located in `<your_install_dir>/tools/checkplus_exp/concept/rules`. Each of these

files is a compiled version of a file defined in the Rules Checker Rule Language. You can view the original source files in `<your_install_dir>/tools/checkplus_exp/concept/rules_source` to gain a better understanding of how the rules work.

For more information on the default rules in the Rules Checker, see Appendix A, “Allegro Design Entry HDL Rules Checker Rules” of *Allegro Design Entry HDL Rules Checker User Guide*.

# 2

# Setting Up Allegro Design Entry HDL Rules Checker

This chapter describes the basic Allegro Design Entry HDL Rules Checker environment and specifies how to change the default settings of the Allegro Design Entry HDL Rules Checker environment, if required. It includes specifying the Allegro Design Entry HDL Rules Checker (Rules Checker) run directory, the initialization file, the rule dependencies, and so on.

**Note:** The Rules Checker settings can be changed only in the interactive mode using the *Allegro Design Entry HDL Rules Checker Setup* dialog box.

## Setting Up the Run Directory

In the Rules Checker you can specify the directory where you want all output files, such as `cp.msg`, `cp.mkr`, and `cp.lst`, to be stored.

**To specify the run directory**

1. Choose *Edit – Setup* to display the Rules Checker Setup dialog box.
2. Select the *Run* tab.
3. Enter the path where you want Rules Checker to store the output files in the *Run Directory* field or use the browse button (...) to choose an existing directory.
4. Click *OK*.

## Specifying the Default .ini File

The first time you run Rules Checker on your design it opens with the default settings. When you exit Rules Checker, it creates an initialization file (`CheckPlus.ini`) that contains the following information:

- Last selected environment

- Selected and deselected rules
- Expanded and unexpanded rules

**To specify the default `.ini` file**

1. Choose *Edit > Setup* to display the Rules Checker SetUp dialog box.
2. Select the *Run* tab.
3. Type the name of the `.ini` file in the *Default Ini File* field.

# Specifying the Type of Pin Direction Check

Rules Checker uses `BIDIRECTIONAL`, `INPUT_LOAD`, `OUTPUT_LOAD` and `OUTPUT_TYPE` properties on pins to determine pin directions.

Rules Checker uses the following table to determine pin directions when *PIN_DIR_CHECK* option is checked.

| Bidirectional | Input-Load | Output-Load | Output-Type | Pin Direction |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | Error |
| 0 | 1 | 0 | 0 | Input |
| 0 | 0 | 1 | 0 | Output |
| 0 | 1 | 1 | 0 | Error |
| 0 | X | X | 1 | Output |
| 1 | X | X | X | InOut |

and the following table when *PIN_DIR_CHECK* option is not checked.

| Bidirectional | Input | Output | Pin Direction |
|---|---|---|---|
| 0 | 0 | 0 | Input |
| 0 | 1 | 0 | Input |
| 1 | X | X | InOut |

| Bidirectional | Input | Output | Pin Direction |
|---|---|---|---|
| 0 | 1 | 1 | InOut |
| 0 | 0 | 1 | Output |

By default Rules Checker assumes *PIN_DIR_CHECK* to be checked.

### To specify the pin direction

1. Choose *Edit > Setup* to display the Rules Checker Setup dialog box.
2. Select the *Run* tab if not already displayed.
3. Select the *Pin Direction Check* option.

   Rules Checker will verify that the correct pin direction has been specified according to the properties in this table. If you do not choose the *Pin Direction Check* option, Rules Checker determines the pin direction according to the properties in this table.
4. Click *OK*.

# Specifying Rule Dependencies

The dependencies between rules decide

- Whether or not the rule needs to be run
- The order in which the rules are run

### When to Use Rule Dependencies

You can specify dependency only among the rules that have been selected for execution in a particular run. If you have any other rule name in your dependency file, it will result in an error.

Name the rule dependency file as `cp.dep` and place it in the Run Directory. This file needs to be created manually.

### To specify rule dependencies

Select the number of rules that you want to run on a particular design.

If you are not concerned about the order in which these rules need to be executed, leave the *dependency* option deselected.

To run these rules in a particular order or to run certain rules only if certain other rules have produced a particular result, switch the *dependency* option on.

**To use rule dependency**

1. Choose *Edit > Setup* to display the Rules Checker Setup dialog box.
2. Select the *Run* tab if not already displayed.
3. Select the *Rule Dependency Option* check box. Rules Checker will assume that `cp.dep` file exists in the Run Directory.
4. Click *OK*.

# Specifying the Rule File/Include File Search Path

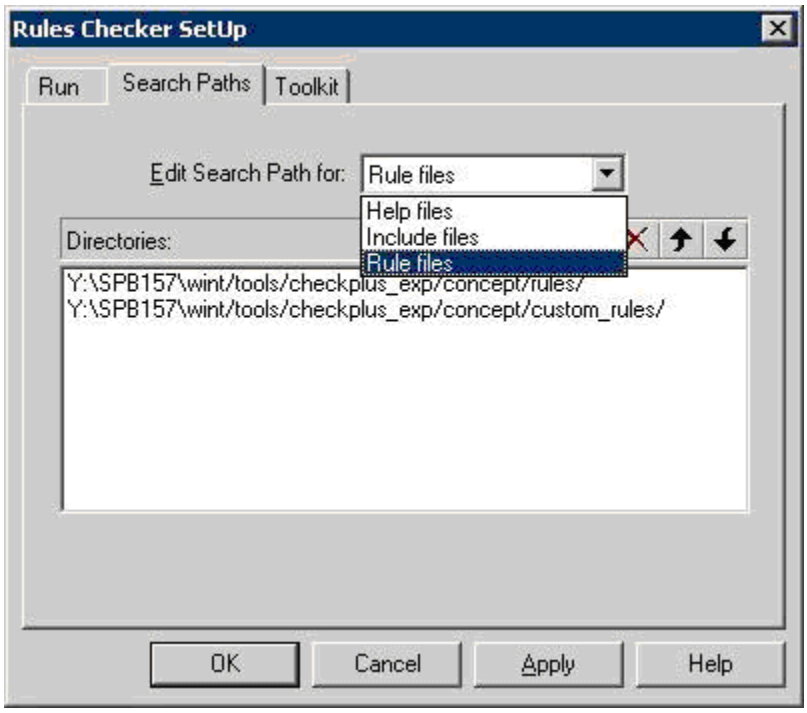
For Rules Checker to check your design, you need to specify the location of the compiled rule or include files.

You must modify the rule search path if you have rule files created with the Rules Checker toolkit located outside the specified search path. Similarly, you must modify the include search path if you have parameter files located outside the specified search path.

To display the Search Paths tab, do the following:

1. Choose *Edit > Setup…* to display the Rules Checker Setup dialog box.

2. Select the *Search Paths* tab. The existing search paths are shown in the *Directories* list box.




You can

- Add search path directories
- Change search path directories
- Delete search path directories
- Move up and move down search path directories

Another way of specifying the locations of the rule files, the include files, and the rule help files is to use the following directives in the `site.cpm` file:

- `RULE_SEARCH_PATH` - Stores the location of the rule files
- `INCLUDEPATH` - Stores the location of the include files
- `HELPPATH` - Stores the location of the rule help files

You set these directives in the `START_CHECKPLUS...END_CHECKPLUS` section of the site.cpm file.

For example to set the path to the rules file, set the following statement:

```
RULE_SEARCH_PATH '$CDS_INST_DIR/
tools\checkplus_exp\concept\custom_rules'
```

You can also have more than one entry for path separated by spaces to specify the order of directories in which you want to search.

Yet another way to specify these paths is to use the environment variable `CDS_SITE`. If this variable is set, Rules Checker looks for custom rules, rule help files and include files in $`CDS_SITE`\custom_rules, $`CDS_SITE`\custom_help and $`CDS_SITE`\custom_rules_include directories.

# Specifying Views

You can choose the configuration and packager views when running Rules Checker in Logical or Physical views.

- **View**, **logical -** This is the expansion type on which Rules Checker runs when the logical environment is selected. The default is the value of the view specified for the Physical Layout expansion type in the *Expansion* tab of the `Project Setup` utility. The packager uses the Physical Layout view when expanding the design.
- **View**, **physical -** The physical view specifies the location of the packager netlist files when the Rules Checker physical environment is selected. The default is the value of the Packaged view specified in the Views tab of the `Project Setup` utility.

### To specify the view

➤ Enter the new view name or choose an existing view from the drop down list box.

# Using Rule Dependency Information

Rules Checker gives rule dependency information about the rules that were executed and those that were not.

### To use rule dependency information

1. Choose *Edit > Setup* to display the *Rules Checker Setup* dialog box.
2. Select the *Run* tab if not already displayed.

3. If the *Rule Dependency Option* check box is selected, click on *Always* or *Never* under *Dump Dependency Information* to determine whether you want to print out the dependency information.
4. Click *OK* to close the dialog box or *Apply* to apply the changes.

If the rule dependency option is selected along with the *Always* option, a file `cp.depinfo` will be created in the Run Directory after completion of the checks. This file gives information on which rules were executed and which were not.

# Specifying Maximum Message Count

As a default, Rules Checker outputs a maximum of 200 messages during a design check. However, you can increase or decrease the number of messages.

**To specify the maximum number of messages output during a design check**

1. Choose *Edit > Setup* to display the Rules Checker Setup dialog box.
2. Select the *Run* tab if not already displayed.
3. Type the desired maximum message count in the *Max Message Count* field.
4. Click *OK*.

# Setting Up Rules Checker Toolkit

The Rules Checker toolkit lets you create your own rules and modify existing rules. Before you use Rules Checker toolkit, you need to setup various options of the toolkit.

**To use the Rules Checker Toolkit**

1. Choose *Edit > Setup* to display the Rules Checker Setup dialog box.
2. Select the *Toolkit* tab.
3. Select the *Edit Rules* check box if you want to use the Rules Checker toolkit to create your own rules or to view the rule source files.
4. Click *OK* to close the dialog box or *Apply* to apply the changes.

# 3

# Customizing Allegro Design Entry HDL Rules Checker

You can customize the following in Allegro Design Entry HDL Rules Checker (Rules Checker):

- Parameters used in rules
- Severity levels of rule violations
- Rules Checker messages

Your system administrator can customize Rules Checker for your entire group, or you can customize it locally. When checking a design, Rules Checker first checks for local customization. If there is no customization at the local level, then Rules Checker applies any global customization that has been made. If no global customization has been made, then Rules Checker uses the Cadence supplied default values for variables and messages.

Although you do not need to do all the customization changes mentioned in this chapter before you use Rules Checker for the first time, you can read this chapter to get an idea of how you and your group can customize Rules Checker.

## Global Customization

Global customization applies to all users who use a particular Rules Checker licence. You can make global changes to variables and messages (including severity as well as the actual message text) for all rules if you are logged in on the account used to install the Cadence software. Usually this is `cdsmgr`.

Global customization lets you carry out the following tasks:

- Changing Parameter Values Used by Many Rules
- Changing Parameters Values for a Single Rule
- Changing Violation Severity Levels
- Editing Rules Checker Messages

This customization will apply to all the users of the corresponding Rules Checker license unless they have performed their own local customization, which will override this global customization. Any changes you make in this new parameter file will override the values specified in the default parameter file.

**Note:** Global customization can only be done manually (in batch mode). It is done in the `custom_rules_include` directory, which is at the same level as `rules_include`.

## Changing Parameter Values Used by Many Rules

To change a macro variable (parameter) used in more than one rule

1. Search the `cp_global.h` file (see the top of the help file) in the `rules_include` directory for the macro and its definition.
2. Copy the

   ```
   STARTGLOBAL
   <Param><macro_name> <macro_value>
   ENDGLOBAL
   ```

   and paste it in the `cp_global.h` file in `custom_rules_include` directory.
3. Edit the `PARAM` statement of the variable you want to change.

## Changing Parameters Values for a Single Rule

To change a parameter used in a single rule

1. Search the appropriate header file (see the top of the help file) in the `rules_include` directory for the macro and its definition.
2. Copy the

   ```
   STARTENV <env-name>
       STARTRULE <rule_name>
       <param> <macro_name> <macro_value>
       ENDRULE
   ENDENV
   ```

   and paste it in the corresponding header file in the `custom_rules_include` directory.
3. Edit the `PARAM` statement of the variable you want to change.

**Note:** The maximum length you can specify for a parameter specified by the PARAM keyword is 4096 characters.

## Changing Violation Severity Levels

To change a severity level

1. Open the help file for the rule whose severity level you want to change.
2. Search the appropriate header file.
3. Copy the SEVERITY *<macro name> <macro value>* from the rule section in the header file in `rules_include` and paste it in the corresponding header file in `custom_rules_include`:

```
STARTENV <env name>
    STARTRULE <rule_name>
    ENDRULE
ENDENV
```

4. Modify the value of the macro.

## Editing Rules Checker Messages

To change a Rules Checker message

1. In Rules Checker (or a text editor, if you cannot run Rules Checker in interactive mode), open the help file for the rule whose message you want to change.

   The error message macros are listed at the bottom of the file.
2. Search the appropriate header file in the `rules_include` directory for the macro and its definition. (See the top of the help file.)
3. Copy the LONG or SHORT *<macro name> <macro value>* from the rule section in the header file in `rules_include` and paste it in the corresponding header file in `custom_rules_include`:

```
STARTENV <env name>
STARTRULE <rule_name>
ENDRULE
ENDENV
```

4. Edit the `LONG` or `SHORT` statement of the macro you want to change.

# Local Customization

Local customization applies to individual Rules Checker users and lets you carry out the following tasks:

- Changing Parameter Values Used by Many Rules
- Changing Parameter Values for a Single Rule
- Changing Violation Severity Levels
- Editing Rules Checker Messages

**Note:** By default, Rules Checker looks for customization information in the `rules_include` and `custom_rules_include` subdirectories of `<your_install_dir>/tools/checkplus_exp/concept`. However, you can set the `include_path` option to another directory that contains the `rules_include` and `custom_rules_include` directories for local customization. You can also override macros by specifying them in a `cp_config.h` file in the `include file` search path or in the `<rundir>`. This lets you customize Rules Checker without the need for special write access.

The `cp_config.h` file is picked up first from the `<rundir>`, and if not found there, then from the other search paths specified in the `.cpm` file. You need to specify the location up to the subdirectory level for the customized `cp_config.h` file.

**Note:** You can do local customization in both batch and interactive modes.

## Changing Parameter Values Used by Many Rules

Rules Checker lets you change rule parameters used by more than one rule.

### Interactive Mode

You can carry out the following tasks:

- Adding a Value to Rule Parameters
- Deleting a Value from the Parameters
- Changing the Value of Parameters

#### *Adding a Value to Rule Parameters*

1. Choose *Edit* > *Global Parameters* to display the Global Parameters dialog box
   –or–
   On the *Rule Browser* in the main *Allegro Design Entry HDL Rules Checker* dialog box, right-click to display a shortcut menu.

2. Choose Global Parameters from the shortcut menu to display the *Global Parameters* dialog box. All parameters associated with that rule appear in the *Parameters* field.
3. Select the parameter for which you want to add a value in the *Name* field. The existing values for that parameter are displayed.
4. Type the new value in the *Value* field at the bottom of the dialog box.
5. Click *Add* to add the new value to the rule parameter.
6. Click *OK*.

### *Deleting a Value from the Parameters*

1. Choose *Edit* > *Global Parameters* to display the *Global Parameters* dialog box
   –or–
   On the *Rule Browser* in the main *Allegro Design Entry HDL Rules Checker* dialog box, right-click to display a shortcut menu.
2. Choose *Global Parameters* to display the *Global Parameters* dialog box. All parameters associated with that rule will appear in the Parameters field.
3. Select the parameter for which you want to delete a value. The existing values for that parameter are displayed.
4. Select the value that you want to delete and click *Delete*. The value is deleted.
5. Click *OK* to close the dialog box.

**Note:** You cannot delete all values of a parameter.

### *Changing the Value of Parameters*

1. Choose *Edit* > *Global Parameters* to display the *Global Parameters* dialog box
   –or–
   On the *Rule Browser* in the main *Allegro Design Entry HDL Rules Checker* dialog box, right-click to display a shortcut menu.
2. Choose *Global Parameters* to display the *Global Parameters* dialog box. All parameters associated with that rule will appear in the *Parameters* field.
3. Select the parameter for which you want to change a value. The existing values for that parameter display.
4. Select the value that you want to change and type the new value in the field at the bottom of the dialog box.

5. Click *Change* to change the value of the parameter.
6. Click *OK* to close the dialog box or *Apply* to apply changes.

### Batch Mode

Rules Checker allows you to change rule parameters used by more than one rule. This parameter is defined in the `cp_global.h` file, so that you don't have to search for the header file.

To change a parameter locally

1. Open the file `cp_global.h` from the `rules_include` directory in a text editor.
2. Copy the PARAM statement of the parameter you want to change and paste it to `cp_config.h` in the <*rundir*> in *STARTGLOBAL ENDGLOBAL* section.
3. Edit the value of the parameter you want to change.

   For example:

   a. Copy the following line from `cp_global_h` to <*rundir*>`/cp_config_h` in the global section.

   ```
   PARAM AC_TECH "54AC", "74AC"
   ```

   a. Change the value to remove "74AC."

The <*rundir*>`/cp_config_h` will have the following lines:

```
STARTGLOBAL
PARAM AC_TECH "54AC"
ENDGLOBAL
```

## Changing Parameter Values for a Single Rule

Rules Checker lets you change rule parameters locally, overriding the preset global parameters.

### Interactive Mode

You can carry out the following tasks:

- Adding Rule Parameter Value for a Single Rule
- Deleting a Value from the Parameter for a Single Rule

■ Changing Value of the Parameter for a Single Rule

### *Adding Rule Parameter Value for a Single Rule*

1. On the Rule Browser in the main Allegro Design Entry HDL Rules Checker dialog box, right-click on the rule (not rule file) whose parameters you want to edit. A shortcut menu is displayed.
2. Choose *Rule Parameters* from the shortcut menu to display the Rule Parameters dialog box. All parameters associated with that rule appear in the Parameters field.
3. Select the parameter for which you want to add a value.
4. Click *Add* and add new values in the empty field next to the *Add* button.
5. Click *OK*.

### *Deleting a Value from the Parameter for a Single Rule*

1. On the Rule Browser in the main *Allegro Design Entry HDL Rules Checker* dialog box, right-click to display a shortcut menu.
2. Choose *Rule Parameters* from the shortcut menu to display the *Rule Parameters* dialog box. All parameters associated with that rule appear in the Parameters field.
3. Select the parameter for which you want to delete a value
4. Select the value that you want to delete. This value appears in the empty field.
5. Click the *Delete* button to delete this value.
6. Click *OK*.

### *Changing Value of the Parameter for a Single Rule*

1. On the Rule Browser in the main *Allegro Design Entry HDL Rules Checker* dialog box, right-click to display a shortcut menu.
2. Choose *Rule Parameters* from the shortcut menu to display the *Rule Parameters* dialog box. All parameters associated with that rule appear in the Parameters field.
3. Select the parameter for which you want to change the value.
4. Select the value that you want to change. This value appears in the empty field.
5. Type the new value and click *Modify*.

6. Click *OK*.

### Batch Mode

1. In Rules Checker Batch (or a text editor, if you cannot run Rules Checker in interactive mode), open the help file for the rule whose variable(s) you want to change.
2. Search the appropriate header file (see top of the help file) in the `rules_include` directory.
3. Copy the

```
STARTENV <env-name>
    STARTRULE <rule-name><macro-name><macro-value>
    ENDRULE
ENDENV
```

   from the `include_file`, and paste it to `cp_config.h` in the <*rundir*>.
4. Edit the parameter value in `cp_config._h`.

#### *Example*

The `prop_name_length_check` rule in `graphic_property_checks.rle` checks for properties with names longer than the value specified in the `GRAPHICAL_MAX_NAME_LENGTH` macro. By default, it checks for names longer than 16 characters.

To check for property names longer than five characters

➤ Change the following macro:

```
GRAPHICAL_MAX_NAME_LENGTH 16
```

to

```
GRAPHICAL_MAX_NAME_LENGTH 5
```

`cp_config.h` file will have the following lines:

```
STARTENV GRAPHICAL
STARTRULE prop_name_length_check
PARAM GRAPHICAL_MAX_NAME_LENGTH 5
ENDRULE
ENDENV
```

Changing this macro in a header file in `<your_install_dir>/tools/checkplus_exp/concept/custom_rules_include` results in global customization.

Changing the macro in the `<rundir>/cp_config.h` file results in local customization.

## Changing Violation Severity Levels

Every rule included with Rules Checker has a default severity level (`Fatal`, `Error`, `Warning`, `Oversight`, or `Info`). The severity is defined in the `<rulefilename>.h` file.

**Note:** You can use only the following severity levels: `Fatal`, `Error`, `Warning`, `Oversight`, or `Info`. Rules Checker stops checking when it encounters a fatal error.

### Interactive Mode

1. On the Rule Browser in the main Allegro Design Entry HDL Rules Checker dialog box, right-click on the rule (not rule file). A shortcut menu is displayed.
2. Choose *Rule Parameters* from the shortcut menu to display the Rule Parameters dialog box.
3. Select the *Messages* tab.
4. Select the severity parameter whose value you want to change. Its current severity appears in the Values field.
5. Use the pull-down menu to choose the required value.
6. Click *OK*.

### Batch Mode

The following instructions show you how to change the severity of a rule using the batch mode.

To change violation severity levels in batch mode

1. Open the help file for the rule whose severity level you want to change.
2. Search the appropriate header file.
3. Copy the severity `<macro name> <macro value>` from the rule section in the header file in `rule_include`.

4. Modify the value of the macro.
5. Save the rule in <*rundir*>/cp_config.h under

```
STARTENV <env name>
STARTRULE <rule_name>
ENDRULE
ENDENV
```

### *Example*

To change the default severity level of the `inputio_check` rule from `Warning` to `Error`

➤ Change the following macro:

```
SEVERITY INPUTIO_CHECK_SEVERITY Warning
```

so that it is defined in the following section in <*rundir*>/cp_config.h as:

```
STARTENV LOGICAL
STARTRULE INPUTIO_CHECK
SEVERITY INPUTIO_CHECK_SEVERITY Error
ENDRULE
ENDENV
```

## Editing Rules Checker Messages

When checking a design, Rules Checker produces messages that give you information about each rule violation it detects. Each rule has a short message and along message associated with it. To supply additional information, you can edit these default long and short messages.

The help file for each rule lists both the long and short message macros associated with it. Short messages describe the violation and long messages describe the violation and use variables (of the form `?name`) to provide the names of elements associated with the violation. When checking a design, Rules Checker outputs long messages to the `cp.msg` file in the <*rundir*>.

The message is defined in the <*rulefilename*>.h file.

Use the following guidelines when editing Rules Checker messages:

- You cannot change the names of variables.
- You cannot create your own variables.

  You must use the variables provided.

- You can add variables to short messages.

  If you add a variable, it must be a legal value from the long message.

- Use C formatting characters (for example, `\n` for newline, `\t` for tab).

  You cannot use the *Return* or *Tab* keys when editing macros.

- Make sure your message is enclosed in double quotes (").

- End each long message with a newline character (`\n`).

### Interactive Mode

1. On the Rule Browser in the main Allegro Design Entry HDL Rules Checker dialog box, right-click to display a shortcut menu.
2. Choose *Rule Parameters* to display the Rule Parameters dialog box. All parameters associated with that rule appear in the Parameters field.
3. Select the *Messages* tab.
4. Select the parameter for which you want to change the message.
5. Type the new short, long or both messages. Click *Apply* to apply the changes.
6. Click *OK* to close the dialog box.

### Batch Mode

1. In Rules Checker (or a text editor, if you cannot run Rules Checker in interactive mode), open the help file for the rule whose message you want to change.
2. Search the appropriate header file (see the top of the help file) in the `rules_include` directory for the macro and its definition.
3. Copy the message macro and paste it in your <*rundir*>`/cp_config.h` file.
4. Edit the LONG or SHORT statement of the macro you want to change.

### *Example*

To change the long message for the `inputio_check` rule, you might change the following macro file in the <*rundir*>`/cp_config.h` header file for local customization:

```
LONG STD_ERR_INPUTIO_CHECK_1 "(#132): "No input on net",
        "\tError in both states",
"\tLogical net name:?sig1\n"
```

so that it is defined as:

```
LONG STD_ERR_INPUTIO_CHECK_1 "(#132): "No input on net ?sig1",
"\tError in both states\n
```

# 4

# Running Allegro Design Entry HDL Rules Checker in Batch Mode

The first step before running Rules Checker in batch mode is to create a `cp.dat` file.

This section describes how to create your `cp.dat` file manually. However, the easiest way to create your `cp.dat` file is to set the desired parameters for your design in Rules Checker interactive mode and use the resulting file.

## Creating the cp.dat File

The `cp.dat` file specifies the following information:

- Your design name (optional)
- Your environment
- Rules you want to check

Before you create the `cp.dat` file, note the following:

- If you have run Rules Checker in batch mode or interactive mode in the run directory, you already have a `cp.dat` file in your <*rundir*> directory.

  You can modify this file instead of creating a new one.
- You must place the `cp.dat` file in your <*rundir*> directory

## Specifying Rule Dependencies

The dependencies between rules decide the following:

- Whether or not the rule needs to be run
- The order in which the rules are run

## When to Use Rule Dependencies

You can specify dependency only among the rules which have been selected for execution in a particular run. If you have any other rule name in your dependency file, it will result in an error.

The rule dependency file is `cp.dep` and is located in the <*rundir*>.

### To specify rule dependencies

➤ Choose the number of rules that you want to run on a particular design.

- ❑ If you are not concerned about the order in which these rules are executed, leave the dependency option deselected

–or–

- ❑ To run these rules in a particular order or to run certain rules only if certain other rules have produced a particular result, switch the dependency option on

Use the dependency option for the following:

- ■ Controlling the order in which the selected rules are run
- ■ Running certain rules depending on the result obtained by running certain other rules

## Controlling the Order for Running Selected Rules

When you switch the dependency toggle on and do not specify a dependency file, the rules are run in the order in which they are specified on the command line. In batch mode you control the order in which rule files are specified on the command line.

For example, consider the following scenario:

Rule1 analyzes the design and creates a file containing information from that analysis. Rule2 reads the file created by rule1 and performs other checks on the design. Rule2 should only be run after rule1. If you place both rules in one file, rule1 first and rule2 second, and turn on the dependency option, the rules are run in the correct order. If the dependency option is not on, Rules Checker can run rule1 and rule2 in any order it finds most efficient.

You can also use the dependency file to explicitly specify an order. For example, in the scenario described previously, you place the following dependency in a file:

```
rule2: all(msg(rule1) or not(msg(rule1)))
```

Execute rule2 only after rule1 has been run for all instances of its basic object, whether or not a message has been given.

## Running Rules Dependent on Results from Other Rules

In some cases you would have to run certain rules only if a certain result has been obtained from running other rules.

For example, assume you use the rule `sig_prop_exists`, which checks for the existence of a property. You use another rule, `sig_prop_range_check`, which assumes that the property does exist and carries out a range check on the value of the property. These checks are dependent in that `sig_prop_range_check` checks only for those signals for which `sig_prop_exists` has passed, that is it has not given an error message. You can specify that the rules be run in this way by including the following in the dependency file:

```
sig_prop_range_check: not(msg(sig_prop_exists))
```

You might also have two or more rule sets, each one checking different types of problems in the design. To save rule execution time, run rules of category 2 only if the rules of category 1 have been successfully run.

For example, consider the rule `loading_check`. One of the prerequisites for `loading_check` is that `INPUT_LOAD` and `OUTPUT_LOAD` properties be specified on all pins in the design. You may want to run `loading_check` only if `input_pin_prop_exists`, `output_pin_prop_exists`, and `pin_prop_range_check` have passed for all the pins in the design. To achieve this, add the following to your dependency file:

```
loading_check: all(not(msg(input_pin_prop_exists)) and
not(msg(output_pin_prop_exists)) and not(msg(pin_prop_range_check)))
```

You might also use the rule dependency option to build a cause/effect relationship among various rules. For example, if you use rule1, which iterates over all the wires in a design and produces a list of wires that does not have some basic properties, rule1 puts this list in a file using file predicates. Assume there is a script which corrects the design by fixing all the wires listed in the file. When you write another rule, rule2 calls the script, using the toolStart predicate. Here rule1 and rule2 are related to each other in that rule2 runs if rule1 fails (as in rule1 gives one or more messages). To achieve this, you put the following in your dependency file:

```
rule2: msg(rule1)
```

## Syntax of the Rule Dependency Expression (RHS)

The syntax of the dependency file is

```
<RuleDep> ::= <RuleName> : <DependencyExpr>
     | <DependencyExpr> or <DependencyExpr>
     | not ( <DependencyExpr> )
     | ( <DependencyExpr> )
     | all ( <DependencyExpr> )
     | any ( <DependencyExpr> )
     | Msg ( <RuleName> )
     | Run ( <RuleName> )
```

Assume the following examples appear in the rule dependency file:

1. `rule1: all(msg(rule2))`

   This statement says to run rule1 only if rule2 produces a message in each of its iterations over its basic object. The operator `all` signifies that the result of rule2 should be checked for all its iterations.

2. `rule1: any(msg(rule2))`

   Execute rule1 only if rule2 produces a message in any of its iterations over its basic object. The operator `any` signifies that the result of rule2 should be checked for all its iterations and must get a message from any one of the iterations to run rule1.

   In examples 1 and 2, we say that rule1 has complete rule dependency over rule2 because rule1 is not executed unless all iterations of rule2 have been completed; that is, rule2 has been run for all instances of its basic object.

3. `rule1: msg(rule2)`

Note that there is no operator `all` or `any`. To run rule1, checking the result of rule2 for all the iterations is not necessary. Here the assumption is that both rule1 and rule2 have the same basic objects, for example, wire, so they have the same number of iterations to run once for each wire. In this case the decision is made on a per wire basis. For every wire, rule2 is run first, and then rule1 is run for the same wire only if rule2 has given a message for that wire.

The dependency in example 3 is called `per instance dependency` because the decision on whether or not to run a rule is taken separately for every instance of its basic object.

**Note:** Two rules can be linked with per instance dependency only if they have the same basic object.

For complete rule dependency there is no such restriction; two rules can be linked with complete rule dependency irrespective of their basic objects.

1. `rule1: all(msg(rule2) or msg(rule3))`

   Here rule1 has complete rule dependency on both rule2 and rule3. Rule2 and rule3 are assumed to have the same basic object, for example, pin. So for each pin rule2 and rule3 will be executed. And for every such iteration of rule2 and rule3, the results will be recorded. Now rule1 will be run only if either rule2 or rule3 has given a message for every pin.

2. `rule1: all(msg(rule2)) or all(msg(rule3))`

   In this example, the condition for running rule1 is that either rule2 has given a message for all the pins in the design or rule3 has given a message for all the pins in the design.

3. `rule1: all(msg(rule2)) or msg(rule3)`

Here rule1 has complete rule dependency on rule2 and per instance dependency on rule3. Rule1 and rule3 are assumed to have the same basic objects, for example, signal. Rule2 may have a different or same basic object, for example, pin. For any signal, the condition for running rule1 is that either rule2 has given a message in all its iterations, that is for all the pins, or rule3 has given a message for a signal. If rule2 has not given a message for some of the pins, then rule1 will not be run for those signals for which rule3 gave no message.

# Selecting the Environment

You must specify an environment before specifying rules in the `cp.dat` file. Environment files are located in `<your_install_dir>/tools/checkplus_exp/concept/env`.

The following table lists the rule files associated with each environment

| Env Name | Environment File | Rule File |
|---|---|---|
| **Body** | precompiled_body.rle | body_cross_view_checks.rle<br>body_drawing_checks.rle<br>body_pin_checks.rle<br>body_property_checks.rle<br>**Note:** Body rules only work on a component library or individual components. For checking schematics, rule selection should only be restricted to graphical, logical, or physical rules. To run physical rules, the design must first be packaged |
| **Graphical** | precompiled_logic.rle | graphic_drawing_checks.rle<br>graphic_property_checks.rle<br>graphic_section_checks.rle<br>graphic_connectivity_checks.rle |
| **Logical** | logical_env.rle | property_checks.rle<br>net_name_checks.rle<br>loading_io_checks.rle<br>electrical_check.rle |
| **Physical** | physical_env.rle | design_guidelines.rle<br>preferred_parts.rle<br>jedec.rle<br>**Note:** If you specify physical_env.rle, you must run Packager-XL on your design before you check i |

### To choose an environment

➤ Enter the following before the rule listing in your `cp.dat` file.

`-r <your_install_dir>/tools/checkplus_exp/env/<env file>`

You can include environment variables in the path name.

#### *Example*

To specify the logical environment

➤ Add the following line to your `cp.dat` file:

`-r <your_install_dir>/tools/checkplus_exp/concept/env/logical_env.rle.`

# Specifying a Design

Specifying your design name in the `cp.dat` file is optional. If you do not specify a name, Rules Checker picks up the design name from the project file.

### To specify your design name in the `cp.dat` file

➤ Insert the name of your design as the first line of the `cp.dat` file.

**Note:** This must be the first line in the `cp.dat` file.

#### *Example*

If your design is named `sample` and is in library `test`, add the following line to the beginning of your `cp.dat` file:

```
test.sample
```

#### *Example*

If the design name consists of more than one word, enclose it in double quotes (" "). For example, if your design is named `risc cpu`, add the following line to the beginning of your `cp.dat` file.

```
"<libname>.risc cpu"
```

# Selecting Rules

When you use Rules Checker in batch mode, you must specify the name of the rule or rule files you want to check. In the `cp.dat` file, Rules Checker lets you

- Select a rule or a subset of rules from a compiled rule file
- Select all rules in a compiled rule file

For a description of available rules, see Appendix A, “Allegro Design Entry HDL Rules Checker Rules” of *Allegro Design Entry HDL Rules Checker User Guide*. Rule files are located in <*your_install_dir*>/tools/checkplus_exp/concept/rules.

## Selecting Rules Within a Rule File

To choose an individual rule or subset of rules within a rule file

➤ Enter the following in your `cp.dat` file:

```
-r <path_name>/rule_file rule_name [ rule_name ... ]
```

You can include environment variables in the path name.

### *Example*

To check your design for violations of the `nets_shorted` rule, which is in the `property_checks.rle` file

➤ Add the following as one line to your `cp.dat` file:

```
-r <your_install_dir>/tools/checkplus_exp/concept/rules/property_checks.rle
nets_shorted
```

### *Example*

To count the number of nets (`count_sig` rule) and the number of instances (`count_inst` rule) in your design

➤ Add the following as one line to your `cp.dat` file:

```
-r <your_install_dir>/tools/checkplus_exp/concept/rules/property_checks.rle
count_sig count_inst
```

## Enabling all Rules in a Rule File

To select all rules in a compiled rule file

➤ Enter the following in your `cp.dat` file:

```
-r <path_name>/rule_file
```

You can include environment variables in the path name.

### *Example*

To check your design for all rules in the `property_checks.rle` file

➤ Add the following line to your `cp.dat` file:

```
-r <your_install_dir>/tools/checkplus_exp/concept/rules/property_checks.rle
```

## Specifying Options

The following options can be specified in the command line or in the `cp.dat` file:

| | |
|---|---|
| `-dep` | Switches on the dependency. |
| `-depinfo` | Dumps the dependency info into the `cp.depinfo` file. |
| `-d <filename>` | Specifies the dependency files. You can use the `-d` option more than once to specify multiple dependency files. |
| `-max_message <number>` | Specifies the maximum number of messages to be reported in a single run of Rules Checker. |
| `-I <path>` | Specifies an include path. If the option is not specified, then rundir and installation default include paths are included. |
| `-e` | Specifies compilation mode of Rules Checker. |

# Checking Your Design

After you have created your `cp.dat` file, you can run Rules Checker to check your design.

To check your design

1. `cd` to the directory containing the `cpm` file.
2. Enter the following at the command line:

   `checkplus -proj <prj file>[ -r rule_file [rule_name]...]`

| | |
|---|---|
| `-proj <proj file>` | Specifies the project file. |
| `-r rule_file [rule_name]` | Lets you specify a compiled rule (or rules) not included in your `cp.dat` file. |

Rules Checker checks your design and displays messages in your shell window.

Rules Checker displays message about errors it encounters in your design. If Rules Checker cannot locate your design in `cp.dat` or a project file, a message is displayed in the shell window where you started Rules Checker. If it cannot run checks when checking in the Logical or Physical environments, a `cp.lst` file is created containing error messages.

When Rules Checker is finished checking your design, the following message appears in the shell window:

```
************************************
*      Stop Run: <Date_and_Time>     *
************************************
```

In addition, Rules Checker creates the following files in the `<rundir>` directory:

- `cp.mkr` contains detailed information about each violation in your design.
- `cp.log` contains the messages displayed in the shell window during your Rules Checker run.
- `cp.lst` contains output if Rules Checker failed to expand the design in Logical and Physical environments.
- `p.msg` contains the long messages for violations encountered in your Rules Checker run
- `cp.verbose` contains debugging information for the rules checked on your design when the `verbose` option is selected.
- `cp.depinfo` provides information about the execution status of all the selected rules when the `depinfo` option is selected.

# Viewing Your Results

After Rules Checker has checked your design, you can view the results in the following ways:

- By highlighting the violations in your Allegro Design Entry HDL schematic

  **Note:** Load the `.mkr` file in Markers in Allegro Design Entry HDL. Markers can be invoked from the Tools section in Allegro Design Entry HDL.

- By reading the `cp.msg` file for information about violations in your schematic.

# 5

# Running Allegro Design Entry HDL Rules Checker in Interactive Mode

You can run Allegro Design Entry HDL Rules Checker (Rules Checker) on your design in the interactive mode. In this mode, you can run various rules on the design using the Rules Checker UI. You can select the body rules, graphical rules, logical rules, and physical rules from the various Rules files and execute them on your design.

## Loading an Initialization File

The first time you run Rules Checker on your design, it opens with the default settings. When you exit Rules Checker, it creates an initialization file (`CheckPlus.ini` in the project file directory*)* that contains the following information:

- Last selected environment
- Selected and deselected rules
- Expanded and unexpanded rules

The next time you start Rules Checker in the same directory, it reads this file by default, so that it displays the same parameters as when you last exited from it.

Instead of using `CheckPlus.ini`, which Rules Checker loads by default from the project file directory, you can read the parameters from an initialization file you have created in an earlier session.

**To load a previously saved initialization file**

1. Choose *File – Open* to display the File Open dialog box.
2. Select the initialization file.
3. Click *OK* or press *Enter* to load the `.ini` file.

# Specifying Rule Dependencies

To use rule dependencies perform the following steps:

1. Choose *Edit – Setup* to display the Rules Checker Setup dialog box.
2. Select the *Run* tab if not already displayed.
3. Select the *Rule Dependency* option. Rules Checker will assume that `cp.dep` file exists in the Run Directory.
4. Click *OK*.

# Selecting the Environment

Before you specify which rules you want to check, you must select an environment. You can view and check rules that correspond only to the selected environment. For example, you cannot view the `non_preferred_part` rule when the Logical environment is selected because it is a rule in the Physical environment.

If you specify the Physical environment, you must run Packager-XL on your design before you check your design.

### To select an environment

➤ Choose the appropriate environment tab from the *Allegro Design Entry HDL Rules Checker* main window. The rule files for the selected environment are displayed.

# Specifying a Design

Before you run Rules Checker on your design, you must specify a design to check.

### To specify the name of the design you want to check

➤ Enter the design name in the Design field on the main *Allegro Design Entry HDL Rules Checker* dialog box.
–or–
Use the *Browse* button and choose the design from the pull-down menu.

# Selecting Rules

After you have specified the design and environment, you need to choose the rules you want Rules Checker to consider when it checks your design. To do this, you choose the rules from the rule browser.

## Displaying Rule Names

The file names that appear in the rule browser on the main Allegro Design Entry HDL Rules Checker dialog box are compiled rule files consisting of individual rules. Use *Edit > Update* to make sure the Browser window displays all the rules that are specified in your search path.

### To display the names of the rules in a rule file

➤ Click on the “+” browser button associated with the file. This button becomes “-”, and the rules in the file are displayed. If necessary, use the scroll bars to view the contents of the Rule Browser window.

### To return the file to its normal view

➤ Click on the “-” associated with the rule file. The rule names disappear and the browser button becomes “+”.

You can now

- Select/deselect all rules in a rule file
- Select/deselect individual rules
- Select/deselect all available rules

# Using Rules in Rules Checker Environments

The rules in the Body and Graphical environments check designs before they are expanded, unlike rules in the Logical and Physical environments, which expand the design during rule checking. This lets you specify which pages, bodies, and body versions to check.

- Body Environment
- Graphical Environment
- Logical Environment

- Physical Environment

# Checking Your Design

After you have selected the necessary parameters—design name, environment, and rules—you can check your design.

You can

- Start the Rule Checker
- Stop the Rule Checker

# Viewing Your Results

After Rules Checker has checked your design, you can view the results in the following ways:

- Highlighting the objects associated with each violation in your Allegro Design Entry HDL schematic.
- Reading the `cp.lst`, `cp.msg`, `cp.verbose`, `cp.log`, and `cp.depinfo` files to get more information about the Rules Checker run. (`cp.lst` only applies to the Logical and Physical environments.)

You can

- View `cp.lst` files
- Read the `cp.msg` file

# Saving an Initialization File

After you have checked your design using Rules Checker, you can save the following parameters to the initialization file:

- Environment
- Selected and deselected rules

You can then load the initialization file to use these same settings on a subsequent run.

By default the `CheckPlus.ini` file is saved in the Run Directory specified in the project file.

**To save a new initialization file**

1. Choose *File – Save As* to display the Save As dialog box.
2. Specify a name and location for the file.
3. Click *OK* to save the file.

**To save an active initialization file**

Choose *File – Save*. This updates the current initialization file. You do not need to do this, if you are using the `CheckPlus.ini` file; Rules Checker automatically updates it on quitting.

# Exiting Rules Checker

Choose *File – Exit*. The Allegro Design Entry HDL Rules Checker window closes.

In addition, Rules Checker creates or updates the `.ini` file in your project file directory. It has information about the rule and environment settings on the Rules Checker. Rules Checker uses these settings the next time you open Rules Checker in the same directory.

# 6

# Creating Rules

Rules Checker lets you create your own rules by editing existing rules or creating new rules.

## Creating (Editing) a Rule

To create your rules

1. Open the rule file in which you want to create a new rule in a text editor or use the Rules Checker UI as

   a. Choose *Edit – Rules* from the main *Allegro Design Entry HDL Rules Checker* dialog box. The Rules Checker Toolkit dialog box appears.

   b. Select the rule file that you want to edit and click *Edit...* Rules Checker loads the rule file into the text editor.

2. Write the new rule and save the file for compilation.

**Note:** While creating new rule files, include `cp_config.h` as the first include file, if parameter customization is required. After all include file directives, add the `use <env_name>` directive, where `env_name` is the environment for which the rule file is intended.

## Compiling a Rule File

After you have created or edited the rule file, you need to compile it. Compilation checks the rule file for errors and creates a compiled form `(.rle)` of the rule file in the compiled file directory.

To compile the rule file in batch mode

➤ Create the `cp.dat` file with the `-e` option in it, the environment file for the environment, and the rule file to be compiled.

Rules Checker compiles the file, and creates a file with an `.rle` extension. Rules Checker places the file in the compiled file directory which can be specified using `compiledfiledir <dir_name>` option in the `cp.dat` file. Default is `./checkplus`.

For example, if you want to compile a file called `myfile.erl` in the logical environment, the `cp.dat` will have the following lines:

```
-e
-r <your_install_dir>/tools/checkplus_exp/concept/env/logical_env.rle
-r <path> myfile.erl
```

This will create a compiled rule file called `myfile.rle` in./`checkplus` by default or in the directory specified with *compiledfiledir* option in `cp.dat`.

A message appears if the rule is successfully compiled. Otherwise an error message is displayed.

To compile the rule file in interactive mode, exit the editor and carry out the following steps:

1. Choose *Edit – Rules* in the main Allegro Design Entry HDL Rules Checker dialog box to display the Rules Checker Toolkit dialog box.
2. Select the file you want to compile.
3. Click *Compile* in the Rules Checker Toolkit dialog box.

Rules Checker Toolkit compiles the file, and creates a file with an `.rle` extension. Rules Checker Toolkit places the file in the compiled file directory specified in the *Toolkit* Tab (from *Edit – Setup*). You can choose to place the compiled files in the same directory as the source file, or in a directory of your choice. Default is ./`CheckPlus`.

A message appears in the Output window if the rule is successfully compiled. Else, an error message is displayed in the Output window.

**Note:** You must choose the environment in which the rule is to be compiled as the current environment.

# Debugging the Rule File

The toolkit generates debugging information, which you can control, as the design is checked against the rule and writes it to a file called `cp.verbose` in the <*rundir*> directory. This will help you ensure that the rule functions as intended.

- To control the type of debugging output from the Allegro Design Entry HDL Rules Checker dialog box in the interactive mode, choose *Edit — Setup — Toolkit — Dump Debug Information*.

**Note:** Specify the Always, If True, and If NULL options on a limited number of rule files, as the `cp.verbose` file can quickly become quite large.

- To control the type of debugging output when running Rules Checker Toolkit in the batch mode, do the following:

  Specify the following option in `cp.dat`, when you run Rules Checker Toolkit in batch mode:

  ```
  -verbose true | null | all
  ```

  - `All` produces information for each condition in the rule as it is evaluated.
  - `true` produces information for each condition in the rule that is evaluated TRUE.
  - `null` produces information for each condition in the rule that is evaluated NULL.

**Note:** You should specify the `Always`, `If True`, and `If NULL` options on a limited number of rule files, as the `cp.verbose` file can quickly become quite large.

After you have debugged the rule, you can place the file in the <*your_install_dir*>/tools/checkplus_exp/concept/custom_rules directory. Be sure to specify this directory in the Search Path field in Rules Checker. You can save the source files for your rules in the <*your_install_dir*>/tools/checkplus_exp/concept/custom_rules_source directory.

# Creating a Help File for the Rule

After you create a rule, it is a good idea to create a help file for it. The help file can contain any information you or your group consider important and appropriate to describe the function of the rule.

To create a help file:

1. Use a text editor to create a file that describes the rule.
2. Name the file <*rulename_prefix*>.<environment>.hlp where environment is one of `body`, `logical`, `physical`, or `graphical`.

   For example, for a rule named `myrule` in the graphical environment, create a file named `myrule.graphical.hlp`.
3. Place the help file in <*your_install_dir*>/tools/checkplus_exp/concept/custom_help

# 7

# Using Advanced Rule Language (ARL)

## Introduction to Rules Checker ARL

As an introduction to the language, this topic includes several basic rules that demonstrate the fundamental concepts behind Rules Checker ARL (Advanced Rule Language).

Every rule file has to be associated with an environment. The environment can be specified by the `use` directive in the rule file. For example `use logical` for logical environment.

This topic describes the following features of ARL:

- Basic Language Constructs
- Variables and Base Objects
- Base Objects and Implied Looping
- Function Calls
- Variable Type
- Assignment Operator and Comparison Operator
- If Construct
- Conditional Operators

### Basic Language Constructs

Several parts construct a rule. The keywords `RuleDefine` and `EndRuleDefine` show the start and end of the rule declaration area of a text file. The keywords `Rule` and `EndRule` specify the logical breakup of separate rules within the `RuleDefine` and `EndRuleDefine` keywords. Every rule must be enclosed by its own `Rule` and `EndRule` set of keywords, unlike the `RuleDefine` and `EndRuleDefine` keywords, which are only required once per text file, but allowed as often as once per rule. Rule names can consist of alphanumeric characters, the underscore and the $ sign; however, the first character in a rule name cannot be a number.

**Note:** The language is case insensitive, so `ruledefine` and `RuleDefine` keywords are equivalent. In general, the syntax used throughout this tutorial mixes uppercase and lowercase letters to clarify the intended purpose of the function.

Inside the rule declaration constructs are the remaining two sections of the rule:

- rule conditional statements
- message statement

The conditional statements determine whether the rule reports the information in the Message statement.

In the following example, the conditional section consists of only one line, `Sig`.Later in this section, the conditional section is a logical grouping of statements that are evaluated to `True` or `Null` according to the functions called and the data supplied to the functions, if the last conditional statement valuates to True, the message will be displayed, else not.

### *Example*

Consider a simple rule that lists all the signals in a design individually. Such a rule uses the following algorithm:

for each signal in the design
print the name of the signal
endfor

In ARL it is coded as

```
Use logical;
RuleDefine/* start of rule declaration */
    Rule print_sig  /* start of rule <rulename> */
        sig1                    /* condition statement */
        Message(Info,sig1,"?sig1");/* msg statement */
    EndRule/* end of rule */
EndRuleDefine/* end of rule declaration */
```

## Variables and Base Objects

Every Rules Checker rule contains exactly one base object. A base object of a rule is a variable that cannot derive its value from any other variable in the conditional section of the rule.

Example

```
RuleDefine
        Rule base_obj
        inst1 := inst(design1) AND  /* inst1 is derived  from design1 */
```

```
        sig1 := sig(design1) /* sig1 is derived  from design1 */
            Message(Info,design1,"?design1");
        EndRule
EndRuleDefine
```

In this example `design1` cannot derive its value from any other variable. Therefore the base object of the rule is `design1`.

Example

```
RuleDefine
    Rule base_obj2
     inst1 := inst(sig1) AND /* inst1  derived from sig1 */
     inst2 := hasProperty(inst1,"COMP_TYPE") /* inst2 is derived from inst1 */
      Message(Info,inst2,"Instances having prop COMP_TYPE are ?inst2");
    EndRule
EndRuleDefine
```

In this example, the base object is `sig1` because `inst1` is derived from it and `inst2` is derived from `inst1`.

The base object also determines the number of times the rule is executed on the existing design. The various base objects are explained later in chapter. However, the rules in this section use the Rules Checker logical environment, which contains six different base objects.

The Signal object is one of these objects. Because Signal is the base object in this example, Rules Checker will run this rule once per each signal in the design. As a result, the maximum number of error messages generated by the rule is equal to the total number of signals in the design. For example, if a design has ten signals then this rule will output at most ten messages.

## Base Objects and Implied Looping

Base Objects constitute one main area of implied looping. In this rule, the engine has not been explicitly instructed to iterate over all signals. However, by specifying Signal as the base object, Rules Checker understands that it must evaluate the rule for every signal in the design. To understand the implied looping execution, you might consider the following pseudo-code:

```
sig1 := First signal in the design
While (!end of signal list) do
{
        Execute the rule
        print message if condition evaluates "True"
        sig1 := Next signal in design
}
```

In the above example, the number of messages printed by the rule are between zero and the number of signals in the design. Later in this chapter, you will learn the rationale behind selecting a base object.

Example

This example further demonstrates the idea of Base Object looping. It uses the following algorithm to list all signals of the design in one message:

```
foreach design
        print all signals in design
endfor


RuleDefine
    Rule print_sig1
        sig1 := sig(design1)
        Message(Info,sig1,"?sig1");
    EndRule
EndRuleDefine
```

In this example the base object is `Design`. When executing a rule in the Rules Checker logical environment, only one item can be assigned to the “Design” object. This item is the name of the Design that was requested as the target of the Rules Checker run. Since there is only one value for the base object, this rule will be executed only one time.

## Function Calls

A function call in the Rules Checker language is similar to a function call in other languages: it takes one or more parameters and returns a value. Before writing or understanding rules in a specific environment, it is necessary to have a list of the predefined functions available for that environment.

**Note:** Currently the Rules Checker Toolkit does not support user-created functions. You must use the functions supplied in the Toolkit.

The rule in the previous example calls the function `Sig` and passes it the base object variable, `Design`. By looking up the definition of the `Sig()` function you find out that it accepts a parameter of type `Design`, and returns a list of signals in that design. Therefore, after executing the `sig()` function, the variable `sig1` will consist of a list of signals in the design.

## Variable Type

Rules Checker 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 environment)

- Non-Object Variables

Assigned mathematical values, strings, and so on.

Object variables are identified by their names, which must start with the object name that they will be assigned. For instance, in the previous example, the return value of the function `sig()` is a signal object, therefore the variable name must begin with `Sig`. Any string can be appended to the base object to create the variable name. Some valid examples of a Signal variable are `sig1,sig2,sig_global,sig_ECL`, and so on. Violations of this variable-naming requirement result in errors when the rule is compiled.

Non-object variables are identified by their names, which do not start with any object name. The non-object variables can be assigned mathematical values, strings or other non-object variables.

An example of non-object variable is `val1 = 2`.

## Assignment Operator and Comparison Operator

In the Rules Checker rule language, assignment is expressed with the:= operator. The syntax for the assignment operator is

`<var> := <expr1>`

where <*var*> is a variable (object or non-object). The type of the variable should be the same as the type of *expr1*. Type checking will be done at compile time. At run time the return value of:= is the RHS of the statement. Reassignment is not allowed. For example:

```
val1 := 5 AND
...AND
val1 := 6
```

The operator == is used for comparison. The syntax for the comparison operator is

`<expr1> == <expr2>`

The types of the left and right operands need to be the same. Type checking will be done at compile time. The condition will return `TRUE` if both operands are the same; otherwise, it will return `NULL`.

## If Construct

The syntax for the if construct is as follows:

`<var> := if(<cond>,<expr1>,<expr2>)`

where <*var*>, <*expr1*>, and <*expr2*> are all of the same type. If <*cond1*> is TRUE, <*expr1*> is evaluated and its result is returned. Otherwise, <*expr2*> is evaluated and its result is returned. Assignments are allowed in <*expr1*> and <*expr2*>. If a variable is assigned in <*expr1*> and used outside it, then assign it in <*expr2*> also. Within an if construct, only:= and == operators are supported.

## Conditional Operators

Conditional operators join multiple boolean results to create complex logical equations. Examples of these operators are AND, OR, and XOR. Each of the conditional statements must evaluate to a TRUE or non-null value if they are joined with the AND operator. If not, the Message construct will not be executed.

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

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

These operators behave as follows in lists:

- AND

  a AND b. If neither list is NULL, then AND returns TRUE. Otherwise, it returns NULL.

- OR

a OR b. If either list is non-NULL, then OR returns TRUE. Otherwise, it returns NULL.

- XOR

  a XOR b. If exactly one list is NULL, then XOR returns TRUE. Otherwise, it returns NULL.

- ==

  a == b. If any element in a is the same as any element in b, then == returns a TRUE.

- /=

  a /= b. If any element in a is not the same as any element in b, then /= returns a TRUE.

- >

  a > b. If any element in a is greater than any element in b, then > returns a TRUE.

- >=

  a >= b. If any element in a is greater than or equal to any element in b, then >= returns a TRUE.

- <

  a < b. If any element in a is less than any element in b, then < returns a TRUE.

- <=

  a <= b. If any element in a is less than or equal to any element in b, then <= returns a TRUE.

For information on lists, see the following topic on list manipulation.

### Sample rule file

```
include "cp_config.h"
use logical;
RuleDefine
 Rule test_1
    design1 AND
    val_ff := 2 ** 3
    Message(INFO, design1, "dddd", "2 ** 3 ?val_ff");  EndRule EndRuleDefine
```

The output would be 8.

# List Manipulation

Lists are the only data structures in the Rules Checker rule language. Every variable type in the language can be used to create a list of one or more elements. In the previous section, implied looping over a list of base objects by the Rules Checker engine was used to create lists. However, the language supports two other basic forms of list creation: Implicit and Explicit list creation. This section describes these two language characteristics and the operations used to manipulate lists.

## What Are Lists?

A Rules Checker list is a collection of one of the following:

- Object Elements
- Non-object Elements

Lists are homogeneous; that is, they can contain only one type of object. For example, a list cannot contain instances and signals, nor can a list contain strings and integers.

Example

In this example, `inst1` is a list of instances (object elements) and `val1` is a list of strings (non-object elements).

```
RuleDefine
        Rule list_def
            inst1 := inst(design1) AND
            val1 := toInt(getHierProperty(inst1, "MAX_DELAY"))
            Message(Info,design1,"Instances ?inst1 , delays : ?val1");
        EndRule
EndRuleDefine
```

Lists are created by functions. The previous section described how lists of Base Objects are created by the engine to evaluate each selected rule. This type of implied list manipulation does not require any special understanding by the programmer. However, the length of the Base Object list determines the number of times the rule is executed on the design database.

Example

In the following example `sig1` is assigned each element of the base object list, one at a time, each time the rule is executed.

```
RuleDefine
    Rule print_sig
        sig1
        Message(Info,sig1,"Name : ?sig1");
    EndRule
EndRuleDefine
```

Before examining the other two types of lists, understand the difference between a “variable list” and a “oneof list.” Variable lists are created by functions in the rule language, whereas oneof lists are enumerated in the rule via curly brackets ({ }). A oneof list is a list of constants that contains Booleans, strings, integers, and floats.

Example

Consider a signal with the value of TEST connected to four instances: “1,” “2,” “3,” and “4.” The dynamic list val1 will contain four elements: “1,” “2,” “3,” “4.”

```
RuleDefine
        Rule match
            inst1 := inst(sig1) and
            val1 := getProperty(inst1, "TEST") and
            sig2 := matchProperty(sig1, "ROUTE_PRIORITY", val1)
            Message (Info, sig2, "Sigs found with matching ROUTE_PRIORITY")
        EndRule
EndRuleDefine
```

In the above example the `matchProperty` function is called

#_elements_list_sig1 * 1 * #_elements_list_val1 = 1*1* 4 times

1 is used because the second argument contains just one element. However, suppose the third argument is a oneof list:

```
RuleDefine
    Rule match
        /* {"1", "2", "3", "4"} is a static list */
        sig2 := matchProperty(sig1, "ROUTE_PRIORITY", {"1","2","3","4"})
        Message (Info, sig1, "Signal found with matching ROUTE_PRIORITY")
    EndRule
EndRuleDefine
```

In this example, `sig2` will be equal to the value of `sig1` if the value of the property “ROUTE_PRIORITY” attached to `sig1` is equal to “1,” “2,” “3,” or “4”. The number of times the `matchProperty` function is called is

#_elements_list_sig1 *1 * 1 = 1* 1 *1 = 1

because the oneof list is treated as a single element by the engine. The whole list is passed to the predicate. The engine does not loop over the oneof list.

## List Manipulation Routines

The difference between list functions and any other functions is that list functions operate on full lists, while other functions operate on one element of the list at a time.

Example

The following rule uses `count` to count all non-resistor instances in the design.

```
RuleDefine
        Rule ListEg4
            inst1 := inst(design1) and
            inst_res := matchHierProperty(inst1, "COMP_TYPE", "RES") and
            inst_nonres := remove(inst1, inst_res) and
            val := count(inst_nonres)
            Message(INFO,inst_names, "?val");
        EndRule
EndRuleDefine
```

In this rule, `matchHierProperty` is called implicitly once on each element in the list inst1. However, `remove` and `count` are called only once because they are list functions.

For example, consider a design with ten instances. In this case, `matchHierProperty` will be executed 10 times (once for each element in the list `inst1`) while `remove` and `count` will be executed once.

The following table describes some of the predicates that operate on lists. Each of the following predicates except `append` returns the result. The input list is not split. For example, `remove(list1, list2)` removes elements of `list2` from `list1`, and the `remove` predicate returns `list3`, which contains all elements of `list1` that are not in `list2`. As an exception, `append(list1, list2)` attaches `list2` to `list1`, actually modifying the input argument.

| Function | Description |
|---|---|
| `car(`*`list`*`)` | Returns the first element |
| `cdr(`*`list`*`)` | Returns the list without the first element |
| `nth(`*`list, i`*`)` | Returns the nth 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 |
| `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 |

| Function | Description |
|---|---|
| `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 formed by concatenating the strings from list1 |
| `append(`*`list1, list2`*`)` | Modifies list1 by appending the values from list2 |

### Examples

The following examples demonstrate the list manipulation functions.

Assume the following list:

```
Vowels = (a e i o u)
```

**Note:** The parentheses are not recognized as a syntax to specify lists. They are used here for clarification purposes only.

The table shows several list operations and their output.

| Operation | Output |
|---|---|
| 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 in these examples actually constitutes a list containing one element.

## Foreach Construct

The previous functions used to manipulate lists assume that you know the location of the element you want from the list. However, this is not always the case. The Foreach function has a mechanism for iterating over all members in a list, and uses the following syntax:

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

**Note:** The return value of the Foreach statement is a new list. Because the Foreach statement is a conditional line in a rule, it evaluates to true if the list that is returned is non-null.

Example

The first parameter of the Foreach construct is a list, which is also used inside the condition statements as the variable, as shown below:

```
/* List all instances that have more than two unused instterms *
RuleDefine
        Rule unused_inst
            inst1 := inst(design1) AND
            inst2 := foreach(inst1, count(unused(instTerm(inst1))) > 2)
            Message(ERROR, inst2,"Instances that have more than two unused
instterms are ?inst2");
         EndRule
EndRuleDefine
```

In this example, `inst1` is the list that is used in the Foreach statement. It is also the variable name used inside the condition. `inst2` is a subset of `inst1` that contains those elements of `inst1` that satisfy the condition section of the foreach construct.

## Saving Intermediate Results Within a foreach Statement

You will often need to collect intermediate results while doing computations inside a foreach statement. This is especially helpful for debugging rules as you develop them. When doing this, the value of the variable used inside the foreach statement can be any value, as it is treated as a temporary variable. The `append` predicate gives a convenient way to accumulate the results in one variable.

Example

The following example shows how to find all the differential outputs of an instance (for example, Y<0> and -Y<0>). The `name` predicate returns the name in the following format: 1P|Y<0>. So, you'll need to extract Y<0> and then look for -Y<0>.

```
RuleDefine
    Rule differential_outputs
            instTerm1 := output(instTerm(inst1)) and
            instTerm2 := instTerm1 and
            instTerm3 := foreach(instTerm1,
            val1 := getSubString(name(instTerm1), "|", "") and
            val2 := "-" + val1 and
            instTerm4 := foreach(instTerm2,
                matchName(getSubString(name(instTerm2), "|", ""), val2)
            ) and
            append(instTerm5, instTerm4)
            ) and
            instTerm_list := concat(instTerm3, instTerm5)
            Message(Info,instTerm_list, "Differential outputs are
?instTerm_list");
    EndRule
EndRuleDefine
```

Printing `instTerm4` in the message results in an unpredictable value, so append is used to accumulate all the instTerm4's in instTerm5. `append` differs from other predicates in the following ways:

- `append` creates its first argument, so it must be an identifier.

  Passing another value, such as a predicate call, to `append` results in a compilation error.

- `append` can create a variable only once, so trying to use `append` twice on a variable results in an error.

  So the same variable cannot be used as the first argument of two different calls to `append`. However, iterating over the same `append` call is allowed using `foreach`, as in the above example.

### Findfirst Construct

The findfirst construct is used to support exiting a foreach loop once a specific condition is met. The syntax for this construct is

```
<var1> := findfirst(<variable>,<expr>)
```

where <*variable*> is a list of objects or non-objects. `findfirst` returns the first object in the list <variable> for which <*expr*> returns `TRUE`. The type of <*var1*> is the same as the type of <*variable*>. This type of checking is done at compile time.

# Environment-Specific Programming

The Advanced Rule Language (ARL) is used by other Cadence tools, so it is necessary that you understand the current environment for which you are programming. This section applies

to Allegro Design Entry HDL Rules Checker; however, the information in this section is applicable to any of the environments supplied by other ARL-based tools.

Rules Checker supplies four environments for rule execution and development.

- Body
- Graphical
- Logical
- Physical

It is important to understand the data supplied in each of the environments, so you can make the correct decision as to the environment to use and the base object to select.

## Rules Checker Body Environment

The Body environment permits access to all data located in the `chips.prt` files and `symbol views` ASCII files. It also supplies minimal signal name information from the corresponding logic file associated with a `symbol views` file. The Body rules supplied with Rules Checker contain rules that provide cross-view checking, property verification, and body drawing standards.

The Rules Checker body environment contains the seven following base objects:

| | |
|---|---|
| **Design** | The `symbol view` file is used as the base object, so each selected rule will be executed once per body file. |
| **Note** | The selected note-based rules will be executed once for every note in the `symbol view`. |
| **Prop** | The selected property-based rules will be executed once for every property in the `symbol view`. |
| **Bodypin** | The selected bodypin-based rules will be executed once for every pin in the `symbol view`. |
| **Seg** | The selected segment-based rules will be executed once for every line segment in the body file. |
| **Arc** | The selected arc-based rules will be executed once for every arc in the body file. |
| **PhysPackType** | The selected physpacktype-based rule will be executed once for each body primitive name specified in the `chips.prt` file. |

Example

The following rule uses this algorithm to print all overlapping visible properties and notes:

foreach property in the design
foreach note in the design
if property visible and overlaps with note
report error
endfor
endfor

```
RuleDefine
        Rule prop_note_overlap
            prop_s := prop(design1) AND
            props := matchNoName(prop_s, {"BUBBLE_GROUP","BUBBLED"}) AND
            prop1 := matchNoName(props, {USER_DEFINED_PROPS_TO_IGNORE})
                AND
            note1 := note(design1) AND
            prop2 := foreach(prop1,(nameVis(prop1) OR valueVis(prop1)
                AND
            note2 := overlap(note1, bbox(prop1)) AND
            append(note3, note2))

            Message(PROP_NOTE_OVERLAP_SEVERITY, prop2 note3,
                STD_SHORT_ERR_PROP_NOTE_OVERLAP,
                STD_ERR_PROP_NOTE_OVERLAP);
        EndRule
EndRuleDefine
```

## Rules Checker Graphical Environment

The graphical environment supplies access to all data located in schematic view and `chips.prt` files. There is also minimal access to some data from the body files. For rules requiring detailed access to `symbol view` files, you might need to write the rule in the Body environment. The graphical rules supplied with Rules Checker contain rules for sectioning checks, property verification, and graphic drawing standards. The graphical environment supplies routines for accessing locational data in a schematic, minimal connectivity data, and property data added by the user or backannotated from another tools.

The Rules Checker graphical environment contains ten base objects:

| | |
|---|---|
| **Design** | The Schematic view is used as the base object, so each selected rule is executed once per logic drawing. |
| **Inst** | The selected instance-based rules are executed once for every instance in the schematic view. |

| | |
|---|---|
| **Body** | The selected body-based rules are executed once for every body in the logic drawing. Iterating over bodies in a drawing is a subset of the `Inst` base object. When Body is selected as the base object, a list of bodies used in the logic drawing is created; therefore, multiple instances of a body in the drawing do not affect the number of times the rule executes. |
| **Pin** | The selected pin-based rules are executed once for every pin attached to an instance in the design. |
| **Wire** | The selected wire-based object is executed for every wire in the drawing. This differs from the seg object in that a wire can be composed of many segments. You can use this object to prevent multiple errors from being reported on the same rule written for segments. Generally, this object is used for rules that check connectivity or properties attached to a wire. |
| **Note** | The selected note-based rules are executed once for every note in the logic drawing. |
| **Prop** | The selected property-based rules are executed once for every property in the logic drawing. |
| **Bodypin** | The selected bodypin-based rules are executed once for every pin attached to the unique list of body names in the logic drawing. (See the Body base object description above.) |
| **Seg** | The selected segment-based rules are executed once for every line segment in the logic drawing. This object needs to be contrasted to the wire object, which is a set of one or more segments. The seg object is used primarily for drawing standard checks, such as overlap and positional information. |
| **PhysPackType** | The selected physpacktype-based rule is executed once for each instance of primitive name specified in the `chips.prt` file. |

Example

The following rule reports all synonym bodies whose signal widths are not same, and uses the following algorithm:

```
foreach synonym body
set the width of the segment connected on either side of the body.
if the widths are not same report an error
endfor
```

```
RuleDefine
        Rule synonym_width_mismatch
            body1 := matchName(body(inst1), {SYNONYM_BODY}) AND
            seg1: = seg(pin(inst1)) AND
            sig_width := signalwidth(seg1) AND
            val1: = nth(sig_width, 1) AND
            val1 /= sig_width
            Message(SYNONYM_WIDTH_MISMATCH_SEVERITY, inst1 seg1,
                STD_SHORT_ERR_SYNONYM_WIDTH_MISMATCH,
                STD_ERR_SYNONYM_WIDTH_MISMATCH);
        EndRule
EndRuledefine
```

## Rules Checker Logical Environment

The Logical environment supplies access to all expanded data. This is not necessary in the Body and Graphical environments because the access routines read directly from the ASCII files supplied as Allegro Design Entry HDL input and output.

Be sure that rules written for the Logical environment actually need to be written in that environment. Because the Logical environment compiles and flattens the design, a rule that could also be written in the Graphical environment should not be written in the Logical environment. Rules that require a flattened database require the logical environment. The Logical rules supplied with Rules Checker contain rules for loading and IO checks, property verification, signal naming, and advanced electrical checks.

The Rules Checker logical environment contains six base objects:

| | |
|---|---|
| **Design** | The design is used as the base object so each selected rule is executed once per design. |
| **Inst** | The selected instance-based rules are executed once for every instance in the design. |
| **Body** | The selected body-based rules are executed once for every body in the design. Iterating over bodies in a design is a subset of the Inst base object. When Body is selected as the base object, a list of bodies used in the design is created; therefore, multiple instances of bodies in the design do not affect the number of times the rule executes. |
| **instTerm** | The selected instterm-based rules are executed once for each pin attached to an instance in the design. |

| | |
|---|---|
| **Sig** | The selected signal-based rules are executed for every signal in the design. This differs from wire and seg objects in other environments because a signal can be composed of many segments and many wires. This signal may also have multiple names associated with it if any synonym or hierarchical bodies are used in the schematic. |
| **Term** | The selected terminal-based rules are executed once for every pin attached to the unique list of body names in the design. (See the Body base object description above.) |

Example

The rule below checks if all the declared power signals exist in the design as global signals, and uses the following algorithm:

foreach instance
get the value of the POWER_GROUP property.
Extract the list of signals from the value.
Check if all the signals specified on RHS are global and
exist in the design
If not error
endfor

```
POWER_GROUP = VCC=VSS;GND=ANALOG_GND;


RuleDefine
        Rule power_group1
            str : = GetAllSubStrings(StripWhiteSpaces(getHierProperty(inst1,
                "POWER_GROUP")),":=",{";",""})  AND
            str2 := GetAllSubStrings(StripWhiteSpaces(getHierProperty(inst1,
                "POWER_GROUP")),{";", ""},{";",""})   AND
            str3 := foreach(str, isNull(isglobal(findsig(str)))) AND
            (
                count(str3) > 0  OR
                sig1 := isnotglobal(findsig(str))
            )
            Message (POWER_GROUP1_SEVERITY, inst1, STD_SHORT_POWER_GROUP1,
                STD_ERR_POWER_GROUP1);
        EndRule power_group1
EndRuleDefine
```

In the logical environment, all signals with the name "NC" and those with 'OPEN_' in their name are considered to be non-existent and are ignored.

You must ensure that signal names do not contain the string 'OPEN_'. This string is used when you want to add a physically visible wire but do not want any connectivity associated with it in the logical environment.

## Rules Checker Physical Environment

The Physical environment supplies access to the packaged design netlist and other physical information. You must successfully run the Packager or Packager-XL before running rules in the physical environment. At this point in the design process, Rules Checker has access to all data passed on to down-stream tools, packaged loading data, and any other information associated with a packaged part. The Physical rules supplied with Rules Checker contain rules for physical loading checks, cost and power analysis, and netlist property verification.

The Rules Checker logical environment contains eight base objects:

| | |
|---|---|
| **Design** | The packaged design is used as the base object, so each selected rule is executed once per design. |
| **Inst** | The selected instance-based rules are executed once for every logical instance in the packaged design. |
| **Body** | The selected body-based rules are executed once for every body in the packaged design. Iterating over bodies in a design is a subset of the Inst base object. When Body is selected as the base object, a list of bodies used in the design is created, so multiple instances of a body in the design do not affect the number of times the rule executes. |
| **instTerm** | The selected instterm-based rules are executed once for every pin attached to an instance in the design. |
| **Sig** | The selected signal-based rules are executed for every signal in the packaged design. This differs from wire and seg objects in other environments because a signal can be composed of many segments and many wires. This signal may also have multiple names associated with it if any synonym or hierarchical bodies are used in the schematic. |
| **packInst** | The selected packaged instance-based rules are executed for each packaged instance in the design; that is, every unique reference designator. Packaged instances are formed from one or more logical instances by the Packager. |
| **Pin** | The selected pin-based rules are executed once per pin that is attached to a packaged instance (packInst). |
| **Term** | The selected term-based rules are executed once for every pin attached to the unique list of body names in the design. (See the Body base object description above.) |

Example

The following rule checks that no non-preferred parts are used in the design and uses the following algorithm:

foreach packaged instance
Get the value of the property specified by STATUS
Compare the value against a constant NONPREF
If match found error
endfor

```
RuleDefine
        Rule non_preferred_part
            prop_val:= getProperty(unique(body(packinst1)),"STATUS")
                AND
            matchName(prop_val ,"NONPREF") AND
            inst1:= inst(packinst1) AND
            prop_name = "STATUS" AND
            val1 := "PREF" AND
            val2 := "NONPREF"
            Message(NON_PREFERRED_PART_SEVERITY, inst1,
                STD_SHORT_ERR_NON_PREFERRED_PART,
                STD_ERR_NON_PREFERRED_PART);
        EndRule
EndRuleDefin
```

# A

# Allegro Design Entry HDL Rules Checker Rules

## Overview

This appendix describes the default rules included with Rules Checker.

Each rule description contains the following:

- A brief description of the rule
- Information about data (such as properties) required by the rule.
- Assumptions made by the rule
- Default variable settings
- The default severity level of the rule
- User customization information

Many of the rules in this chapter contain user-definable parameters you can set.

## General Rules

This section contains information on the general rules included with Rules Checker. These rules are contained in the `property_checks.rle` file.

Be sure to choose the logical environment when using these rules.

### biput_pin_prop_exists

Reports biput pins where a user-specified property either exists, or is missing (determined by user).

### Required Data

None

### Assumptions

None

### Default Variables

As a default, the rule checks for the existence of OUTPUT_CAP and PIN1 properties on all biput pins.

### Default Severity

Info

### User Customization Information

You can customize the variable macros, reported severity, and error messages for this rule:
**Variable Macros**

| Macro | Definition | Default |
|---|---|---|
| LOGICAL_BIPUT_PIN_PROP_EXIST_NAME | Name of property to check | “OUTPUT_CAP” |
| LOGICAL_BIPUT_PIN_PROP_EXISTS | If set to 0, checks for absence of property. If set to 1, checks for existence of property. | 1 |

**Reported Severity**

| Macro | Default |
|---|---|
| LOGICAL_BIPUT_PIN_PROP_EXISTS_SEVERITY | Info |

#### *Error Messages*

STD_SHORT_ERR_LOGICAL_BIPUT_PIN_PROP_EXISTS
STD_ERR_LOGICAL_BIPUT_PIN_PROP_EXISTS

## count_inst

Prints the number of instances in the design.

### Required Data

None

### Assumptions

None

### Default Variables

None

### Default Severity

Info

### User Customization Information

You can customize the reported severity and error messages for this rule:
**Reported Severity**

| Macro | Default |
|---|---|
| COUNT_INST_SEVERITY | Info |

#### *Error Messages*

STD_SHORT_ERR_COUNT_INST
STD_ERR_COUNT_INST

## count_pins

Prints the number of pins in the design.

### Required Data

None

### Assumptions

None

### Default Variables

None

### Default Severity

Info

### User Customization Information

You can customize the reported severity and error messages for this rule:
**Reported Severity**

| Macro | Default |
|---|---|
| COUNT_PIN_SEVERITY | Info |

#### *Error Messages*

STD_SHORT_ERR_COUNT_PIN
STD_ERR_COUNT_PIN

## count_sig

Prints the number of signals in the design.

### Required Data

None

### Assumptions

None

### Default Variables

None

### Default Severity

Info

### User Customization Information

You can customize the reported severity and error messages for this rule:
**Reported Severity**

| Macro | Default |
|---|---|
| COUNT_SIG_SEVERITY | Info |

#### *Error Messages*

STD_SHORT_ERR_COUNT_SIG
STD_ERR_COUNT_SIG

## input_pin_prop_exists

Reports input pins where a user-specified property either exists, or is missing (determined by user).

### Required Data

None

### Assumptions

None

### Default Variables

As a default, the rule checks for the existence of the INPUT_CAP property on all input pins in the design.

### Default Severity

Info

### User Customization Information

You can customize the variable macros, reported severity, and error messages for this rule:
**Variable Macros**

| Macro | Definition | Default |
|---|---|---|
| LOGICAL_INPUT_PIN_PROP_EXIST_NAME | Name of property to check | “INPUT_CAP” |
| LOGICAL_INPUT_PIN_PROP_EXISTS | If set to 0, checks for absence of property. If set to 1, checks for existence of property. | 1 |

**Reported Severity**

| Macro | Default |
|---|---|
| LOGICAL_INPUT_PIN_PROP_EXISTS_SEVERITY | Info |

#### *Error Messages*

STD_SHORT_ERR_LOGICAL_INPUT_PIN_PROP_EXISTS
STD_ERR_LOGICAL_INPUT_PIN_PROP_EXISTS

## inst_prop_exists

Highlights instances where a user-specified property(s) either exists, or is missing (determined by user).

### Required Data

None

### Assumptions

None

### Default Variables

As a default, the rule checks for the existence of MAX_DELAY property on each instance in the design.

### Default Severity

Info

### User Customization Information

You can customize the variable macros, reported severity, and error messages for this rule:
**Variable Macros**

| Macro | Definition | Default |
|---|---|---|
| LOGICAL_INSTANCE_PROP_EXIST_NAME | Name of property to check | “MAX_DELAY” |
| LOGICAL_INSTANCE_PROP_EXISTS | If set to 0, checks for absence of property. If set to 1, checks for existence of property. | 1 |

### Reported Severity

| Macro | Default |
|---|---|
| LOGICAL_INST_PROP_EXISTS_SEVERITY | Info |

### *Error Messages*

STD_SHORT_ERR_LOGICAL_INST_PROP_EXISTS
STD_ERR_LOGICAL_INST_PROP_EXISTS

## inst_prop_range_check

Reports instances in which the value of a specified property(s) is outside of a user-defined range.

### Required Data

None

### Assumptions

This rule assumes the value for property is of type float.

### Default Variables

As a default, the rule checks to ensure that MAX_DELAY property on an instance has a value between 1 and 100000.

### Default Severity

Error

### User Customization Information

You can customize the variable macros, reported severity, and error messages for this rule:
**Variable Macros**

| Macro | Definition | Default |
|---|---|---|
| LOGICAL_INST_PROP_RANGE_NAME | Comma-separated list of property(s) to check | “MAX_DELAY” |
| LOGICAL_INST_PROP_MIN_ VALUE | Comma-separated minimum value(s) allowed for instance property(s) | “1” |

### Variable Macros

| Macro | Definition | Default |
|---|---|---|
| LOGICAL_INST_PROP_MAX_ VALUE | Comma-separated maximum value(s) allowed for instance property(s) | “100000” |

### Reported Severity

| Macro | Default |
|---|---|
| LOGICAL_INST_PROP_RANGE_CHECK_SEVERITY | Error |

#### *Error Messages*

STD_SHORT_ERR_LOGICAL_INST_PROP_RANGE_CHECK
STD_ERR_LOGICAL_INST_PROP_RANGE_CHECK

## invalid_ref_des_assignment

Checks whether two instances of different parts have the same reference designator.

### Required Data

The LOCATION and CDS_LOCATION properties must be specified on each instance in the drawing.

### Assumptions

None

### Default Variables

None

### Default Severity

Error

### User Customization Information

You can customize the reported severity and error messages for this rule:
**Reported Severity**

| Macro | Default |
|---|---|
| INVALID_REF_DES_ASSIGNMENT_SEVERITY | Error |

#### *Error Messages*

STD_SHORT_ERR_INVALID_REF_DES_ASSIGNMENT
STD_ERR_INVALID_REF_DES_ASSIGNMENT

## invalid_ref_des_count

Checks to ensure the number of instances of the same part with the same reference designator does not exceed the number of sections in the package.

### Required Data

The LOCATION and CDS_LOCATION properties must be specified.

### Assumptions

None

### Default Variables

None

### Default Severity

Error

### User Customization Information

You can customize the reported severity and error messages for this rule:
**Reported Severity**

| Macro | Default |
|---|---|
| INVALID_REF_DES_COUNT_SEVERITY | Error |

#### *Error Messages*

STD_SHORT_ERR_INVALID_REF_DES_COUNT
STD_ERR_INVALID_REF_DES_COUNT

## nets_shorted

Checks if the specified nets are shorted.

### Required Data

None

### Assumptions

None

### Default Variables

As a default, Rules Checker checks the VCC and GND nets, VDD and GND1 nets, and VEE and GND2 nets.

### Default Severity

Error

### User Customization Information

You can customize the variable macros, reported severity, and error messages for this rule:
**Variable Macros**

| Macro | Definition | Default |
|---|---|---|
| NET_NAME1 | Name of signal to test against NET_NAME2 | “VCC” |
| NET_NAME2 | Name of signal to test against NET_NAME1 | “GND” |
| NET_NAME3 | Name of signal to test against NET_NAME4 | “VEE” |
| NET_NAME4 | Name of signal to test against NET_NAME3 | “GND1” |
| NET_NAME5 | Name of signal to test against NET_NAME6 | “VDD” |
| NET_NAME6 | Name of signal to test against NET_NAME5 | “GND2” |

If specifying multiple values:

| Macro | Default |
|---|---|
| NET_NAME1 | “VCC,” “VEE,” “VDD” |
| NET_NAME2 | “GND,” “GND1,” “GND2” |

### Reported Severity

| Macro | Default |
|---|---|
| NETS_SHORTED_SEVERITY | Error |

#### *Error Messages*

STD_SHORT_ERR_NETS_SHORTED STD_ERR_NETS_SHORTED

## null_body_prop_val

Checks for null values on properties attached to bodies.

### Required Data

None

### Assumptions

None

### Default Variables

None

### Default Severity

Error

### User Customization Information

You can customize the reported severity and error messages for this rule:
**Reported Severity**

| Macro | Default |
|---|---|
| NULL_BODY_PROP_VAL_SEVERITY | Error |

#### *Error Messages*

STD_SHORT_ERR_NULL_BODY_PROP_VAL STD_ERR_NULL_BODY_PROP_VAL

## null_inst_prop_val

Checks for null values on properties attached to instances.

### Required Data

None

### Assumptions

None

### Default Variables

None

### Default Severity

Error

### User Customization Information

You can customize the variable macro, reported severity, and error messages for this rule:
**Variable Macro**

| Macro | Definition | Default |
|---|---|---|
| LOGICAL_IGNORE_NULL_INST_VAL_PROP | Property(s) on instance with null values to ignore | “SEC_TYPE” |

**Reported Severity**

| Macro | Default |
|---|---|
| NULL_INST_PROP_VAL_SEVERITY | Error |

#### *Error Messages*

STD_SHORT_ERR_NULL_INST_PROP_VAL STD_ERR_NULL_INST_PROP_VAL

## null_pin_prop_val

Checks for null values on properties attached to pins.

### Required Data

None

### Assumptions

None

### Default Variables

None

### Default Severity

Error

### User Customization Information

You can customize the variable macro, reported severity, and error messages for this rule:
**Variable Macro**

| Macro | Definition | Default |
|---|---|---|
| LOGICAL_IGNORE_NULL_PIN_VAL_PROP | Property(s) on pin with null values to ignore | “SEC_TYPE” |

**Reported Severity**

| Macro | Default |
|---|---|
| NULL_PIN_PROP_VAL_SEVERITY | Error |

#### *Error Messages*

STD_SHORT_ERR_NULL_PIN_PROP_VAL STD_ERR_NULL_PIN_PROP_VAL

## null_sig_prop_val

Checks for null values on properties attached to signals.

### Required Data

None

### Assumptions

None

### Default Variables

None

### Default Severity

Error

### User Customization Information

You can customize the reported severity and error messages for this rule:
**Reported Severity**

| Macro | Default |
|---|---|
| NULL_SIG_PROP_VAL_SEVERITY | Error |

#### *Error Messages*

STD_SHORT_ERR_NULL_SIG_PROP_VAL STD_ERR_NULL_SIG_PROP_VAL

## output_pin_prop_exists

Reports output pins where a user-specified property either exists, or is missing (determined by user).

### Required Data

None

### Assumptions

None

### Default Variables

As a default, the rule checks for the existence of the OUTPUT_CAP property on all output pins in the design.

### Default Severity

Info

### User Customization Information

You can customize the variable macros, reported severity, and error messages for this rule:
**Variable Macros**

| Macro | Definition | Default |
|---|---|---|
| LOGICAL_OUTPUT_PIN_PROP_EXIST_NAME | Name of property to check | “OUTPUT_CAP” |
| LOGICAL_OUTPUT_PIN_PROP_ EXISTS | If set to 0, checks for absence of property. If set to 1, checks for existence of property. | 1 |

**Reported Severity**

| Macro | Default |
|---|---|
| LOGICAL_OUTPUT_PIN_PROP_EXISTS_SEVERITY | Info |

#### *Error Messages*

STD_SHORT_ERR_LOGICAL_OUTPUT_PIN_PROP_EXISTS
STD_ERR_LOGICAL_OUTPUT_PIN_PROP_EXISTS

## pin_prop_range_check

Highlights pins in which the value of a specified property is outside of a user-defined range.

### Required Data

None

### Assumptions

This rule assumes:

- The value for property is of type float
- The value for LOGICAL_PIN_PROP_MIN_VALUE is less than the value for LOGICAL_PIN_PROP_MAX_VALUE

### Default Variables

As a default, the rule checks that the INPUT_CAP property on pins has a value between 1 and 100.

### Default Severity

Error

### User Customization Information

You can customize the variable macros, reported severity, and error messages for this rule:
**Variable Macros**

| Macro | Definition | Default |
|---|---|---|
| LOGICAL_PIN_PROP_RANGE_NAME | Name of pin property | “INPUT_CAP” |
| LOGICAL_PIN_PROP_MIN_VALUE | Minimum value allowed for pin property | “1” |
| LOGICAL_PIN_PROP_MAX_VALUE | Maximum value allowed for pin property | “100” |

**Reported Severity**

| Macro | Default |
|---|---|
| LOGICAL_PIN_PROP_RANGE_CHECK_SEVERITY | Error |

***Error Messages***

STD_SHORT_ERR_LOGICAL_PIN_PROP_RANGE_CHECK
STD_ERR_LOGICAL_PIN_PROP_RANGE_CHECK

## power_group1

Checks that reassigned power pins (specified by the POWER_GROUP property) on instances are reassigned to global nets. Rules Checker issues a warning if any of an instance's power pins are reassigned to non-global nets or nonexistent nets.

### Required Data

None

### Assumptions

The rule assumes you have declared nets used in POWER_GROUP properties as global signals. This is necessary for accurately checking POWER_GROUP property reassignments.

### Default Variables

None

### Default Severity

Warning

### User Customization Information

You can customize the reported severity and error messages for this rule:
**Reported Severity**

| Macro | Default |
|---|---|
| POWER_GROUP1_SEVERITY | Warning |

### *Error Messages*

STD_SHORT_POWER_GROUP1
STD_ERR_POWER_GROUP1

## power_group2

Checks that any reassigned power pins have been assigned to nets defined in the `chips_prt` file. For example, this rule detects if a POWER_GROUP property has a value of VCP=NEW_VCC, but VCP is not defined in the POWER_PINS line in the `chips_prt` file.

### Required Data

None

### Assumptions

None

### Default Variables

None

### Default Severity

Error

### User Customization Information

You can customize the reported severity and error messages for this rule:
**Reported Severity**

| Macro | Default |
|---|---|
| POWER_GROUP2_SEVERITY | Error |

### *Error Messages*

STD_SHORT_POWER_GROUP2
STD_ERR_POWER_GROUP2

## power_group3

Checks that the default power pin assignments on instances with default and reassigned power pins are valid. For example, if you use a POWER_GROUP property to reassign VCC on an instance, this rule ensures that VCC is a valid signal in the `chips_prt` file.

### Required Data

None

### Assumptions

The rule assumes that nets declared in the POWER_PIN entries in the `chips_prt` file are global signals. This is necessary for accurately checking POWER_PIN information in your design.

### Default Variables

None

### Default Severity

Warning

### User Customization Information

You can customize the reported severity and error messages for this rule:
**Reported Severity**

| Macro | Default |
|---|---|
| POWER_GROUP3_SEVERITY | Warning |

### *Error Messages*

STD_SHORT_POWER_GROUP3
STD_ERR_POWER_GROUP3

## power_group4

Checks to ensure power pins on instances without POWER_GROUP properties are tied to global signals that exist in the schematic, that is, it ensures that the signals specified in the POWER_PINS entry in the `chips_prt` file actually exist in the schematic.

### Required Data

None

### Assumptions

The rule assumes that global nets declared in the POWER_PIN entries in the `chips_prt` file are global signals. This is necessary for accurately checking POWER_PIN information in your design.

### Default Variables

None

### Default Severity

Warning

### User Customization Information

You can customize the reported severity and error messages for this rule:
**Reported Severity**

| Macro | Default |
|---|---|
| POWER_GROUP4_SEVERITY | Warning |

### *Error Messages*

STD_SHORT_POWER_GROUP4
STD_ERR_POWER_GROUP4

## self_loop_check

Checks every instance in your design to ensure its input and output pins are not connected to each other.

### Required Data

None

### Assumptions

None

### Default Variables

None

### Default Severity

Warning

### User Customization Information

You can customize the reported severity and error messages for this rule:
**Reported Severity**

| Macro | Default |
|---|---|
| SELF_LOOP_CHECK_SEVERITY | Warning |

### *Error Messages*

STD_SHORT_ERR_SELF_LOOP_CHECK STD_ERR_SELF_LOOP_CHECK

## sig_prop_exists

Checks that the specified property exists (or does not exist, as per user preference) on every signal in the design.

### Required Data

None

### Assumptions

None

### Default Variables

As a default, the rule checks for the existence of the ROUTE_PRIORITY property on each signal in the design.

### Default Severity

Warning

### User Customization Information

You can customize the variable macros, reported severity, and error messages for this rule:
**Variable Macros**

| Macro | Definition | Default |
|---|---|---|
| LOGICAL_SIGNAL_PROP_EXIST_NAME | Name of property to check. | “ROUTE_PRIORITY” |
| LOGICAL_SIGNAL_PROP_EXISTS | If set to 0, checks for absence of property. If set to 1, checks for existence of property. | 1 |

### Reported Severity

| Macro | Default |
|---|---|
| LOGICAL_SIG_PROP_EXISTS_SEVERITY | Info |

#### *Error Messages*

STD_SHORT_ERR_LOGICAL_SIG_PROP_EXISTS
STD_ERR_LOGICAL_SIG_PROP_EXISTS

## sig_prop_range_check

Checks that the values for a specific signal property are within a specified range.

### Required Data

None

### Assumptions

The rule assumes the following:

- The property value is an integer.
- LOGICAL_SIGNAL_PROP_MIN_VALUE is less than LOGICAL_SIGNAL_PROP_MAX_VALUE.

### Default Variables

As a default, the rule checks that the values of all ROUTE_PRIORITY properties are in the 0-99 range.

### Default Severity

Error

### User Customization Information

You can customize the variable macros, reported severity, and error messages for this rule:
**Variable Macros**

| Macro | Definition | Default |
|---|---|---|
| LOGICAL_SIGNAL_PROP_RANGE_NAME | Name of property to check | “ROUTE_PRIORITY” |
| LOGICAL_SIGNAL_PROP_MIN_VALUE | Minimum value for property | “0” |
| LOGICAL_SIGNAL_PROP_MAX_VALUE | Maximum value for property | “99” |

**Reported Severity**

| Macro | Default |
|---|---|
| LOGICAL_SIG_PROP_RANGE_CHECK_SEVERITY | Error |

#### *Error Messages*

STD_SHORT_ERR_LOGICAL_SIG_PROP_RANGE_CHECK
STD_ERR_LOGICAL_SIG_PROP_RANGE_CHECK

## unconnected_biput_pins

Checks each instance in the design to ensure its bidirectional pins are connected to other objects.

### Required Data

None

### Assumptions

None

### Default Variables

None

### Default Severity

Warning

### User Customization Information

You can customize the reported severity and error messages for this rule:
**Reported Severity**

| Macro | Default |
|---|---|
| UNCONNECTED_BIPUT_PINS_SEVERITY | Warning |

#### *Error Messages*

STD_SHORT_ERR_UNCONNECTED_BIPUT_PINS
STD_ERR_UNCONNECTED_BIPUT_PINS

## unconnected_input_pins

Checks each instance in the design to ensure its input pins are connected to other objects.

### Required Data

None

### Assumptions

None

### Default Variables

None

### Default Severity

Warning

### User Customization Information

You can customize the reported severity and error messages for this rule:
**Reported Severity**

| Macro | Default |
|---|---|
| UNCONNECTED_INPUT_PINS_SEVERITY | Warning |

#### *Error Messages*

STD_SHORT_ERR_UNCONNECTED_INPUT_PINS
STD_ERR_UNCONNECTED_INPUT_PINS

## unconnected_instance

Checks each instance in the design to ensure it is connected to another object.

If Rules Checker detects a violation of this rule, it passes information about the body to the `cp.mkr` file.

### Required Data

None

### Assumptions

None

### Default Variables

None

### Default Severity

Warning

### User Customization Information

You can customize the reported severity and error messages for this rule:
**Reported Severity**

| Macro | Default |
|---|---|
| UNCONNECTED_INSTANCE_SEVERITY | Warning |

#### *Error Messages*

STD_SHORT_ERR_UNCONNECTED_INSTANCE
STD_ERR_UNCONNECTED_INSTANCE

## unconnected_output_pins

Checks each instance in the design to ensure its output pins are connected to other objects.

### Required Data

None

### Assumptions

None

### Default Variables

None

### Default Severity

Warning

### User Customization Information

You can customize the reported severity and error messages for this rule:
**Reported Severity**

| Macro | Default |
|---|---|
| UNCONNECTED_OUTPUT_PINS_SEVERITY | Warning |

#### *Error Messages*

STD_SHORT_ERR_UNCONNECTED_OUTPUT_PINS
STD_ERR_UNCONNECTED_OUTPUT_PINS

# Loading I/O Rules

This section contains information on the input/output rules included with Rules Checker. These rules are contained in the `loading_io_checks.rle` file.

Be sure to choose the Logical environment when using these rules.

## check_sign

Checks each signal and issues a violation if any of the following conditions exist:

- INPUT_LOAD property values on all pins are not all positive or all negative.
- OUTPUT_LOAD property values on all pins are not all positive or all negative.
- INPUT_LOAD and OUTPUT_LOAD property values on all pins are all positive or all negative.

**Note:** The INPUT_LOAD and OUTPUT_LOAD property values have the same sign for pins of PSpice library parts. So, the *check_sign* rule is violated.

Attaching the UNKNOWN_LOADING property to a pin cancels the check on all signals attached to the pin. The rule will check the net according to the value of the NO_LOAD_CHECK property:

| | NO_LOAD_CHECK= | | |
|---|---|---|---|
| | **LOW** | **HIGH** | **TRUE/BOTH** |
| **LOW state** | Ignore pin | Check | Ignore pin |
| **HIGH state** | Check | Ignore pin | Ignore pin |

### Required Data

INPUT_LOAD and OUTPUT_LOAD properties must be specified on all pins. This is usually specified in the `chips_prt` file.

### Assumptions

None

### Default Variables

None

### Default Severity

Error

### User Customization Information

You can customize the reported severity and error messages for this rule:
**Reported Severity**

| Macro | Default |
|---|---|
| LOGICAL_CHECK_SIGN_SEVERITY | Error |

### *Error Messages*

STD_SHORT_ERR_LOGICAL_CHECK_SIGN_1 STD_ERR_LOGICAL_CHECK_SIGN_1
STD_SHORT_ERR_LOGICAL_CHECK_SIGN_2 STD_ERR_LOGICAL_CHECK_SIGN_2
STD_SHORT_ERR_LOGICAL_CHECK_SIGN_3 STD_ERR_LOGICAL_CHECK_SIGN_3
STD_SHORT_ERR_LOGICAL_CHECK_SIGN_4 STD_ERR_LOGICAL_CHECK_SIGN_4
STD_SHORT_ERR_LOGICAL_CHECK_SIGN_5 STD_ERR_LOGICAL_CHECK_SIGN_5
STD_SHORT_ERR_LOGICAL_CHECK_SIGN_6 STD_ERR_LOGICAL_CHECK_SIGN_6

## inputio_check

Checks that each signal is connected to at least two pins, and that at least one of the pins is an input pin. It also checks to ensure that a signal connected to a bidirectional pin is also connected to an input or output pin.

To check if two inputs are connected to each other

➤ Use the outputio_check rule.

Attaching the UNKNOWN_LOADING property to a pin cancels the check on all nets attached to the pin. The rule will check the signal according to the value of the NO_IO_CHECK property:

| | NO_IO_CHECK= | | |
|---|---|---|---|
| | **LOW** | **HIGH** | **TRUE/BOTH** |
| **LOW state** | Ignore pin | Check | Ignore pin |
| **HIGH state** | Check | Ignore pin | Ignore pin |

If you are using non-Cadence libraries, you must define GROUND_SYMBOL and POWER_SYMBOL in your `cp_config.h` file.

### Assumptions

The rule does not check signals specified by POWER_SYMBOL and GROUND_SYMBOL.

### Default Variables

None

### Default Severity

Warning

### User Customization Information

You can customize the reported severity and error messages for this rule:
**Reported Severity**

| Macro | Default |
|---|---|
| INPUTIO_CHECK_SEVERITY | Warning |

### *Error Messages*

STD_SHORT_ERR_INPUTIO_CHECK_1 STD_ERR_INPUTIO_CHECK_1
STD_SHORT_ERR_INPUTIO_CHECK_2 STD_ERR_INPUTIO_CHECK_2
STD_SHORT_ERR_INPUTIO_CHECK_3 STD_ERR_INPUTIO_CHECK_3
STD_SHORT_ERR_INPUTIO_CHECK_4 STD_ERR_INPUTIO_CHECK_4

## loading_check

Checks each signal (in high state and low state) for load violations.

The rule does not check signals that have more than two bidirectional pins.

Attaching the UNKNOWN_LOADING property to a pin cancels the check on all signals attached to the pin. The rule will check the signal according to the value of the NO_LOAD_CHECK property:

| | NO_LOAD_CHECK= | | |
|---|---|---|---|
| | **LOW** | **HIGH** | **TRUE/BOTH** |
| **LOW state** | Ignore pin | Check | Ignore pin |
| **HIGH state** | Check | Ignore pin | Ignore pin |

### Required Data

INPUT_LOAD and OUTPUT_LOAD must be specified on all pins in the design.

### Assumptions

None

### Default Variables

None

### Default Severity

Error

### User Customization Information

You can customize the reported severity and error messages for this rule:
**Reported Severity**

| Macro | Default |
|---|---|
| LOGICAL_LOAD_CHECK_SEVERITY | Error |

#### *Error Messages*

STD_SHORT_ERR_LOGICAL_LOAD_CHECK_0 STD_ERR_LOGICAL_LOAD_CHECK_0
STD_SHORT_ERR_LOGICAL_LOAD_CHECK_1 STD_ERR_LOGICAL_LOAD_CHECK_1

## out_check

Checks each signal and issues a violation if all of the following conditions exist:

- The signal does not have an ALLOW_CONNECT property.
- The signal has more than one output or bidirectional pins that do not have an ALLOW_CONNECT property.
- One or more of the pins do not have an OUTPUT_TYPE property, or all pins have an OUTPUT_TYPE property, but their values are not equal.

**Note:** If you want to suppress violations of this rule on a particular signal, attach the ALLOW_CONNECT property to the signal.

### Required Data

If you are using non-Cadence libraries, you must define the OUTPUT_TYPE property for each pin in the `pin/chips_prt` file.

### Assumptions

None

### Default Variables

None

### Default Severity

Error

### User Customization Information

You can customize the reported severity and error messages for this rule:
**Reported Severity**

| Macro | Default |
|---|---|
| OUT_CHECK_SEVERITY | Error |

#### *Error Messages*

STD_SHORT_ERR_OUT_CHECK
STD_ERR_OUT_CHECK

## outputio_check

Checks that each signal is connected to at least two pins, and that at least one of the pins is an output pin. To check if two outputs are connected to each other, use the inputio_check rule.

Attaching the UNKNOWN_LOADING property to a pin cancels the check on all signals attached to the pin. The rule will check the net according to the value of the NO_IO_CHECK property:

| | NO_IO_CHECK= | | |
|---|---|---|---|
| | **LOW** | **HIGH** | **TRUE/BOTH** |
| **LOW state** | Ignore pin | Check | Ignore pin |
| **HIGH state** | Check | Ignore pin | Ignore pin |

### Required Data

If you are using non-Cadence libraries, you must define GROUND_SYMBOL and POWER_SYMBOL in your `cp_config.h` file.

### Assumptions

None

### Default Variables

None

### Default Severity

Oversight

### User Customization Information

You can customize the reported severity and error messages for this rule:
**Reported Severity**

| Macro | Default |
|---|---|
| OUTPUTIO_SEVERITY | Oversight |

***Error Messages***

STD_SHORT_ERR_OUTPUTIO_CHECK_1 STD_ERR_OUTPUTIO_CHECK_1
STD_SHORT_ERR_OUTPUTIO_CHECK_2 STD_ERR_OUTPUTIO_CHECK_2
STD_SHORT_ERR_OUTPUTIO_CHECK_3 STD_ERR_OUTPUTIO_CHECK_3

# Design Guidelines

This section contains information on the general design guideline checks included with Rules Checker. These rules are contained in the `design_guidelines.rle` file.

Be sure to choose the Physical environment when checking these rules.

## check_sign

Checks each signal and issues a violation if any of the following conditions exist:

- INPUT_LOAD property values on all pins are not all positive or all negative.
- OUTPUT_LOAD property values on all pins are not all positive or all negative.
- INPUT_LOAD and OUTPUT_LOAD property values on all pins are all positive or all negative.

**Note:** The INPUT_LOAD and OUTPUT_LOAD property values have the same sign for pins of PSpice library parts. So, the *check_sign* rule is violated.

Attaching the UNKNOWN_LOADING property to a pin cancels the check on all signals attached to the pin. The rule will check the net according to the value of the NO_LOAD_CHECK property:

| | **NO_LOAD_CHECK=** | | |
|---|---|---|---|
| | **LOW** | **HIGH** | **TRUE/BOTH** |
| **LOW state** | Ignore pin | Check | Ignore pin |
| **HIGH state** | Check | Ignore pin | Ignore pin |

### Required Data

INPUT_LOAD and OUTPUT_LOAD properties must be specified on all pins. This is usually specified in the `chips.prt` file.

### Assumptions

None

### Default Variables

None

### Default Severity

Error

### User Customization Information

You can customize the reported severity and error messages for this rule:
**Reported Severity**

| Macro | Default |
|---|---|
| PHYSICAL_CHECK_SIGN_SEVERITY | Error |

### *Error Messages*

STD_SHORT_ERR_PHYSICAL_CHECK_SIGN_1
STD_ERR_PHYSICAL_CHECK_SIGN_1
STD_SHORT_ERR_PHYSICAL_CHECK_SIGN_2
STD_ERR_PHYSICAL_CHECK_SIGN_2
STD_SHORT_ERR_PHYSICAL_CHECK_SIGN_3
STD_ERR_PHYSICAL_CHECK_SIGN_3
STD_SHORT_ERR_PHYSICAL_CHECK_SIGN_4
STD_ERR_PHYSICAL_CHECK_SIGN_4
STD_SHORT_ERR_PHYSICAL_CHECK_SIGN_5
STD_ERR_PHYSICAL_CHECK_SIGN_5
STD_SHORT_ERR_PHYSICAL_CHECK_SIGN_6
STD_ERR_PHYSICAL_CHECK_SIGN_6

## cost_check

Checks that the cost of the parts in your design is less than or equal to the specified dollar (U.S.) amount.

### Required Data

You must specify the COST property for each component used in your design.

### Assumptions

The rule assumes that in the PPT file you have specified the COST property for each component in your design.

#### *Default Variables*

As a default, the rule checks that the cost of the parts is less than or equal to $100 (U.S.).

### Default Severity

Info

### User Customization Information

You can customize the variable macros, reported severity, and error messages for this rule:
**Variable Macros**

| Macro | Definition | Default |
|---|---|---|
| MAXIMUM_COST | Maximum cost ($U.S.) permitted | 100 |

**Reported Severity**

| Macro | Default |
|---|---|
| COST_CHECK_SEVERITY | Info |

***Error Messages***

STD_SHORT_ERR_COST_CHECK
STD_ERR_COST_CHECK

## loading_check

Checks each signal (in high state and low state) for load violations.

The rule does not check signals that have more than two bidirectional pins.

Attaching the UNKNOWN_LOADING property to a pin cancels the check on all nets attached to the pin. The rule will check the net according to the value of the NO_LOAD_CHECK property:

| | NO_LOAD_CHECK= | | |
|---|---|---|---|
| | **LOW** | **HIGH** | **TRUE/BOTH** |
| **LOW state** | Ignore pin | Check | Ignore pin |
| **HIGH state** | Check | Ignore pin | Ignore pin |

### Required Data

INPUT_LOAD and OUTPUT_LOAD properties must be specified on all pins in the design. This is usually done in the `chips_prt` file.

### Assumptions

None

### Default Variables

None

### Default Severity

Error

### User Customization Information

You can customize the reported severity and error messages for this rule:
**Reported Severity**

| Macro | Default |
|---|---|
| PHYSICAL_LOAD_CHECK_SEVERITY | Error |

#### *Error Messages*

STD_SHORT_ERR_PHYSICAL_LOAD_CHECK_0
STD_ERR_PHYSICAL_LOAD_CHECK_0
STD_SHORT_ERR_PHYSICAL_LOAD_CHECK_1
STD_ERR_PHYSICAL_LOAD_CHECK_1

## max_power_check

Checks that the power dissipation of your design is within a specified limit.

### Required Data

You must specify the POWER_DISSIPATION property for each component used in your design.

### Assumptions

You have specified the POWER_DISSIPATION property for each component in your design in ptf file/on component.

### Default Variables

Maximum specified power is 10 watts.

### Default Severity

Info

### User Customization Information

You can customize the variable macros, reported severity, and error messages for this rule:
**Variable Macros**

| Macro | Definition | Default |
|---|---|---|
| MAXIMUM_POWER_ DISSIPATION | Maximum power dissipation allowed | 10 |
| POWER_DISSIPATION | Used to specify power for each component | “POWER” |

**Reported Severity**

| Macro | Default |
|---|---|
| MAX_POWER_CHECK_SEVERITY | Info |

***Error Messages***

STD_SHORT_ERR_MAX_POWER_CHECK STD_ERR_MAX_POWER_CHECK

## phys_unconnected_pins

Checks for unconnected pins on each packaged body in your design.

**Note:** This rule requires that you specify the Physical environment and that you run the Packager before you check your design.

### Required Data

None

### Assumptions

None

### Default Variables

None

### Default Severity

Warning

### User Customization Information

You can customize the reported severity and error messages for this rule:
**Reported Severity**

| Macro | Default |
|---|---|
| PHYS_UNCONNECTED_PINS_SEVERITY | Warning |

#### *Error Messages*

STD_SHORT_ERR_PHYS_UNCONNECTED_PINS
STD_ERR_PHYS_UNCONNECTED_PINS

# Jedec Rules

These rules check if JEDEC_TYPE properties on the design match those defined in either the `chips_prt` file of the ptf file. These rules are contained in the `jedec.rle` file.

Be sure to choose the Physical environment when checking these rules.

## alt_sym_check_class

Checks whether a subclass of the ALT_SYMBOLS property is a valid subclass.

### Required Data

You must specify the valid subclasses.

### Assumptions

The rule assumes that the value of the ALT_SYMBOLS property is specified as:

```
ALT_SYMBOLS = (Subclass:Symbol,Symbol,...;Subclass:Symbol,Symbol,...)
```

where:

Subclass is either TOP (or T) for top layers, or BOTTOM (or B) for bottom layers. TOP is assumed if no subclass is specified.

Symbol is a standard value for JEDEC_TYPE. Be sure to separate each `Symbol` value with a comma.

**Note:** Open/close parenthesis () are required. The following error occurs if they are omitted

```
Error in ALT_SYMBOLS property for device '<device_name>':'Encountered an error
while parsing alternate symbol.'
```

### Default Variables

None

### Default Severity

Error

### User Customization Information

You can customize the reported severity and error messages for this rule
:

**Reported Severity**

| Macro | Default |
|---|---|
| ALT_SYM_CHECK_CLASS_SEVERITY | Error |

### *Error Messages*

STD_SHORT_ERR_ALT_SYM_CHECK_CLASS STD_ERR_ALT_SYM_CHECK_CLASS

## alt_sym_check_value

Checks whether the symbol value of the ALT_SYMBOLS property is one of the specified JEDEC_TYPE values.

### Required Data

You must use the JEDEC_TYPE_VALUES macro in the `jedec.h` file to specify valid JEDEC_TYPE values. By default, JEDEC_TYPE_VALUES uses Cadence-defined values (CADENCE_DEFINED_JEDEC_TYPE_VALUES) and user-defined values (USER_DEFINED_JEDEC_TYPE_VALUES).

### Assumptions

The rule assumes that the value of the ALT_SYMBOLS property is specified as:

```
ALT_SYMBOLS = (Subclass:Symbol,Symbol,...;Subclass:Symbol,Symbol,...)
```

where:

`Subclass` is either TOP (or T) for top layers, or BOTTOM (or B) for bottom layers. TOP is assumed if no subclass is specified.

`Symbol` is a standard value for JEDEC_TYPE. Be sure to separate each `Symbol` value with a comma.

**Note:** Open/close parenthesis () are required. The following error occurs if they are omitted

```
Error in ALT_SYMBOLS property for device '<device_name>':'Encountered an error
while parsing alternate symbol.'
```

### Default Variables

CADENCE_DEFINED_JEDEC_TYPE_VALUES (See `jedec.h` for a listing of values specified by this macro.)

### Default Severity

Error

### User Customization Information

You can customize the JEDEC_TYPE values, reported severity and error messages for this rule:

### *JEDEC_TYPE Values*

Use the USER_DEFINED_JEDEC_TYPE_VALUES macro (see `jedec.h`) to specify your own JEDEC_TYPE values

.

**Reported Severity**

| Macro | Default |
|---|---|
| ALT_SYM_CHECK_VALUE_SEVERITY | Error |

### *Error Messages*

STD_SHORT_ERR_ALT_SYM_CHECK_VALUE STD_ERR_ALT_SYM_CHECK_VALUE

## alt_sym_missing_parens

Checks the syntax (opening and closing parenthesis) of the symbol value of the ALT_SYMBOLS property.

### Required Data

None

### Assumptions

The rule assumes that the value of the ALT_SYMBOLS property is specified as:

```
ALT_SYMBOLS = (Subclass:Symbol,Symbol,...;Subclass:Symbol,Symbol,...)
```

where:

`Subclass` is either TOP (or T) for top layers, or BOTTOM (or B) for bottom layers. TOP is assumed if no subclass is specified.

`Symbol` is a standard value for JEDEC_TYPE. Be sure to separate each `Symbol` value with a comma.

### Default Variables

None

### Default Severity

Error

### User Customization Information

You can customize the reported severity and error messages for this rule
:
**Reported Severity**

| Macro | Default |
|---|---|
| ALT_SYM_MISSING_PARENS_SEVERITY | Error |

#### *Error Messages*

STD_SHORT_ERR_ALT_SYM_MISSING_PARENS
STD_ERR_ALT_SYM_CHECK_MISSING_PARENS

## jedec_type_exist_check

Checks that a JEDEC_TYPE property exists on each instance or its body after packaging.

### Required Data

None

### Assumptions

None

### Default Variables

None

### Default Severity

Error

### User Customization Information

You can customize the reported severity and error messages for this rule
:

**Reported Severity**

| Macro | Default |
|---|---|
| JEDEC_TYPE_EXIST_CHECK | Error |

#### *Error Messages*

STD_SHORT_ERR_JEDEC_TYPE_EXIST_CHECK
STD_ERR_JEDEC_TYPE_EXIST_CHECK

## jedec_type_match_check

Checks whether the JEDEC_TYPE property on each packaged instance matches the value on the chip.

### Required Data

None

### Assumptions

The drawing has been packaged using Packager-XL.

### Default Variables

None

### Default Severity

Warning

### User Customization Information

You can customize the reported severity and error messages for this rule:
**Reported Severity**

| Macro | Default |
|---|---|
| JEDEC_TYPE_MATCH_CHECK | Warning |

#### *Error Messages*

STD_SHORT_ERR_JEDEC_TYPE_MATCH_CHECK
STD_ERR_JEDEC_TYPE_MATCH_CHECK

# Net Name Rules

This section contains information on the signal name rules included with Rules Checker. These rules are contained in the `net_name_checks.rle` file.

Be sure to choose the Logical environment when using these rules.

## multiple_signames

Utility that reports any nets with multiple signal names that are synonymed. You can use this to check if any pairs of nets in your design are shorted.

### Required Data

None

### Assumptions

None

### Default Variables

None

### Default Severity

Oversight

### User Customization Information

You can customize the reported severity and error messages for this rule:
**Reported Severity**

| Macro | Default |
|---|---|
| LOGICAL_MULTIPLE_SIGNAMES_SEVERITY | Oversight |

#### *Error Messages*

STD_SHORT_ERR_LOGICAL_MULTIPLE_SIGNAMES
STD_ERR_LOGICAL_MULTIPLE_SIGNAMES

## named_single_page_net

Checks that if a signal is on a single page, it is not named.

### Required Data

None

### Assumptions

The rule does not check signals (even if they are local) connected to an instance with a CONN_PROP_NAME property that has a value from CONN_PROP_VALUE.

### Default Variables

None

### Default Severity

Warning

### User Customization Information

You can customize the global macros, reported severity and error messages for this rule:
**Global Macros**

| Macro | Description |
|---|---|
| CONN_PROP_NAME | Specifies the property used to look for a connector, for example, BODY_TYPE |
| CONN_PROP_VALUE | Specifies the value assigned to the property CONN_PROP_NAME |

### Reported Severity

| Macro | Default |
|---|---|
| NAMED_SINGLE_PAGE_NET_SEVERITY | Warning |

#### *Error Messages*

STD_SHORT_ERR_NAMED_SINGLE_PAGE_NET
STD_ERR_NAMED_SINGLE_PAGE_NET

## single_node_net

Checks that every signal has at least two nodes (pins) attached to it.

### Required Data

None

### Assumptions

None

### Default Variables

None

### Default Severity

Error

### User Customization Information

You can customize the reported severity and error messages for this rule:
**Reported Severity**

| Macro | Default |
|---|---|
| SINGLE_NODE_NET_SEVERITY | Warning |

#### *Error Messages*

STD_SHORT_ERR_SINGLE_NODE_NET STD_ERR_SINGLE_NODE_NET

# Preferred Parts Rules

This section contains information on the preferred parts rules included with Rules Checker. These rules are contained in the `preferred_parts.rle` file.

Be sure to choose the Physical environment when using these rules.

## invalid_pref_part_value

Checks that a user-specified property on a part has a value other than the user-specified list of preferred values and non-preferred values.

### Required Data

None

### Assumptions

None

### Default Variables

As a default, the rule checks for parts that have a STATUS property with a value other than the preferred value PREF and the non-preferred value NONPREF.

### Default Severity

Error

### User Customization Information

You can customize the variable macros, reported severity, and error messages for this rule:
**Variable Macros**

| Macro | Definition | Default |
|---|---|---|
| PREFERRED_PROP_NAME | Name of property to check | STATUS |
| PREFERRED_PROP_VALUE | Preferred value for property | PREF |
| NONPREFERRED_PROP_ VALUE | Non-preferred value for property | NONPREF |

**Reported Severity**

| Macro | Default |
|---|---|
| INVALID_PREF_PART_VALUE_SEVERITY | Error |

#### *Error Messages*

STD_SHORT_ERR_INVALID_PREF_PART_VALUE
STD_ERR_INVALID_PREF_PART_VALUE

## non_preferred_part

Checks for parts with non-preferred values of the specified property.

### Required Data

None

### Assumptions

None

### Default Variables

As a default, the rule checks for parts that have a STATUS property with a value of NONPREF.

### Default Severity

Warning

### User Customization Information

You can customize the variable macros, reported severity, and error messages for this rule:
**Variable Macros**

| Macro | Definition | Default |
|---|---|---|
| PREFERRED_PROP_NAME | Name of property to check | STATUS |
| NONPREFERRED_PROP_VALUE | Non-preferred property values for parts | NONPREF |

**Reported Severity**

| Macro | Default |
|---|---|
| NON_PREFERRED_PART_SEVERITY | Warning |

#### *Error Messages*

STD_SHORT_ERR_NON_PREFERRED_PART STD_ERR_NON_PREFERRED_PART

# Body Cross View Checks

This section contains information on the rules included with Rules Checker. These rules are contained in the `body_cross_view_checks.rle` file.

Be sure to choose the Body environment when using these rules.

## body_to_logic_check

Checks that each pin in the body drawing also exists in the logic drawing.

### Required Data

None

### Assumptions

None

### Default Variables

As a default, the rule checks the version 1 drawing for pins labeled \I.

### Default Severity

Error

### User Customization Information

You can customize the variable macros, reported severity, and error messages for this rule:
**Variable Macros**

| Macro Name | Definition | Default |
|---|---|---|
| LOGIC_VERSION | Version of logic drawing that the body drawing will be checked against. | “1” |
| INTERFACE_SIGNAL_ MACRO | Extension that specifies interface signals | “\I” |

**Reported Severity**

| Macro | Default |
|---|---|
| BODY_TO_LOGIC_CHECK_SEVERITY | Error |

### *Error Messages*

STD_SHORT_ERR_BODY_TO_LOGIC_CHECK
STD_ERR_BODY_TO_LOGIC_CHECK

## body_to_physical_check

Finds pins whose names do not have corresponding entries in the `chips_prt` file.

### Required Data

None

### Assumptions

None

### Default Variables

None

### Default Severity

Error

### User Customization Information

You can customize the reported severity and error messages for this rule:
**Reported Severity**

| Macro | Default |
|---|---|
| BODY_TO_PHYSICAL_CHECK_SEVERITY | Error |

### *Error Messages*

STD_SHORT_ERR_BODY_TO_PHYSICAL_CHECK
STD_ERR_BODY_TO_PHYSICAL_CHECK

## body_to_verilog_check

Verifies that the bodypin(s), PIN_MAP END_PIN in verilog_map (if present) and port names in the Verilog module are all in synchronization.

### Required Data

None

### Assumptions

The low assertion character for the pin-name is converted to an underscore located at end of pin-name.

Example

A pin name A<3..0>* is converted to A_[3:0].

### Default Variables

None

### Default Severity

Error

### User Customization Information

You can customize the reported severity and error messages for this rule:
**Reported Severity**

| Macro | Default |
|---|---|
| BODY_TO_VERILOG_CHECK_SEVERITY | Error |

#### *Error Messages*

STD_SHORT_ERR_BODY_TO_VERILOG_CHECK1

STD_ERR_BODY_TO_VERILOG_CHECK0

STD_ERR_BODY_TO_VERILOG_CHECK1

STD_SHORT_ERR_BODY_TO_VERILOG_CHECK2

STD_ERR_BODY_TO_VERILOG_CHECK2

STD_SHORT_ERR_BODY_TO_VERILOG_CHECK3

STD_ERR_BODY_TO_VERILOG_CHECK3

STD_SHORT_ERR_BODY_TO_VERILOG_CHECK4

STD_ERR_BODY_TO_VERILOG_CHECK4

STD_ERR_BODY_TO_VERILOG_CHECK5

## input_pin_port_dir_check

Check that input pin in body is declared as input port in Verilog module.

### Required Data

None

### Assumptions

The low assertion character for the pin-name is converted to an underscore located at end of pin-name.

Example

A pin name A<3..0>* is converted to A_[3:0].

### Default Variables

None

### Default Severity

Error

### User Customization Information

You can customize the reported severity and error messages for this rule:
**Reported Severity**

| Macro | Default |
|---|---|
| PIN_PORT_DIR_CHECK_SEVERITY | Error |

#### *Error Messages*

STD_SHORT_ERR_PIN_PORT_DIR_CHECK
STD_ERR_PIN_PORT_DIR_CHECK

## inout_pin_port_dir_check

Checks that inout pin in body is declared as inout port in Verilog model.

### Required Data

None

### Assumptions

The low assertion character for the pin-name is converted to an underscore located at end of pin-name.

Example

A pin name A<3..0>* is converted to A_[3:0].

### Default Variables

None

### Default Severity

Error

### User Customization Information

You can customize the reported severity and error messages for this rule:
**Reported Severity**

| Macro | Default |
|---|---|
| PIN_PORT_DIR_CHECK_SEVERITY | Error |

#### *Error Messages*

STD_SHORT_ERR_PIN_PORT_DIR_CHECK
STD_ERR_PIN_PORT_DIR_CHECK

## invalid_part_name

Checks that physical components (except those without a physical packaging, with a logic drawing, or in a user-defined list) have valid PART_NAME values in the `chips_prt` file. Property PART_NAME from `chips_prt` is checked against PART_NAME property from part file.If part file not present that PART_NAME from `chips_prt` is checked with the name in <*`library`*>`.lib` file in If PART_NAME in `chips_prt` is not valid then Rules Checker highlights the body of the component.

### Required Data

If you are going to specify bodies to ignore, you must specify them via the USER_DEFINED_BODIES_TO_IGNORE_FOR_BODY_ PART_NAME_CHECK macro in `body_cross_view_checks.h`.

### Assumptions

The rule assumes that the body is non-PLUMBING and not a COMMENT body.

### Default Variables

The rule does not check the following bodies: DRAWING, DECLARATIONS, HDL_DECS, VHDL_DECS, VERILOG_DECS

### Default Severity

Error

### User Customization Information

You can customize the variable macros, reported severity, and error messages for this rule:
**Variable Macros**

| Macro | Definition | Default |
|---|---|---|
| USER_DEFINED_BODIES_TO_IGNORE_FOR_BODY_PART_NAME_CHECK | list of bodies not to check | “DRAWING,” “DECLARATIONS,” “HDL_DECS,” “VHDL_DECS,” “VERILOG_DECS” |

### Reported Severity

| Macro | Default |
|---|---|
| BODY_INVALID_PART_NAME_SEVERITY | Error |

#### *Error Messages*

STD_SHORT_ERR_BODY_INVALID_PART_NAME
STD_ERR_BODY_INVALID_PART_NAME

## logic_to_body_check

Checks that each pin in the logic drawing also exists in the body drawing.

### Required Data

None

### Assumptions

None

### Default Variables

As a default, the rule checks that “\I” signals in the version 1 logic drawing also exist in the body drawing.

### Default Severity

Error

### User Customization Information

You can customize the variable macros, reported severity, and error messages for this rule:
**Variable Macros**

| Macro | Definition | Default |
|---|---|---|
| INTERFACE_SIGNAL_ MACRO | Extension that specifies interface signals | “\I” |
| LOGIC_VERSION | Version of logic drawing that body drawing will be checked against. | 1 |

**Reported Severity**

| Macro | Default |
|---|---|
| LOGIC_TO_BODY_CHECK_SEVERITY | Error |

#### *Error Messages*

STD_SHORT_ERR_LOGIC_TO_BODY_CHECK
STD_ERR_LOGIC_TO_BODY_CHECK

## output_pin_port_dir_check

Checks that output pin in body is declared as output port in Verilog model.

### Required Data

None

### Assumptions

The low assertion character for the pin-name is converted to an underscore located at end of pin-name.

Example

A pin name A<3..0>* is converted to A_[3:0].

### Default Variables

None

### Default Severity

Error

### User Customization Information

You can customize the reported severity and error messages for this rule:
**Reported Severity**

| Macro | Default |
|---|---|
| PIN_PORT_DIR_CHECK_SEVERITY | Error |

#### *Error Messages*

STD_SHORT_ERR_PIN_PORT_DIR_CHECK
STD_ERR_PIN_PORT_DIR_CHECK

## physical_to_body_check

Checks that each pin name listed in the `chips_prt` file exists in the body drawing.

### Required Data

None

### Assumptions

None

### Default Variables

None

### Default Severity

Error

### User Customization Information

You can customize the reported severity and error messages for this rule:
**Reported Severity**

| Macro | Default |
|---|---|
| PHYSICAL_TO_BODY_CHECK_SEVERITY | Error |

#### *Error Messages*

STD_SHORT_ERR_PHYSICAL_TO_BODY_CHECK
STD_ERR_PHYSICAL_TO_BODY_CHECK

## property_parameter_check

Verifies that the Property...End Property section of `verilog_map` file is consistent with `parameter(s) in the Verilog module.

### Required Data

None

### Assumptions

None

### Default Variables

None

### Default Severity

Error

### User Customization Information

You can customize the reported severity and error messages for this rule:
**Reported Severity**

| Macro | Default |
|---|---|
| PROP_PARAM_CHECK_SEVERITY | Error |

#### *Error Messages*

STD_SHORT_ERR_PROP_PARAM_CHECK
STD_ERR_PROP_PARAM_CHECK

# Body Drawing Checks

This section contains information on the body drawing rules included with Rules Checker. These rules are contained in the `body_drawing_checks.rle` file.

Be sure to choose the Body environment when using these rules.

## body_exceeds_max_size

Checks whether the size of a body is within the specified maximum height and width values.

### Required Data

None

### Assumptions

None

### Default Variables

The rules uses 2000 as the maximum height and width values, and ignores the following bodies: A SIZE PAGE, B SIZE PAGE, C SIZE PAGE, D SIZE PAGE.

### Default Severity

Error

### User Customization Information

You can customize the variable macros, reported severity, and error messages for this rule:
**Variable Macros**

| Macro | Definition | Default |
|---|---|---|
| MAX_SPECIFIED_HEIGHT | Maximum allowed body height | 2000 |
| MAX_SPECIFIED_WIDTH | Maximum allowed body width | 2000 |
| USER_DEFINED_BODIES_TO_IGNORE_FOR_MAX_SIZE | User-defined list of bodies not to check | A SIZE PAGE,” “B SIZE PAGE,” “C SIZE PAGE,” “D SIZE PAGE” |

**Reported Severity**

| Macro | Default |
|---|---|
| BODY_EXCEEDS_MAX_SIZE_SEVERITY | Error |

#### *Error Messages*

STD_SHORT_ERR_BODY_EXCEEDS_MAX_SIZE
STD_ERR_BODY_EXCEEDS_MAX_SIZE

## body_less_than_min_size

Checks whether the size of a body (except those specified on a user-defined list) is smaller than the specified minimum height and width values.

### Required Data

None

### Assumptions

None

### Default Variables

The rules checks for bodies with height and width less than 5 units and ignores DRAWING bodies (5 units = .01 inch).

### Default Severity

Error

### User Customization Information

You can customize the variable macros, reported severity, and error messages for this rule:
**Variable Macros**

| Macro | Definition | Default |
|---|---|---|
| MIN_SPECIFIED_HEIGHT | Minimum allowed body height | 5 |
| MIN_SPECIFIED_WIDTH | Minimum allowed body width | 5 |
| USER_DEFINED_BODIES_TO_IGNORE_FOR_MIN_SIZE | User-defined list of bodies not to check | “DRAWING” |

### Reported Severity

| Macro | Default |
|---|---|
| BODY_LESS_THAN_MIN_SIZE_SEVERITY | Error |

### *Error Messages*

STD_SHORT_ERR_BODY_LESS_THAN_MIN_SIZE
STD_ERR_BODY_LESS_THAN_MIN_SIZE

## color_check

Checks that only user-specified colors are used on notes, signals, properties, and arcs in the design.

### Required Data

None

### Assumptions

None

### Default Variables

As a default, the rule checks to make sure the colors of all notes, segments, properties and arcs are one of the following colors:

Mono, Red, Green, Blue, Yellow, Orange, Salmon, Violet, Brown, SkyBlue, White, Peach, Pink, Purple, Aqua, Gray

### Default Severity

Warning

### User Customization Information

You can customize the variable macros, reported severity, and error messages for this rule:
**Variable Macros**

| Macro | Definition | Default |
|---|---|---|
| USER_DEFINED_COLORS_FOR_BODY_NOTE | Colors allowed on notes | “Mono”, “Red”, “Green”, “Blue”, “Yellow”, “Orange” |

**Variable Macros**

| Macro | Definition | Default |
|---|---|---|
| USER_DEFINED_COLORS_FOR_BODY_SEGS | Colors allowed on segments | Brown," "Sky Blue," "White," "Peach," |
| USER_DEFINED_COLORS_FOR_BODY_PROP | Colors allowed on properties | "Purple", "Aqua", "Gray" |
| USER_DEFINED_COLORS_FOR_BODY_ARC | Colors allowed on arcs | |

**Reported Severity**

| Macro | Default |
|---|---|
| BODY_COLOR_CHECK_SEVERITY | Warning |

***Error Messages***

STD_SHORT_ERR_BODY_COLOR_CHECK STD_ERR_BODY_COLOR_CHECK

## invisible_prop_location

Highlights invisible properties attached to the Origin that are not placed within a specified distance from the Origin.

**Required Data**

None

**Assumptions**

The rule only checks properties of Origin, in which both the name and the value are invisible.

The property attached to the Origin is the property in the design not attached to segments/ bodypin of design.

### Default Variables

As a default, the rule checks to ensure all invisible properties attached to the Origin are placed within 500 units from the Origin (500 units = 1 inch).

### Default Severity

Error

### User Customization Information

You can customize the variable macros, reported severity, and error messages for this rule:
**Variable Macros**

| Macro | Definition | Default |
|---|---|---|
| MAX_PROP_DIST_FROM_ ORIGIN | Maximum distance allowed between an invisible property and the origin | 1000 |

**Reported Severity**

| Macro | Default |
|---|---|
| INVISIBLE_PROP_LOCATION_SEVERITY | Error |

#### *Error Messages*

STD_SHORT_ERR_INVISIBLE_PROP_LOCATION
STD_ERR_INVISIBLE_PROP_LOCATION

## non_centered_origin

Checks that the Origin of the drawing lies within the body, and that the center of the drawing is within the specified distance from the Origin.

### Required Data

None

### Assumptions

None

### Default Variables

As a default, the rule checks to ensure the centers of the Origin and the body drawing are within 250 units from each other (250 units = 0.5 inch).

### Default Severity

Info

### User Customization Information

You can customize the variable macros, reported severity, and error messages for this rule:
**Variable Macros**

| Macro | Definition | Default |
|---|---|---|
| MAX_ALLOWED_ORIGIN_OFFSET | Maximum distance allowed between Origin and center of body drawing. | 250 |

**Reported Severity**

| Macro | Default |
|---|---|
| NON_CENTERED_ORIGIN_SEVERITY | Info |

#### *Error Messages*

STD_SHORT_ERR_NON_CENTERED_ORIGIN1
STD_ERR_NON_CENTERED_ORIGIN1
STD_SHORT_ERR_NON_CENTERED_ORIGIN2
STD_ERR_NON_CENTERED_ORIGIN2

## prop_note_overlap

Checks that visible properties (other than those specified to ignore) do not overlap other visible notes in the design.

### Required Data

You must use the USER_DEFINED_PROPERTIES_TO_IGNORE macro to specify properties to ignore in the check.

### Assumptions

It is assumed that USER_DEFINED_PROPS_TO_IGNORE includes the BUBBLE_GROUP and BUBBLED properties, as the body file has no information on the placement of these properties.

### Default Variables

None

### Default Severity

Error

### User Customization Information

You can customize the variable macros, reported severity and error messages for this rule:
**Variable Macros**

| Macro | Definition | Default |
|---|---|---|
| USER_DEFINED_PROPS_TO_IGNORE | User-defined list of properties ignored by the rule | (None) |

### Reported Severity

| Macro | Default |
|---|---|
| PROP_NOTE_OVERLAP_SEVERITY | Error |

### *Error Messages*

STD_SHORT_ERR_PROP_NOTE_OVERLAP STD_ERR_PROP_NOTE_OVERLAP

## prop_seg_overlap

Checks that visible properties (other than those specified to ignore) do not overlap other visible segments in the design.

### Required Data

You must use the USER_DEFINED_PROPERTIES_TO_IGNORE macro to specify properties to ignore in the check.

### Assumptions

It is assumed that USER_DEFINED_PROPS_TO_IGNORE includes the BUBBLE_GROUP and BUBBLED properties, as the body file has no information on the placement of these properties.

### Default Variables

None

### Default Severity

Error

### User Customization Information

You can customize the variable macros, reported severity, and error messages for this rule:
**Variable Macros**

| Macro | Definition | Default |
|---|---|---|
| USER_DEFINED_PROPS_TO_IGNORE | User-defined list of properties ignored by the rule. | (None) |

### Reported Severity

| Macro | Default |
|---|---|
| PROP_SEG_OVERLAP_SEVERITY | Error |

### *Error Messages*

STD_SHORT_ERR_PROP_SEG_OVERLAP STD_ERR_PROP_SEG_OVERLAP

## props_overlap

Checks that visible properties (other than those specified to ignore) do not overlap in the design.

### Required Data

You must use the USER_DEFINED_PROPERTIES_TO_IGNORE macro to specify properties to ignore in the check.

### Assumptions

It is assumed that USER_DEFINED_PROPS_TO_IGNORE includes the BUBBLE_GROUP and BUBBLED properties, as the body file has no information on the placement of these properties.

### Default Variables

None

### Default Severity

Error

### User Customization Information

You can customize the variable macros, reported severity, and error messages for this rule:
**Variable Macros**

| Macro | Definition | Default |
|---|---|---|
| USER_DEFINED_PROPS_TO_IGNORE | User-defined list of properties ignored by the rule | (None) |

**Reported Severity**

| Macro | Default |
|---|---|
| PROPS_OVERLAP_SEVERITY | Error |

***Error Messages***

STD_SHORT_ERR_PROPS_OVERLAP STD_ERR_PROPS_OVERLAP

# Body Pin Checks

This section contains information on the body pin rules included with Rules Checker. These rules are contained in the `body_pin_checks.rle` file.

Be sure to choose the Body environment when using these rules.

## biput_pin_wrong_orient

Finds biput pins in the design whose orientation is not specified in a user-defined list.

**Required Data**

None

**Assumptions**

The rule only checks non-passthru biput pins.

**Default Variables**

As a default, the rule considers biput pins with EAST orientations to be valid.

**Default Severity**

Warning

### User Customization Information

You can customize the variable macros, reported severity, and error messages for this rule:
**Variable Macros**

| Macro | Definition | Default |
|---|---|---|
| BIPUT_ ORIENTATIONS | Orientation values allowed for biput pins in the design | “EAST” |

**Reported Severity**

| Macro | Default |
|---|---|
| BIPUT_PIN_WRONG_ORIENT_SEVERITY | Warning |

***Error Messages***

STD_SHORT_ERR_BIPUT_PIN_WRONG_ORIENT
STD_ERR_BIPUT_PIN_WRONG_ORIENT

## bottom_pins_incorrect_spacing

Reports pins with orientation SOUTH that are placed less than a specified minimum distance from each other in the drawing.

### Required Data

None

### Assumptions

None

### Default Variables

As a default, the rule checks for pins placed less than 50 units from each other (50 units = .1 inch).

### Default Severity

Error

### User Customization Information

You can customize the variable macros, reported severity, and error messages for this rule:
**Variable Macros**

| Macro | Definition | Default |
|---|---|---|
| MIN_DISTANCE_BETWEEN_PINS | Minimum distance allowed between pins with the same orientation | 50 |

**Reported Severity**

| Macro | Default |
|---|---|
| BOTTOM_PINS_INCORRECT_SPACING_SEVERITY | Error |

#### *Error Messages*

STD_SHORT_ERR_BOTTOM_PINS_INCORRECT_SPACING
STD_ERR_BOTTOM_PINS_INCORRECT_SPACING

## input_pin_wrong_orient

Reports input pins in the design whose orientations are not specified in a user-defined list.

### Required Data

None

### Assumptions

The rule only checks non-passthru input pins.

### Default Variables

As a default, the rule checks to make sure all input pins have orientations of SOUTH or WEST.

### Default Severity

Warning

### User Customization Information

You can customize the variable macros, reported severity, and error messages for this rule:
**Variable Macros**

| Macro | Definition | Default |
|---|---|---|
| INPUT_ ORIENTATIONS | Orientation values allowed for input pins in the design | “SOUTH,” “WEST” |

**Reported Severity**

| Macro | Default |
|---|---|
| INPUT_PIN_WRONG_ORIENT_SEVERITY | Warning |

#### *Error Messages*

STD_SHORT_ERR_INPUT_PIN_WRONG_ORIENT
STD_ERR_INPUT_PIN_WRONG_ORIENT

## invalid_passthru_pin

Highlights passthru body pins whose names are not specified in a user-defined list.

### Required Data

None

### Assumptions

None

### Default Variables

As a default, the rule checks to ensure that all passthru body pins are of the following types: CLK, SET, OE, E*.

### Default Severity

Info

### User Customization Information

You can customize the variable macros, reported severity, and error messages for this rule:
**Variable Macros**

| Macro | Definition | Default |
|---|---|---|
| USER_DEFINED_PASSTHRU_BODYPINS_LIST | List containing names of allowed passthru pins | “CLK”, “SET”, “OE”, “E*” |

**Reported Severity**

| Macro | Default |
|---|---|
| INVALID_PASSTHRU_PIN_SEVERITY | Info |

#### *Error Messages*

STD_SHORT_ERR_INVALID_PASSTHRU_PIN STD_ERR_INVALID_PASSTHRU_PIN

## invalid_top_bottom_pins

Finds pins with NORTH or SOUTH orientations that are not specified as control pins from a user-defined list.

### Required Data

None

### Assumptions

None

### Default Variables

As a default, the rule considers pins with the following labels to be valid control pins: CLK, SET, OE, E*.

### Default Severity

Info

### User Customization Information

You can customize the variable macros, reported severity, and error messages for this rule:
**Variable Macros**

| Macro | Definition | Default |
|---|---|---|
| USER_DEFINED_CONTROL_BODYPINS | List of control pins | “CLK,” “SET,” “OE,” “E*” |

**Reported Severity**

| Macro | Default |
|---|---|
| INVALID_TOP_BOTTOM_PINS_SEVERITY | Info |

#### *Error Messages*

STD_SHORT_ERR_INVALID_TOP_BOTTOM_PINS
STD_ERR_INVALID_TOP_BOTTOM_PINS

## left_pins_incorrect_spacing

Reports pins with orientation WEST that are placed less than a specified minimum distance from each other in the drawing.

### Required Data

None

### Assumptions

None

### Default Variables

As a default, the rule checks to make sure that all pins oriented WEST are placed within at least 50 units of each other (50 units = .1 inch).

### Default Severity

Error

### User Customization Information

You can customize the variable macros, reported severity, and error messages for this rule:
**Variable Macros**

| Macro | Definition | Default |
|---|---|---|
| MIN_DISTANCE_BETWEEN_PINS | Minimum distance allowed between pins with the same orientation | 50 |

**Reported Severity**

| Macro | Default |
|---|---|
| LEFT_PINS_INCORRECT_SPACING_SEVERITY | Error |

#### *Error Messages*

STD_SHORT_ERR_LEFT_PINS_INCORRECT_SPACING
STD_ERR_LEFT_PINS_INCORRECT_SPACING

## misaligned_passthru_pin

Reports each passthru pin that is not aligned with its corresponding visible pin.

### Required Data

None

### Assumptions

None

### Default Variables

None

### Default Severity

Info

### User Customization Information

You can customize the reported severity and error messages for this rule:
**Reported Severity**

| Macro | Default |
|---|---|
| MISALIGNED_PASSTHRU_PIN_SEVERITY | Info |

#### *Error Messages*

STD_SHORT_ERR_MISALIGNED_PASSTHRU_PIN
STD_ERR_MISALIGNED_PASSTHRU_PIN

## output_pin_wrong_orient

Reports output pins in the design whose orientations are not specified in a user-defined list.

### Required Data

None

### Assumptions

The rule only checks non-passthru output pins.

### Default Variables

As a default, the rule checks to ensure all output pins have EAST orientation.

### Default Severity

Warning

### User Customization Information

You can customize the variable macros, reported severity, and error messages for this rule:
**Variable Macros**

| Macro | Definition | Default |
|---|---|---|
| OUTPUT_ ORIENTATIONS | Orientation values allowed for output pins in the design | “EAST” |

**Reported Severity**

| Macro | Default |
|---|---|
| OUTPUT_PIN_WRONG_ORIENT_SEVERITY | Warning |

#### *Error Messages*

STD_SHORT_ERR_OUTPUT_PIN_WRONG_ORIENT
STD_ERR_OUTPUT_PIN_WRONG_ORIENT

## right_pin_wrong_orient

Reports pins with orientation EAST that are placed less than a specified minimum distance from each other in the drawing.

### Required Data

None

### Assumptions

None

### Default Variables

As a default, the rule checks that all pins with EAST orientation are placed at least 50 units from each other on the drawing (50 units = .1 inch).

### Default Severity

Error

### User Customization Information

You can customize the variable macros, reported severity, and error messages for this rule:
**Variable Macros**

| Macro | Definition | Default |
|---|---|---|
| MIN_DISTANCE_BETWEEN_PINS | Minimum distance allowed between pins with the same orientation | 50 |

**Reported Severity**

| Macro | Default |
|---|---|
| RIGHT_PINS_INCORRECT_SPACING_SEVERITY | Error |

### *Error Messages*

STD_SHORT_ERR_RIGHT_PINS_INCORRECT_SPACING
STD_ERR_RIGHT_PINS_INCORRECT_SPACING

## top_pins_incorrect_spacing

Reports pins with orientation NORTH that are placed less than a specified minimum distance from each other in the drawing.

### Required Data

None

### Assumptions

None

### Default Variables

As a default, the rule checks that all pins with NORTH orientation are placed at least 50 units from each other on the drawing (50 units = .1 inch).

### Default Severity

Error

### User Customization Information

You can customize the variable macros, reported severity, and error messages for this rule:
**Variable Macros**

| Macro | Definition | Default |
|---|---|---|
| MIN_DISTANCE_BETWEEN_PINS | Minimum distance allowed between pins with the same orientation | 50 |

**Reported Severity**

| Macro | Default |
|---|---|
| TOP_PINS_INCORRECT_SPACING_SEVERITY | Error |

***Error Messages***

STD_SHORT_ERR_TOP_PINS_INCORRECT_SPACING
STD_ERR_TOP_PINS_INCORRECT_SPACING

# Body Property Checks

This section contains information on the body property rules included with Rules Checker. These rules are contained in the `body_property_checks.rle` file.

Be sure to choose the Body environment when using these rules.

## body_prop_exists

Finds the existence (or absence) of a user-specified property on a body drawing.

**Required Data**

None

**Assumptions**

None

**Default Variables**

The rule reports on all bodies that have a MAX_DELAY property.

**Default Severity**

Info

### User Customization Information

You can customize the variable macros, reported severity, and error messages for this rule:
**Variable Macros**

| Macro | Definition | Default |
|---|---|---|
| BODY_PROP_EXISTS | If set to 0, reports absence of property. If set to 1, reports existence of property. | 1 |
| BODY_PROP_EXIST_NAME | Name of property to check | “MAX_DELAY” |

### Reported Severity

| Macro | Default |
|---|---|
| BODY_PROP_EXISTS_SEVERITY | Info |

#### *Error Messages*

STD_SHORT_ERR_BODY_PROP_EXISTS STD_ERR_BODY_PROP_EXISTS

## body_prop_range_check

Reports occurrences of a user-specified property attached to the Origin whose values are not within a user-defined range.

### Required Data

None

### Assumptions

The rule assumes:

- The value of the property is of type Float
- BODY_PROP_MIN_VALUE is less than BODY_PROP_MAX_VALUE (for example, an ascending range must be specified)

### Default Variables

The rule checks to ensure all MAX_DELAY properties have values between 1 and 100000.

### Default Severity

Error

### User Customization Information

You can customize the variable macros, reported severity, and error messages for this rule:
**Variable Macros**

| Macro | Definition | Default |
|---|---|---|
| BODY_PROP_RANGE_NAME | Name of property to check | “MAX_DELAY” |
| BODY_PROP_MIN_VALUE | Minimum value allowed for the specified property | 1 |
| BODY_PROP_MAX_VALUE | Maximum value allowed for the specified property | 100000 |

**Reported Severity**

| Macro | Default |
|---|---|
| BODY_PROP_RANGE_CHECK_SEVERITY | Error |

#### *Error Messages*

STD_SHORT_ERR_BODY_PROP_RANGE_CHECK
STD_ERR_BODY_PROP_RANGE_CHECK

## body_prop_visibility

Utility that reports occurrences of the visibility of the property name and property value of a specified property (or all properties) corresponding to user-specified parameters.

### Required Data

None

### Assumptions

None

### Default Variables

As a default, the rule checks to ensure all PIN_NAME properties have visible values.

### Default Severity

Error

### User Customization Information

You can customize the variable macros, reported severity, and error messages for this rule:
**Variable Macros**

| Macro | Definition | Default |
|---|---|---|
| BODY_PROP_NAME_ VISIBILITY | Check for property name visibility (VISIBLE), invisibility (INVISIBLE), or don't care (*) | "*" |
| BODY_PROP_VALUE_ VISIBILITY | Check for property value visibility (VISIBLE), invisibility (INVISIBLE), or don't care (*) | "Invisible" |
| BODY_PROP_NAME | Name of property to check (or * to check all properties) | "PIN_NAME" |

### Reported Severity

| Macro | Default |
|---|---|
| BODY_PROP_VISIBILITY_SEVERITY | Error |

### *Error Messages*

STD_SHORT_ERR_BODY_PROP_VISIBILITY STD_ERR_BODY_PROP_VISIBILITY

## note_length_check

Reports each note in the design whose text is longer than a user-defined length.

### Required Data

None

### Assumptions

None

### Default Variables

As a default, the rule checks to make sure all notes are less than 500 characters long.

### Default Severity

Warning

### User Customization Information

You can customize the variable macros, reported severity, and error messages for this rule:
**Variable Macros**

| Macro | Definition | Default |
|---|---|---|
| MAX_NOTE_LENGTH | Maximum note length allowed | 500 |

### Reported Severity

| Macro | Default |
|---|---|
| NOTE_LENGTH_CHECK_SEVERITY | Warning |

***Error Messages***

STD_SHORT_ERR_NOTE_LENGTH_CHECK
STD_ERR_SIZE_OF_NOTE_LENGTH_CHECK

## pin_dir_check

Checks that directional property data is specified on each pin, and that the specified data is both consistent and unambiguous.

### Required Data

One or more of the following properties must be specified on each pin: BIDIRECTIONAL, OUTPUT_TYPE, OUTPUT_LOAD, INPUT_LOAD.

### Assumptions

None

### Default Variables

None

### Default Severity

Warning or Error, according to the following table:

| BIDIRECTIONAL | OUTPUT_TYPE | OUTPUT_LOAD | INPUT_LOAD | Direction | Severity |
|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | Unknown | Error |
| 0 | 0 | 1 | 1 | Bidir | Error |
| 0 | 1 | 0 | 1 | Output | Warning |
| 1 | 0 | 0 | 1 | Bidir | Warning |
| 1 | 0 | 1 | 0 | Bidir | Warning |
| 1 | 1 | 0 | 1 | Bidir | Warning |
| 1 | 1 | 1 | 0 | Bidir | Warning |

### User Customization Information

You can customize the reported variable macros, severity, and error messages for this rule:
**Reported Severity**

| Macro | Default |
|---|---|
| PIN_DIR_CHECK_NO_INFO_SEVERITY | Error |
| PIN_DIR_CHECK_INCONSIS_INFO_SEVERITY | Warning |
| PIN_DIR_CHECK_AMBIG_INFO_SEVERITY | Error |

#### *Error Messages*

STD_SHORT_ERR_PIN_DIR_CHECK_NO_INFO
STD_ERR_PIN_DIR_CHECK_NO_INFO
STD_SHORT_ERR_PIN_DIR_CHECK_AMBIG_INFO
STD_ERR_PIN_DIR_CHECK_AMBIG_INFO
STD_SHORT_ERR_PIN_DIR_CHECK_INCONSIS_INFO
STD_ERR_PIN_DIR_CHECK_INCONSIS_INFO

| Macro | Definition | Default |
|---|---|---|
| PIN_TYPE_TO_IGNORE | List of PIN_TYPE properties to be used to ignore a pin for direction checks | “POWER”, “NC”, “ANALOG” |

## prop_name_length_check

Reports properties whose name is longer than a user-defined number of characters.

### Required Data

None

### Assumptions

None

### Default Variables

As a default, the rule checks to ensure all body property names are less than 16 characters long.

### Default Severity

Error

### User Customization Information

You can customize the variable macros, reported severity, and error messages for this rule:
**Variable Macros**

| Macro | Definition | Default |
|---|---|---|
| BODY_MAX_NAME_LENGTH | Maximum length allowed for property names in the design | 16 |

**Reported Severity**

| Macro | Default |
|---|---|
| BODY_PROP_NAME_LENGTH_CHECK_SEVERITY | Error |

#### *Error Messages*

STD_SHORT_ERR_BODY_PROP_NAME_LENGTH_CHECK
STD_ERR_BODY_PROP_NAME_LENGTH_CHECK

## prop_value_length_check

Highlights each property whose value is longer than a user-defined number of characters.

### Required Data

None

### Assumptions

None

### Default Variables

As a default, the rule checks to ensure all property values are less than 256 characters long.

### Default Severity

Error

### User Customization Information

You can customize the variable macros, reported severity, and error messages for this rule:
**Variable Macros**

| Macro | Definition | Default |
|---|---|---|
| BODY_MAX_VALUE_LENGTH | Maximum length allowed for property values in the design | 256 |

**Reported Severity**

| Macro | Default |
|---|---|
| BODY_PROP_VALUE_LENGTH_CHECK_SEVERITY | Error |

#### *Error Messages*

STD_SHORT_ERR_BODY_PROP_VALUE_LENGTH_CHECK
STD_ERR_BODY_PROP_VALUE_LENGTH_CHECK

## unknown_body_prop

Highlights properties whose names are not specified on a user-defined list of allowed properties.

### Required Data

None

### Assumptions

None

### Default Variables

As a default, the rule uses the values defined in the CADENCE_DEFINED_BODY_PROP_LIST macro and the user-defined list in the `body_property_checks.h` file.

### Default Severity

Info

### User Customization Information

You can customize the variable macros, reported severity, and error messages for this rule:
**Variable Macros**

| Macro | Definition | Default |
|---|---|---|
| USER_DEFINED_BODY_PROP_LIST | User-defined list of properties allowed on bodies. Edit this macro instead of the default Cadence macro specified above. | (None) |

### Reported Severity

| Macro | Default |
|---|---|
| UNKNOWN_BODY_PROP_SEVERITY | Info |

#### *Error Messages*

STD_SHORT_ERR_UNKNOWN_BODY_PROP
STD_ERR_BODY_UNKNOWN_BODY_PROP

# Graphic Connectivity Checks

This section contains information on the graphic connectivity rules included with Rules Checker. These rules are contained in the `graphic_connectivity_checks.rle` file.

Be sure to choose the Graphical environment when using these rules.

## bit_number_mismatch

Checks that the number of bits specified on each tap body matches that of its connecting segment, and is in the range of the connecting bussed signal.

### Required Data

The rule only checks taps that have a BN property attached.

### Assumptions

The rule only checks for single-bit taps; it does not check for occurrences of scalar signal tapping.

### Default Variables

As a default, the rule considers TAP and CTAP to be tap bodies in the design.

### Default Severity

Error

### User Customization Information

You can customize the variable macros, reported severity, and error messages for this rule:
**Variable Macros**

| Macro | Definition | Default |
|---|---|---|
| TAP_BODY | Name of tap body. | “TAP,””CTAP” |

### Reported Severity

| Macro | Default |
|---|---|
| BIT_NUMBER_MISMATCH_SEVERITY | Error |

#### *Error Messages*

STD_SHORT_ERR_BIT_NUMBER_MISMATCH STD_ERR_BIT_NUMBER_MISMATCH
STD_SHORT_ERR_BIT_NUMBER_MISMATCH2 STD_ERR_BIT_NUMBER_MISMATCH2
STD_SHORT_ERR_BIT_NUMBER_MISMATCH3 STD_ERR_BIT_NUMBER_MISMATCH3

## four_way_junction

Finds all 4-way junctions (locations where two wires with all four segments logically connected cross) in the design.

### Required Data

None

### Assumptions

None

### Default Variables

None

### Default Severity

Warning

### User Customization Information

You can customize the reported severity and error messages for this rule:
**Reported Severity**

| Macro | Default |
|---|---|
| FOUR_WAY_JUNCTION_SEVERITY | Warning |

#### *Error Messages*

STD_SHORT_ERR_FOUR_WAY_JUNCTION STD_ERR_FOUR_WAY_JUNCTION

## global_and_interface

Reports wires associated with a common signal that has been declared as global and as an interface signal.

### Required Data

None

### Assumptions

None

### Default Variables

As a default, the rule considers signals labelled “\G” as global signals, and signals labelled “\I” as interface signals.

### Default Severity

Warning

### User Customization Information

You can customize the variable macros, reported severity, and error messages for this rule:
**Variable Macros**

| Macro | Definition | Default |
|---|---|---|
| GLOBAL_SIGNAL_MACRO | Extension that specifies a global signal in drawing | “\G” |
| INTERFACE_SIGNAL_MACRO | Extension that specifies an interface signal in drawing | “\I” |

**Reported Severity**

| Macro | Default |
|---|---|
| SIGNAL_SCOPE_CHECK_SEVERITY | Warning |

***Error Messages***

STD_SHORT_ERR_GLOBAL_INTERFACE_CHECK
STD_ERR_GLOBAL_INTERFACE_CHECK

## graphic_unconnected_pin

Highlights pins (except passthru pins) that are not connected to a signal.

### Required Data

None

### Assumptions

None

### Default Variables

None

### Default Severity

Warning

### User Customization Information

You can customize the reported severity and error messages for this rule:
**Reported Severity**

| Macro | Default |
|---|---|
| GRAPHIC_UNCONNECTED_PIN_SEVERITY | Warning |

#### *Error Messages*

STD_SHORT_ERR_GRAPHIC_UNCONNECTED_PIN
STD_ERR_GRAPHIC_UNCONNECTED_PIN

## illegal_signal_name

Reports wires whose associated signals have names that begin with a character that is user-specified as illegal.

### Required Data

None

### Assumptions

None

### Default Variables

As a default, the rule considers signal names that begin with “(” to be illegal.

### Default Severity

Error

### User Customization Information

You can customize the variable macros, reported severity, and error messages for this rule:
**Variable Macros**

| Macro | Definition | Default |
|---|---|---|
| SIGNAME_UNALLOWED_FIRST_CHAR | Characters specified illegal as the first character in a signal name. | (*) |

**Reported Severity**

| Macro | Default |
|---|---|
| ILLEGAL_SIGNAL_NAME_CHECK_SEVERITY | Error |

***Error Messages***

STD_SHORT_ERR_ILLEGAL_SIGNAL_NAME_CHECK
STD_ERR_ILLEGAL_SIGNAL_NAME_CHECK

## inst_signal_width_mismatch

Reports instances with a mismatch in the port and signal width

### Required Data

None

### Assumptions

None

### User Customization Information

You can customize the rule parameters, message parameters (severity, short message and long message) for this rule through UI or using a text editor. Refer to the Chapter 3, “Customizing Allegro Design Entry HDL Rules Checker,” for instructions on how to customize these at the user level.

### Rule Parameter(s)

| Parameter Name | Default |
|---|---|

### Message Parameters

### Severity

| | |
|---|---|
| INST_SIGNAL_WIDTH_MISMATCH_SEVERITY1 | Warning |
| INST_SIGNAL_WIDTH_MISMATCH_SEVERITY2 | Error |

### Short Message

| | |
|---|---|
| STD_SHORT_INST_SIGNAL_WIDTH_MISMATCH1 | "Width of signal is less than width of port" |
| STD_SHORT_INST_SIGNAL_WIDTH_MISMATCH2 | "Width of signal is more than width of port" |

### Long Message

| | |
|---|---|
| STD_ERR_INST_SIGNAL_WIDTH_MISMATCH1 | "\tPin name : ?pin_err",<br>"\tInstance : ?inst1",<br>"\tSize of instance : ?val1",<br>"\tSeg name : ?seg_err” |
| STD_ERR_INST_SIGNAL_WIDTH_MISMATCH2 | "\n\tWidth of signal is more than width of port",<br>"\tInstance : ?inst1",<br>"\tSize of instance : ?val1",<br>"\tSeg name : ?seg_err” |

## local_and_global

Reports wires associated with a common signal that has been declared as local and global.

### Required Data

None

### Assumptions

None

### Default Variables

As a default, the rule considers all signals labeled “\G” to be global signals, and all signals labeled “\I” to be interface signals.

### Default Severity

Warning

### User Customization Information

You can customize the variable macros, reported severity, and error messages for this rule:
**Variable Macros**

| Macro | Definition | Default |
|---|---|---|
| GLOBAL_SIGNAL_MACRO | Extension that specifies a global signal in drawing | “\G” |
| INTERFACE_SIGNAL_MACRO | Extension that specifies an interface signal in drawing | “\I” |

**Reported Severity**

| Macro | Default |
|---|---|
| SIGNAL_SCOPE_CHECK_SEVERITY | Warning |

#### *Error Messages*

STD_SHORT_ERR_LOCAL_GLOBAL_CHECK STD_ERR_LOCAL_GLOBAL_CHECK

## local_and_interface

Reports wires associated with a common signal that has been declared as local and as an interface signal.

### Required Data

None

### Assumptions

None

### Default Variables

As a default, the rule considers all signals labeled “\G” to be global signals, and signals labeled “\I” to be interface signals.

### Default Severity

Warning

### User Customization Information

You can customize the variable macros, reported severity, and error messages for this rule:
**Variable Macros**

| Macro | Definition | Default |
|---|---|---|
| GLOBAL_SIGNAL_MACRO | Extension that specifies a global signal in drawing | “\G” |
| INTERFACE_SIGNAL_MACRO | Extension that specifies an interface signal in drawing | “\I” |

**Reported Severity**

| Macro | Default |
|---|---|
| SIGNAL_SCOPE_CHECK_SEVERITY | Warning |

### *Error Messages*

STD_SHORT_ERR_LOCAL_INTERFACE_CHECK
STD_ERR_LOCAL_INTERFACE_CHECK

## mismatched_parenthesis

Reports properties whose values contain mismatched pairs of parentheses ( ) and angle brackets < >.

### Required Data

None

### Assumptions

None

### Default Variables

List of properties to be ignored by the rule.

### Default Severity

Error

### User Customization Information

You can customize the variable macros, reported severity, and error messages for this rule:

### Reported Severity

| Macro | Default |
|---|---|
| MISMATCH_PARENTHESIS_CHECK_SEVERITY | Error |

| Macro | Definition | Default |
|---|---|---|
| PROPS_TO_IGNORE | List of properties to be ignored by the rule | “$XRO” |

#### *Error Messages*

STD_SHORT_ERR_MISMATCH_PARENTHESIS_CHECK
STD_ERR_MISMATCH_PARENTHESIS_CHECK

## multiple_signames

Reports wires that have the following attached to them:

- Multiple occurrences of the same signal name
- Conflicting signal names

### Required Data

None

### Assumptions

None

### Default Variables

None

### Default Severity

Multiple occurrences of signal name: Info
Conflicting signal names: Error

### User Customization Information

You can customize the reported severity and error messages for this rule:

### Reported Severity

| Macro | Default |
|---|---|
| MULTIPLE_SIGNAMES_PROPS_SEVERITY1 | Info |
| MULTIPLE_SIGNAMES_PROPS_SEVERITY2 | Error |

#### *Error Messages*

STD_SHORT_MULTIPLE_SIGNAMES_PROPS1
STD_ERR_MULTIPLE_SIGNAMES_PROPS1
STD_SHORT_MULTIPLE_SIGNAMES_PROPS2
STD_ERR_MULTIPLE_SIGNAMES_PROPS2

## synonym_width_mismatch

Checks each SYNONYM body in the design to ensure that the width of the signals on either side of the body are identical.

### Required Data

None

### Assumptions

None

### Default Variables

As a default, the rule considers bodies labeled “SYNONYM” to be synonym bodies.

### Default Severity

Error

### User Customization Information

You can customize the variable macros, reported severity, and error messages for this rule:
**Variable Macros**

| Macro | Definition | Default |
|---|---|---|
| SYNONYM_BODY | Specifies the name of the synonym body | “SYNONYM” |

### Reported Severity

| Macro | Default |
|---|---|
| SYNONYM_WIDTH_MISMATCH_SEVERITY | Error |

#### *Error Messages*

STD_SHORT_ERR_SYNONYM_WIDTH_MISMATCH
STD_ERR_SYNONYM_WIDTH_MISMATCH

## vector_and_scalar

Reports wires that have been declared as both scalar and vector.

### Required Data

None

### Assumptions

None

### Default Variables

None

### Default Severity

Error

### User Customization Information

You can customize the reported severity and error messages for this rule:
**Reported Severity**

| Macro | Default |
|---|---|
| VECTOR_SCALAR_CHECK_SEVERITY | Error |

#### *Error Messages*

STD_SHORT_ERR_VECTOR_SCALAR_CHECK STD_ERR_VECTOR_SCALAR_CHECK

## flag_body_global_net

Reports flag bodies connected to global signals.

### Required Data

None

### Assumptions

None

### User Customization Information

You can customize the rule parameters, message parameters (severity, short message and long message) for this rule through UI or using a text editor. See Chapter 3, “Customizing

Allegro Design Entry HDL Rules Checker," for instructions on how to customize these at the user level.

**Rule Parameter(s)**

| Parameter Name | Default |
|---|---|
| GLOBAL_SIGNAL_MACRO | "\G" |
| FLAG_BODY_PROP_NAME | BODY_TYPE" |
| FLAG_BODY_PROP_VAL | "FLAG" |

These parameters are defined as follows:

| | |
|---|---|
| GLOBAL_SIGNAL_MACRO | Extension that specifies a global signal in drawing |
| FLAG_BODY_PROP_NAME | Name of the property to identify a flag body |
| FLAG_BODY_PROP_VALUE | Value of the property to identify a flag body |

**Message Parameters**

| Severity | Default |
|---|---|
| FLAG_BODY_GLOBAL_NET_SEVERITY | Error |

Short Message

STD_SHORT_FLAG_BODY_GLOBAL_NET

Long Message

STD_ERR_FLAG_BODY_GLOBAL_NET

Defaults can be found in header file '`graphic_connectivity_checks.h`'.

## offpage_body_global_net

Reports offpage bodies connected to global signals.

**Required Data**

None

### Assumptions

None

### User Customization Information

You can customize the rule parameters, message parameters (severity, short message and long message) for this rule through UI or using a text editor. Refer to Chapter 3, "Customizing Allegro Design Entry HDL Rules Checker," for instructions on how to customize these at the user level.

### Rule Parameter(s)

| Parameter Name | Default |
|---|---|
| GLOBAL_SIGNAL_MACRO | "\G" |
| OFFPAGE_BODY_PROP_NAME | "OFFPAGE" |
| OFFPAGE_BODY_PROP_VAL | "TRUE" |

These parameters are defined as follows:

| | |
|---|---|
| GLOBAL_SIGNAL_MACRO | Extension that specifies a global signal in drawing. |
| OFFPAGE_BODY_PROP_NAME | Name of the property to identify an offpage body. |
| OFFPAGE_BODY_PROP_VAL | Value of the property to identify an offpage body. |

### Message Parameters

| Severity | Default |
|---|---|
| OFFPAGE_BODY_GLOBAL_NET_SEVERITY | Error |

Short Message

STD_SHORT_OFFPAGE_BODY_GLOBAL_NET

Long Message

STD_ERR_OFFPAGE_BODY_GLOBAL_NET

Defaults can be found in header file '`graphic_connectivity_checks.h`'.

## unnamed_net_flag_body

Reports flag bodies connected to unnamed signals.

### Required Data

None

### Assumptions

None

### User Customization Information

You can customize the rule parameters, message parameters (severity, short message and long message) for this rule through the UI or using a text editor. Refer to Chapter 3, "Customizing Allegro Design Entry HDL Rules Checker," for instructions on how to customize these at the user level.

**Rule Parameter(s)**

| Parameter Name | Default |
|---|---|
| FLAG_BODY_PROP_NAME | "BODY_TYPE" |
| FLAG_BODY_PROP_VAL | "FLAG" |

These parameters are defined as follows:

| | |
|---|---|
| FLAG_BODY_PROP_NAME | Name of the property to identify a flag body |
| FLAG_BODY_PROP_VALUE | Value of the property to identify a flag body |

### Message Parameters

| Severity | Default |
|---|---|
| UNNAMED_NET_FLAG_BODY_SEVERITY | Error |

Short Message

STD_SHORT_UNNAMED_NET_FLAG_BODY

Long Message

STD_ERR_UNNAMED_NET_FLAG_BODY

Defaults can be found in header file '`graphic_connectivity_checks.h`'.

## unnamed_net_offpage_body

Reports offpage bodies connected to unnamed signals.

### Required Data

None

### Assumptions

None

### User Customization Information

You can customize the rule parameters, message parameters (severity, short message and long message) for this rule through the user interface or using a text editor. Refer to Chapter 3, "Customizing Allegro Design Entry HDL Rules Checker," for instructions on how to customize these at the user level.

**Rule Parameter(s)**

| Parameter Name | Default |
|---|---|
| OFFPAGE_BODY_PROP_NAME | "OFFPAGE" |
| OFFPAGE_BODY_PROP_VAL | "TRUE" |

These parameters are defined as follows:

| | |
|---|---|
| OFFPAGE_BODY_PROP_NAME | Name of the property to identify an offpage body |
| OFFPAGE_BODY_PROP_VAL | Value of the property to identify an offpage body |

### Message Parameters

| Severity | Default |
|---|---|
| UNNAMED_NET_OFFPAGE_BODY_SEVERITY | Error |

Short Message

STD_SHORT_UNNAMED_NET_OFFPAGE_BODY

Long Message

STD_ERR_UNNAMED_NET_OFFPAGE_BODY

Defaults can be found in header file '`graphic_connectivity_checks.h`'.

## local_signal_offpage_body

Reports local signals connected to offpage bodies.

### Required Data

None

### Assumptions

None

### User Customization Information

You can customize the rule parameters, message parameters (severity, short message and long message) for this rule through the user interface or using a text editor. Refer to Chapter 3, "Customizing Allegro Design Entry HDL Rules Checker," for instructions on how to customize these at the user level.

### Rule Parameter(s)

| Parameter Name | Default |
|---|---|
| OFFPAGE_BODY_PROP_NAME | "OFFPAGE" |
| OFFPAGE_BODY_PROP_VAL | "TRUE" |

These parameters are defined as follows:

| | |
|---|---|
| OFFPAGE_BODY_PROP_NAME | Name of the property to identify an offpage body |
| OFFPAGE_BODY_PROP_VAL | Value of the property to identify an offpage body |

**Message Parameters**

| Severity | Default |
|---|---|
| LOCAL_SIG_OFFPAGE_BODY_SEVERITY | Error |

Short Message

STD_SHORT_LOCAL_SIG_OFFPAGE_BODY

Long Message

STD_ERR_LOCAL_SIG_OFFPAGE_BODY

Defaults can be found in header file '`graphic_connectivity_checks.h`'.

## offpage_signal_no_offpage_body

Reports offpage signals that do not have offpage signals connected to them.

**Required Data**

None

**Assumptions**

None

**User Customization Information**

You can customize the rule parameters, message parameters (severity, short message and long message) for this rule through the user interface or using a text editor. Refer to

Chapter 3, “Customizing Allegro Design Entry HDL Rules Checker,” for instructions on how to customize these at the user level.

**Rule Parameter(s)**

| Parameter Name | Default |
|---|---|
| OFFPAGE_BODY_PROP_NAME | “OFFPAGE” |
| OFFPAGE_BODY_PROP_VAL | “TRUE” |

These parameters are defined as follows:

| | |
|---|---|
| OFFPAGE_BODY_PROP_NAME | Name of the property to identify an offpage body |
| OFFPAGE_BODY_PROP_VAL | Value of the property to identify an offpage body |

**Message Parameters**

| Severity | Default |
|---|---|
| OFFPAGE_SIGNAL_NO_OFFPAGE_BODY_SEVERITY | Error |

Short Message

STD_SHORT_OFFPAGE_SIG_NO_OFFPAGE_BODY

Long Message

STD_ERR_OFFPAGE_SIG_NO_OFFPAGE_BODY

Defaults can be found in header file ‘`graphic_connectivity_checks.h`’.

# Graphic Drawing Checks

This section contains information on the graphical drawing rules included with Rules Checker. These rules are contained in the `graphic_drawing_checks.rle` file.

Be sure to choose the Graphical environment when using these rules.

## color_check

Checks that only user-specified colors are used on notes, signals, properties, and instances in the design.

### Required Data

None

### Assumptions

None

### Default Variables

As a default, the rule checks to make sure the colors of all notes, segments, properties, and instances are one of the following colors:

Mono, Red, Green, Blue, Yellow, Orange, Salmon, Violet, Brown, SkyBlue, White, Peach, Pink, Purple, Aqua, Gray

### Default Severity

Warning

### User Customization Information

You can customize the variable macros, reported severity, and error messages for this rule:

### Variable Macros

| Macro | Definition | Default |
|---|---|---|
| USER_DEFINED_COLORS_FOR_GRAPHICAL_NOTE | Colors allowed on notes | "Mono," "Red," "Green," "Blue," "Yellow," "Orange," "Salmon," "Violet," "Brown," "SkyBlue," "White," "Peach," "Pink," "Purple," "Aqua," "Gray" |
| USER_DEFINED_COLORS_FOR_GRAPHICAL_SEGS | Colors allowed on segments | |
| USER_DEFINED_COLORS_FOR_GRAPHICAL_PROP | Colors allowed on properties | |
| USER_DEFINED_COLORS_FOR_GRAPHICAL_INST | Colors allowed on instances | |

### Reported Severity

| Macro | Default |
|---|---|
| GRAPHICAL_COLOR_CHECK_SEVERITY | Warning |

#### *Error Messages*

STD_SHORT_ERR_GRAPHICAL_COLOR_CHECK
STD_ERR_GRAPHICAL_COLOR_CHECK

## inst_note_overlap

Checks for instances that overlap notes in the design.

### Required Data

The rule assumes that the CADENCE_PAGE_BORDERS macro has been defined.

### Assumptions

None

### Default Variables

As a default, the rule does not check the following instances, which are considered to be page borders: A SIZE PAGE, B SIZE PAGE, C SIZE PAGE, D SIZE PAGE, DRAWING.

### Default Severity

Error

### User Customization Information

You can customize the variable macros, reported severity, and error messages for this rule:
**Variable Macros**

| Macro | Definition | Default |
|---|---|---|
| CADENCE_PAGE_BORDERS | Names of page border bodies | “A SIZE PAGE,”<br>“B SIZE PAGE,”<br>“C SIZE PAGE,”<br>“D SIZE PAGE,”<br>“DRAWING” |

**Reported Severity**

| Macro | Default |
|---|---|
| INST_NOTE_OVERLAP_SEVERITY | Error |

#### *Error Messages*

STD_SHORT_ERR_INST_NOTE_OVERLAP
STD_ERR_INST_NOTE_OVERLAP

## inst_overlap

Reports instances (except page borders) whose bounding boxes overlap.

### Required Data

None

### Assumptions

None

### Default Variables

As a default, the rule does not check the following instances, which are considered to be page borders: A SIZE PAGE, B SIZE PAGE, C SIZE PAGE, D SIZE PAGE, DRAWING

### Default Severity

Error

### User Customization Information

You can customize the variable macros, reported severity, and error messages for this rule:

### Variable Macros

| Macro | Definition | Default |
|---|---|---|
| CADENCE_PAGE_BORDERS | Names of page border bodies | “A SIZE PAGE,” “B SIZE PAGE,” “C SIZE PAGE,” “D SIZE PAGE,” “DRAWING” |

### Reported Severity

| Macro | Default |
|---|---|
| INST_OVERLAP_SEVERITY | Error |

### *Error Messages*

STD_SHORT_ERR_INST_OVERLAP STD_ERR_INST_OVERLAP

## inst_prop_offset

Reports instances and attached visible properties that are separated by more than a maximum specified distance.

### Required Data

None

### Assumptions

None

### Default Variables

As a default, the rule checks for instances and attached properties that are separated by more than 1000 units (1000 units = 2 inches).

### Default Severity

Error

### User Customization Information

You can customize the variable macros, reported severity, and error messages for this rule:

#### Variable Macros

| Macro | Definition | Default |
|---|---|---|
| MAX_INST_PROP_OFFSET | Maximum distance allowed between instance and an attached property. | 1000 |

#### Reported Severity

| Macro | Default |
|---|---|
| INST_PROP_OFFSET_SEVERITY | Error |

#### *Error Messages*

STD_SHORT_ERR_INST_PROP_OFFSET STD_ERR_INST_PROP_OFFSET

## inst_prop_overlap

Checks whether each instance overlaps with its visible property (name and/or value).

### Required Data

None

### Assumptions

None

### Default Variables

None

### Default Severity

Error

### User Customization Information

You can customize the variable macros and error messages for this rule:

### Reported Severity

| Macro | Default |
|---|---|
| INST_PROP_OVERLAP_SEVERITY | Error |

#### *Error Messages*

STD_SHORT_ERR_INST_PROP_OVERLAP STD_ERR_INST_PROP_OVERLAP

## inst_seg_overlap

Checks for instances that overlap segments in the design.

### Required Data

None

### Assumptions

None

### Default Variables

As a default, the rule does not check the following instances, which are considered to be page borders: A SIZE PAGE, B SIZE PAGE, C SIZE PAGE,
D SIZE PAGE, DRAWING

### Default Severity

Error

### User Customization Information

You can customize the variable macros, reported severity, and error messages for this rule:

#### Variable Macros

| Macro | Definition | Default |
|---|---|---|
| CADENCE_PAGE_BORDERS | Names of page border bodies | “A SIZE PAGE,” “B SIZE PAGE,” “C SIZE PAGE,” “D SIZE PAGE,” “DRAWING” |

#### Reported Severity

| Macro | Default |
|---|---|
| INST_SEG_OVERLAP_SEVERITY | Error |

#### *Error Messages*

STD_SHORT_ERR_INST_SEG_OVERLAP STD_ERR_INST_SEG_OVERLAP

## min_wire_spacing

Reports parallel segments in the drawing that are less than a specified distance from each other.

### Required Data

None

### Assumptions

None

### *Default Variables*

As a default, the rule checks for parallel segments that are less than 25 units (.05 inches) away from each other.

### Default Severity

Error

### User Customization Information

You can customize the variable macros, reported severity, and error messages for this rule:

### Variable Macros

| Macro | Definition | Default |
|---|---|---|
| SPECIFIED_DISTANCE_BETWEEN_PARALLEL_SEGMENTS | Minimum distance allowed between parallel segments in the drawing. | 25 |

### Reported Severity

| Macro | Default |
|---|---|
| MIN_WIRE_SPACING_SEVERITY | Error |

### *Error Messages*

STD_SHORT_ERR_MIN_WIRE_SPACING1 STD_ERR_MIN_WIRE_SPACING1
STD_SHORT_ERR_MIN_WIRE_SPACING2 STD_ERR_MIN_WIRE_SPACING2
STD_SHORT_MIN_WIRE_SPACING3 STD_ERR_MIN_WIRE_SPACING3

## non_orthogonal_wires

Reports segments in the drawing that are not aligned horizontal or vertical.

### Required Data

None

### Assumptions

None

### Default Variables

None

### Default Severity

Warning

### User Customization Information

You can customize the reported severity and error messages for this rule:

### Reported Severity

| Macro | Default |
|---|---|
| NON_ORTHOGONAL_WIRES_SEVERITY | Warning |

### *Error Messages*

STD_SHORT_ERR_NON_ORTHOGONAL_WIRES
STD_ERR_NON_ORTHOGONAL_WIRES

## note_overlap

Reports notes whose bounding boxes overlap in the drawing.

### Required Data

None

### Assumptions

None

### Default Variables

None

### Default Severity

Error

### User Customization Information

You can customize the reported severity and error messages for this rule:

### Reported Severity

| Macro | Default |
|---|---|
| NOTE_OVERLAP_SEVERITY | Error |

### *Error Messages*

STD_SHORT_ERR_NOTE_OVERLAP STD_ERR_NOTE_OVERLAP

## note_prop_overlap

Checks for properties and notes that overlap in the design.

### Required Data

None

### Assumptions

None

### Default Variables

None

### Default Severity

Error

### User Customization Information

You can customize the reported severity and error messages for this rule:

### Reported Severity

| Macro | Default |
|---|---|
| NOTE_PROP_OVERLAP_SEVERITY | Error |

### *Error Messages*

STD_SHORT_ERR_NOTE_PROP_OVERLAP STD_ERR_NOTE_PROP_OVERLAP

## prop_overlap

Reports visible properties whose bounding boxes overlap in the drawing.

### Required Data

None

### Assumptions

The rule considers a property with a visible name and/or a visible value to be visible; both name and value do not have to be visible.

### Default Variables

None

### Default Severity

Error

### User Customization Information

You can customize the reported severity and error messages for this rule:

**Reported Severity**

| Macro | Default |
|---|---|
| PROP_OVERLAP_SEVERITY | Error |

#### *Error Messages*

STD_SHORT_ERR_PROP_OVERLAP STD_ERR_PROP_OVERLAP

## seg_note_overlap

Checks for overlapping segments and notes in the design.

### Required Data

None

### Assumptions

None

### Default Variables

None

### Default Severity

Error

### User Customization Information

You can customize the reported severity and error messages for this rule:

### Reported Severity

| Macro | Default |
|---|---|
| SEG_NOTE_OVERLAP_SEVERITY | Error |

#### *Error Messages*

STD_SHORT_ERR_SEG_NOTE_OVERLAP STD_ERR_SEG_NOTE_OVERLAP

## seg_prop_overlap

Checks for overlapping segments and properties on wires in the design.

### Required Data

None

### Assumptions

None

### Default Variables

None

### Default Severity

Error

### User Customization Information

You can customize the reported severity and error messages for this rule:

### Reported Severity

| Macro | Default |
|---|---|
| SEG_PROP_OVERLAP_SEVERITY | Error |

#### *Error Messages*

STD_SHORT_ERR_SEG_PROP_OVERLAP STD_ERR_SEG_PROP_OVERLAP

## seg_wire_prop_offset

Reports wires and attached visible properties that are separated by more than a maximum specified distance.

### Required Data

None

### Assumptions

None

### Default Variables

As a default, the rule checks for wires and attached properties that are separated by more than 100 units (100 units = 0.2 inch).

### Default Severity

Error

### User Customization Information

You can customize the variable macros, reported severity, and error messages for this rule:

### Variable Macros

| Macro | Definition | Default |
|---|---|---|
| WIRE_PROP_OFFSET | Maximum distance allowed between a wire and an attached property. | 100 |

### Reported Severity

| Macro | Default |
|---|---|
| WIRE_PROP_OFFSET_SEVERITY | Error |

#### *Error Messages*

STD_SHORT_ERR_WIRE_PROP_OFFSET STD_ERR_WIRE_PROP_OFFSET

# Graphic Property Checks

This section contains information on the graphic property rules included with Rules Checker. These rules are contained in the `graphic_property_checks.rle` file.

Be sure to choose the Graphical environment when using these rules.

## biput_pin_prop_exists

Reports biput pins where a user-specified property either exists, or is missing (determined by user).

### Required Data

None

### Assumptions

None

### Default Variables

As a default, the rule reports any existence of an OUTPUT_CAP property on biput pins.

### Default Severity

Info

### User Customization Information

You can customize the variable macros, reported severity, and error messages for this rule:

#### Variable Macros

| Macro | Definition | Default |
|---|---|---|
| GRAPHICAL_BIPUT_PIN_PROP_EXIST_NAME | If set to 0, checks for absence of property. If set to 1, checks for existence of property. | “OUTPUT_CAP” |
| GRAPHICAL_BIPUT_PIN_PROP_EXISTS | Name of property to check | 1 |

#### Reported Severity

| Macro | Default |
|---|---|
| GRAPHICAL_BIPUT_PIN_PROP_EXISTS_SEVERITY | Info |

#### *Error Messages*

STD_SHORT_ERR_GRAPHICAL_BIPUT_PIN_PROP_EXISTS
STD_ERR_GRAPHICAL_BIPUT_PIN_PROP_EXISTS

## input_pin_prop_exists

Reports input pins where a user-specified property either exists, or is missing (determined by user).

### Required Data

None

### Assumptions

None

### Default Variables

As a default, the rule reports any occurrence of the INPUT_CAP property on input pins in the design.

### Default Severity

Info

### User Customization Information

You can customize the variable macros, reported severity, and error messages for this rule:

### Variable Macros

| Macro | Definition | Default |
|---|---|---|
| GRAPHICAL_INPUT_PIN_PROP_EXIST_NAME | Comma-separated list of property(s) to check. | “INPUT_CAP” |
| GRAPHICAL_INPUT_PIN_PROP_EXISTS | If set to 0, checks for absence of property. If set to 1, checks for existence of property. | 1 |

### Reported Severity

| Macro | Default |
|---|---|
| GRAPHICAL_INPUT_PIN_PROP_EXISTS_SEVERITY | Info |

#### *Error Messages*

STD_SHORT_ERR_GRAPHICAL_INPUT_PIN_PROP_EXISTS
STD_ERR_GRAPHICAL_INPUT_PIN_PROP_EXISTS

## inst_prop_exists

Reports instances where a user-specified property either exists, or is missing (determined by user).

### Required Data

None

### Assumptions

None

### Default Variables

As a default, the rule checks for the existence of MAX_DELAY property on each instance in the design.

### Default Severity

Info

### User Customization Information

You can customize the variable macros, reported severity, and error messages for this rule:

### Variable Macros

| Macro | Definition | Default |
|---|---|---|
| GRAPHICAL_INSTANCE_PROP_EXIST_NAME | Comma-separated list of property(s) to check. | “MAX_DELAY” |
| GRAPHICAL_INSTANCE_PROP_EXISTS | If set to 0, checks for absence of property. If set to 1, checks for existence of property. | 1 |

### Reported Severity

| Macro | Default |
|---|---|
| GRAPHICAL_INST_PROP_EXISTS_SEVERITY | Info |

#### *Error Messages*

STD_SHORT_ERR_GRAPHICAL_INST_PROP_EXISTS
STD_ERR_GRAPHICAL_INST_PROP_EXISTS

## inst_prop_range_check

Reports instances in which the value of a specified property(s) is outside of a user-defined range.

### Required Data

None

### Assumptions

This rule assumes the value for property is of type float.

### Default Variables

As a default, the rule checks to ensure that MAX_DELAY property on an instance has a value between 1 and 100000.

### Default Severity

Error

### User Customization Information

You can customize the variable macros, reported severity, and error messages for this rule:

### Variable Macros

| Macro | Definition | Default |
|---|---|---|
| GRAPHICAL_INST_PROP_RANGE_NAME | Comma-separated name(s) of instance property(s) | “MAX_DELAY” |
| GRAPHICAL_INST_PROP_MIN_VALUE | Comma-separated minimum value(s) allowed for instance | “1” |
| GRAPHICAL_INST_PROP_MAX_VALUE | Comma-separated maximum value(s) allowed for instance | “100000” |

### Reported Severity

| Macro | Default |
|---|---|
| GRAPHICAL_INST_PROP_RANGE_CHECK_SEVERITY | Error |

### *Error Messages*

STD_SHORT_ERR_GRAPHICAL_INST_PROP_RANGE_CHECK
STD_ERR_GRAPHICAL_INST_PROP_RANGE_CHECK

## inst_prop_visibility

Highlights user-specified instance property name-value pairs that match the specified visibility parameters.

### Required Data

None

### Assumptions

None

### Default Variables

As a default, the rule reports any LOCATION properties on instances that have an invisible value.

### Default Severity

Warning

### User Customization Information

You can customize the variable macros, reported severity, and error messages for this rule:

### Variable Macros

| Macro | Definition | Default |
|---|---|---|
| INST_PROP_NAME_ VISIBILITY | Visibility to check for on property names: INVISIBLE, VISIBLE, or “*” (don’t care) | “*” |
| INST_PROP_VALUE_ VISIBILITY | Visibility to check for on property values: INVISIBLE, VISIBLE, or “*” (don’t care) | “INVISIBLE” |
| INST_PROP_NAME | Name of properties to check. “*” specifies all properties | “LOCATION” |

### Reported Severity

| Macro | Default |
|---|---|
| INST_PROP_VISIBILITY_SEVERITY | Warning |

#### *Error Messages*

STD_SHORT_ERR_INST_PROP_VISIBILITY STD_ERR_INST_PROP_VISIBILITY

## null_prop

Checks for null (value = “”) properties on the design.

### Required Data

None

### Assumptions

None

### Default Variables

None

### Default Severity

Warning

### User Customization Information

You can customize the variable macros, reported severity, and error messages for this rule:

### Reported Severity

| Macro | Default |
|---|---|
| NULL_PROP_VAL_SEVERITY | Warning |

### *Error Messages*

STD_SHORT_ERR_NULL_PROP_VAL STD_ERR_NULL_PROP_VAL

## output_pin_prop_exists

Reports output pins where a user-specified property either exists, or is missing (determined by user).

### Required Data

None

### Assumptions

None

### Default Variables

As a default, the rule reports any existence of the OUTPUT_CAP property on output pins in the design.

### Default Severity

Info

### User Customization Information

You can customize the variable macros, reported severity, and error messages for this rule:

### Variable Macros

| Macro | Definition | Default |
|---|---|---|
| GRAPHICAL_OUTPUT_PIN_PROP_EXIST_NAME | Comma-separated list of property(s) to check. | “OUTPUT_CAP” |
| GRAPHICAL_OUTPUT_PIN_PROP_EXISTS | If set to 0, checks for absence of property. If set to 1, checks for existence of property. | 1 |

### Reported Severity

| Macro | Default |
|---|---|
| GRAPHICAL_OUTPUT_PIN_PROP_EXISTS_ SEVERITY | Info |

#### *Error Messages*

STD_SHORT_ERR_GRAPHICAL_OUTPUT_PIN_PROP_EXISTS
STD_ERR_GRAPHICAL_OUTPUT_PIN_PROP_EXISTS

## pin_prop_range_check

Reports pins in which the value of a specified property is outside of a user-defined range.

### Required Data

None

### Assumptions

This rule assumes:

- The value for property is of type float
- The value for GRAPHICAL_PIN_PROP_MIN_VALUE is less than the value for GRAPHICAL_PIN_PROP_MAX_VALUE

### Default Variables

As a default, the rule checks that the INPUT_CAP property on pins has a value between 1 and 100.

### Default Severity

Error

### User Customization Information

You can customize the variable macros, reported severity, and error messages for this rule:

### Variable Macros

| Macro | Definition | Default |
|---|---|---|
| GRAPHICAL_PIN_PROP_ RANGE_NAME | Name of pin property | “INPUT_CAP” |
| GRAPHICAL_PIN_PROP_MIN_VALUE | Minimum value allowed for pin property | “1” |
| GRAPHICAL_PIN_PROP_MAX_VALUE | Maximum value allowed for pin property | “100” |

### Reported Severity

| Macro | Default |
|---|---|
| GRAPHICAL_PIN_PROP_RANGE_CHECK_SEVERITY | Error |

#### *Error Messages*

STD_SHORT_ERR_GRAPHICAL_PIN_PROP_RANGE_CHECK
STD_ERR_GRAPHICAL_PIN_PROP_RANGE_CHECK

## pin_prop_visibility

Highlights user-specified pin property name-value pairs that match the specified visibility parameters.

### Required Data

None

### Assumptions

None

### Default Variables

As a default, the rule reports any MAX_DELAY properties on pins in the design that have invisible values.

### Default Severity

Warning

### User Customization Information

You can customize the variable macros, reported severity, and error messages for this rule:

### Variable Macros

| Macro | Definition | Default |
|---|---|---|
| PIN_PROP_NAME_ VISIBILITY | Visibility to check for on property names: INVISIBLE, VISIBLE, or * (don't care) | "*" |
| PIN_PROP_VALUE_ VISIBILITY | Visibility to check for on property values: INVISIBLE, VISIBLE, or * (don't care) | "Invisible" |
| PIN_PROP_NAME | Name of properties to check. "*" specifies all properties. | "MAX_DELAY" |

### Reported Severity

| Macro | Default |
|---|---|
| PIN_PROP_VISIBILITY_SEVERITY | Warning |

#### *Error Messages*

STD_SHORT_ERR_PIN_PROP_VISIBILITY STD_ERR_PIN_PROP_VISIBILITY

## prop_name_length_check

Reports each property whose name is longer than a user-defined number of characters.

### Required Data

None

### Assumptions

None

### Default Variables

As a default, the rule checks that all property names are less than 16 characters long.

### Default Severity

Error

### User Customization Information

You can customize the variable macros, reported severity, and error messages for this rule:

**Variable Macros**

| Macro | Definition | Default |
|---|---|---|
| GRAPHICAL_MAX_NAME_LENGTH | Maximum length allowed for property names in the design | 16 |

**Reported Severity**

| Macro | Default |
|---|---|
| GRAPHICAL_PROP_NAME_LENGTH_CHECK_SEVERITY | Error |

#### *Error Messages*

STD_SHORT_ERR_GRAPHICAL_PROP_NAME_LENGTH_CHECK
STD_ERR_GRAPHICAL_PROP_NAME_LENGTH_CHECK

## prop_value_length_check

Reports each property whose value is longer than a user-defined number of characters.

### Required Data

None

### Assumptions

None

### Default Variables

As a default, the rule checks to ensure the lengths of all property values in the design is less than 256 characters.

### Default Severity

Error

### User Customization Information

You can customize the variable macros, reported severity, and error messages for this rule:
**Variable Macros**

| Macro | Definition | Default |
|---|---|---|
| GRAPHICAL_MAX_VALUE_LENGTH | Maximum length allowed for property values in the design | 256 |

**Reported Severity**

| Macro | Default |
|---|---|
| GRAPHICAL_PROP_VALUE_LENGTH_CHECK_SEVERITY | Error |

### *Error Messages*

STD_SHORT_ERR_GRAPHICAL_PROP_VALUE_LENGTH_CHECK
STD_ERR_GRAPHICAL_PROP_VALUE_LENGTH_CHECK

## unknown_inst_prop

Reports instances with properties that are not included on a user-defined list of instance properties allowed on the design.

### Required Data

None

### Assumptions

None

### Default Variables

As a default, the rule checks for instance properties defined in the CADENCE_DEFINED_INST_PROP_LIST macro in `graphic_property_checks.h`.

### Default Severity

Info

### User Customization Information

You can customize the variable macros, reported severity, and error messages for this rule:

### Variable Macros

| Macro | Definition | Default |
|---|---|---|
| USER_DEFINED_INST_PROP_LIST | User-defined list of properties allowed on instances in the design. Modify this macro instead of the Cadence-defined list in `graphic_property_checks.h` | (None) |

### Reported Severity

| Macro | Default |
|---|---|
| UNKNOWN_INST_PROP_SEVERITY | Info |

#### *Error Messages*

STD_SHORT_ERR_UNKNOWN_INST_PROP STD_ERR_UNKNOWN_INST_PROP

## unknown_pin_prop

Reports pins with properties that are not included on a user-defined list of pin properties allowed on the design.

### Required Data

None

### Assumptions

None

### Default Variables

As a default, the rule checks for pin properties defined in the CADENCE_DEFINED_PIN_PROP_LIST macro in `graphic_property_checks.h`.

### Default Severity

Info

### User Customization Information

You can customize the variable macros, reported severity, and error messages for this rule:

### Variable Macros

| Macro | Definition | Default |
|---|---|---|
| USER_DEFINED_PIN_PROP_LIST | User-defined list of properties allowed on pins in the design. Modify this macro instead of the Cadence defined list in `graphic_property_checks.h`. | (None) |

### Reported Severity

| Macro | Default |
|---|---|
| UNKNOWN_PIN_PROP_SEVERITY | Info |

#### *Error Messages*

STD_SHORT_ERR_UNKNOWN_PIN_PROP STD_ERR_UNKNOWN_PIN_PROP

## unknown_wire_prop

Reports wires with properties that are not included on a user-defined list of wire properties allowed on the design.

### Required Data

None

### Assumptions

None

### Default Variables

As a default, the rule checks for wire properties defined in the CADENCE_DEFINED_WIRE_PROP_LIST macro in `graphic_property_checks.h`.

### Default Severity

Info

### User Customization Information

You can customize the variable macros, reported severity, and error messages for this rule:

### Variable Macros

| Macro | Definition | Default |
|---|---|---|
| USER_DEFINED_WIRE_PROP_LIST | User-defined list of properties allowed on wires in the design. Modify this macro instead of the Cadence defined list in `graphic_property_checks.h`. | (None) |

### Reported Severity

| Macro | Default |
|---|---|
| UNKNOWN_WIRE_PROP_SEVERITY | Info |

#### *Error Messages*

STD_SHORT_ERR_UNKNOWN_WIRE_PROP STD_ERR_UNKNOWN_WIRE_PROP

## wire_prop_exists

Reports wires where a user-specified property either exists, or is missing (determined by user).

### Required Data

None

### Assumptions

None

### Default Variables

As a default, the rule checks for the existence of a ROUTE_PRIORITY property on each wire in the design.

### Default Severity

Info

### User Customization Information

You can customize the variable macros, reported severity, and error messages for this rule:

#### Variable Macros

| Macro | Definition | Default |
|---|---|---|
| GRAPHICAL_WIRE_PROP_EXIST_NAME | Comma-separated list of property(s) to check. | “ROUTE_PRIORITY” |
| GRAPHICAL_WIRE_PROP_EXISTS | If set to 0, checks for absence of property. If set to 1, checks for existence of property. | 1 |

#### Reported Severity

| Macro | Default |
|---|---|
| GRAPHICAL_WIRE_PROP_EXISTS_SEVERITY | Info |

***Error Messages***

STD_SHORT_ERR_GRAPHICAL_WIRE_PROP_EXISTS
STD_ERR_GRAPHICAL_WIRE_PROP_EXISTS

## wire_prop_range_check

Reports wires in which the value of a specified property is outside of a user-defined range.

### Required Data

None

### Assumptions

This rule assumes:

- The value for property is of type float
- The value for GRAPHICAL_WIRE_PROP_MIN_VALUE is less than the value for GRAPHICAL_WIRE_PROP_MAX_VALUE

### Default Variables

As a default, the rule checks to ensure that ROUTE_PRIORITY property on a wire has a value between 0 and 99.

### Default Severity

Error

### User Customization Information

You can customize the variable macros, reported severity, and error messages for this rule:

### Variable Macros

| Macro | Definition | Default |
|---|---|---|
| GRAPHICAL_WIRE_PROP_ RANGE_NAME | Name of wire property | “ROUTE_PRIORITY” |
| GRAPHICAL_WIRE_PROP_MIN_VALU E | Minimum value allowed for wire property | “0” |
| GRAPHICAL_WIRE_PROP_MAX_VALU E | Maximum value allowed for wire property | “99” |

### Reported Severity

| Macro | Default |
|---|---|
| GRAPHICAL_WIRE_PROP_RANGE_CHECK_ SEVERITY | Error |

#### *Error Messages*

STD_SHORT_ERR_GRAPHICAL_WIRE_PROP_RANGE_CHECK
STD_ERR_GRAPHICAL_WIRE_PROP_RANGE_CHECK

## wire_prop_visibility

Reports user-specified wire property name-value pairs that match the specified visibility parameters.

### Required Data

None

### Assumptions

None

### Default Variables

As a default, the rule reports any SIG_NAME properties on wires that have an invisible value.

### Default Severity

Warning

### User Customization Information

You can customize the variable macros, reported severity, and error messages for this rule:

### Variable Macros

| Macro | Definition | Default |
|---|---|---|
| WIRE_PROP_NAME_ VISIBILITY | Visibility to check for on property names: INVISIBLE, VISIBLE, or “*” (don’t care) | “*” |
| WIRE_PROP_VALUE_ VISIBILITY | Visibility to check for on property values: INVISIBLE, VISIBLE, or “*” (don’t care) | “INVISIBLE” |
| WIRE_PROP_NAME | Name of properties to check. “*” specifies all properties. | “SIG_NAME” |

### Reported Severity

| Macro | Default |
|---|---|
| WIRE_PROP_VISIBILITY_SEVERITY | Warning |

### *Error Messages*

STD_SHORT_ERR_WIRE_PROP_VISIBILITY STD_ERR_WIRE_PROP_VISIBILITY

# Graphic Section Checks

This section contains information on the graphic section rules included with Rules Checker. These rules are contained in the `graphic_section_checks.rle` file.

Be sure to choose the Graphical environment when using these rules.

## invalid_part_name

Checks that physical components (except those without a physical packaging, with a logic drawing, or in a user-defined list) have valid PART_NAME values in the `chips_prt` file. If not, Rules Checker highlights the body of the component.

### Required Data

If you are going to specify bodies to ignore, you must specify them via the USER_DEFINED_BODIES_TO_IGNORE_FOR_BODY_PART_NAME_CHECK macro in `body_cross_view_checks.h`.

### Assumptions

The rule assumes that the body is non-PLUMBING and not a COMMENT body.

### Default Variables

As a default, the rule does not check the following components: “DRAWING,” “DECLARATIONS,” “HDL_DECS,” “VHDL_DECS,” “VERILOG_DECS”

### Default Severity

Error

### User Customization Information

You can customize the variable macros, reported severity, and error messages for this rule:

### Variable Macros

| Macro | Definition | Default |
|---|---|---|
| USER_DEFINED_BODIES_TO_ IGNORE_FOR_GRAPHICAL_PART_NAME _CHECK | List of bodies not to be checked | “DRAWING,” “DECLARATIONS,” “HDL_DECS,” “VHDL_DECS,”“VERILOG_DECS” |

### Reported Severity

| Macro | Default |
|---|---|
| GRAPHICAL_INVALID_PART_NAME_SEVERITY | Error |

#### *Error Messages*

STD_SHORT_ERR_GRAPHICAL_INVALID_PART_NAME
STD_ERR_GRAPHICAL_INVALID_PART_NAME

## invalid_pin_assignment

Finds instances whose pins have assigned invalid pin numbers, by checking the section of each pin, and by checking pin $PN and PN values against pin numbers specified in the `chips_prt` file.

### Required Data

None

### Assumptions

None

### Default Variables

None

### Default Severity

Error

### User Customization Information

You can customize the reported severity and error messages for this rule:

### Reported Severity

| Macro | Default |
|---|---|
| INVALID_PIN_ASSIGNMENT_SEVERITY | Error |

#### *Error Messages*

STD_SHORT_ERR_INVALID_PIN_ASSIGNMENT
STD_ERR_INVALID_PIN_ASSIGNMENT

## pack_sec_type_mismatch

Checks that the PACK_TYPE and SEC_TYPE values match on each instance in the design.

### Required Data

None

### Assumptions

None

### Default Variables

None

### Default Severity

Error

### User Customization Information

You can customize the reported severity and error messages for this rule:

### Reported Severity

| Macro | Default |
|---|---|
| PACK_SEC_TYPE_MISMATCH_SEVERITY | Error |

#### *Error Messages*

STD_SHORT_ERR_PACK_SEC_TYPE_MISMATCH
STD_ERR_PACK_SEC_TYPE_MISMATCH

## section_pin_mismatch

Finds instances whose pin numbers do not match the pin numbers defined by the section number (SEC property).

### Required Data

None

### Assumptions

None

### Default Variables

None

### Default Severity

Error

### User Customization Information

You can customize the reported severity and error messages for this rule:

### Reported Severity

| Macro | Default |
|---|---|
| SECTION_PIN_MISMATCH_SEVERITY | Error |

#### *Error Messages*

STD_SHORT_ERR_SECTION_PIN_MISMATCH1
STD_ERR_SECTION_PIN_MISMATCH1
STD_SHORT_ERR_SECTION_PIN_MISMATCH2
STD_ERR_SECTION_PIN_MISMATCH2

# Electrical Rules

This section contains information on the graphical drawing rules included with Rules Checker.

## cap_check

Checks every signal to ensure that its total load capacitance is not higher than what the source can drive.

### Required Data

You must define the INPUT_CAP and OUTPUT_CAP properties (described below) in the `cp_config.h` file.

If you are using parts from a non-Cadence library, you must specify the TECH property (using a value from `cp_global.h`) on the components in your schematic.

### Assumptions

None

### Default Variables

None

### Default Severity

Warning

### User Customization Information

You can customize the global macros, reported severity, and error messages for this rule:

### Global Macros

| Macro | Description |
|---|---|
| INPUT_CAP | Input capacitance technology you are using |
| OUTPUT_CAP | Output capacitance of technology you are using |

### Reported Severity

| Macro | Default |
|---|---|
| CAP_CHECK_SEVERITY | Warning |

#### *Error Messages*

STD_SHORT_ERR_CAP_CHECK
STD_ERR_CAP_CHECK

## cmos_no_pullup

Checks that input terminals on CMOS components are connected to a pull-up resistor.

### Required Data

If you are using parts from a non-Cadence library, you must

- Define the POWER_SYMBOL macro in the `cp_config.h` file.

  The default values are VCC, VDD, VEE, VEE1, VEE2, and VSS.

- Specify the TECH property (using a value from `cp_global.h`) on the appropriate components on your schematic to identify them as CMOS components.
- Specify the property name (RES_PROP_NAME) and value (RES_PROP_VALUE) used to specify resistors.

  The default value of RES_PROP_NAME is COMP_TYPE, and the default for RES_PROP_VALUE is RES.

### Assumptions

None

### Default Variables

None

### Default Severity

Error

### User Customization Information

You can customize the global macros, reported severity, and error messages for this rule.

### Global Macros

| Macro | Description |
|---|---|
| POWER_SYMBOL | Comma-separated list of power pins/nets found in the design |
| CMOS_TECH | Allowed technologies for CMOS |
| RES_PROP_NAME | Property to identify resistor |
| RES_PROP_VALUE | Value of property RES_PROP_NAME to identify a resistor |

### Reported Severity

| Macro | Default |
|---|---|
| CMOS_NO_PULL_UP_SEVERITY | Error |

***Error Messages***

STD_SHORT_ERR_CMOS_NO_PULL_UP_1
STD_ERR_CMOS_NO_PULL_UP_1
STD_SHORT_ERR_CMOS_NO_PULL_UP_2
STD_ERR_CMOS_NO_PULL_UP_2
STD_SHORT_ERR_CMOS_NO_PULL_UP_3
STD_ERR_CMOS_NO_PULL_UP_3

## conn_mos

Checks each interface signal in your design to make sure that you use a resistor when connecting any of the signal's input or output terminals (physical connectors) to MOS inputs.

### Required Data

You need to specify each input, output, or I/O terminal in your design, by attaching the "\I" suffix to each signal name.

If you are using parts from a non-Cadence library, you must specify the TECH property (using a value from `cp_global.h`) on the components in your schematic.

### Assumptions

The rule assumes that an interface signal results in a pin attached to a non-primitive object, such as a hierarchical body. Also, the rule does not check for non-resistive components between MOS inputs and interface signals.

### Default Variables

None

### Default Severity

Warning

### User Customization Information

You can customize the global macros, reported severity, and error messages for this rule.

### Global Macros

| Macro | Description |
|---|---|
| MOS_TECH | Identifies MOS technology used. |

### Reported Severity

| Macro | Default |
|---|---|
| CONN_MOS_SEVERITY | Warning |

#### *Error Messages*

STD_SHORT_ERR_CONN_MOS
STD_ERR_CONN_MOS

## diff_gnds

Checks to make sure that components on a signal that are connected to ground symbols are not connected to different ground symbols.

### Required Data

If you are using parts from a non-Cadence library, you must define the GROUND_SYMBOL macro in the `cp_config.h` file. The default values are GND, GND1, and GND2.

### Assumptions

The rule assumes that different ground symbol names refer to different ground signals.

### Default Variables

None

### Default Severity

Error

### User Customization Information

You can customize the global macros, reported severity, and error messages for this rule.

### Global Macros

| Macro | Description |
|---|---|
| GROUND_SYMBOL | Comma-separated list of ground pins/nets found on components in the design. |

### Reported Severity

| Macro | Default |
|---|---|
| DIFF_GNDS_SEVERITY | Error |

#### *Error Messages*

STD_SHORT_ERR_DIFF_GNDS
STD_ERR_DIFF_GNDS

## diff_vcc

Checks to make sure that components on a pin that are connected to power symbols are not connected to different power symbols.

### Required Data

If you are using parts from a non-Cadence library, you must define the POWER_SYMBOL macro in the `cp_config.h` file. The default values are VCC, VDD, VEE, VEE1, VEE2, and VSS.

### Assumptions

The rule assumes that different power symbol names correspond to different signals.

### Default Variables

None

### Default Severity

Error

### User Customization Information

You can customize the global macros, reported severity, and error messages for this rule.

### Global Macros

| Macro | Description |
|---|---|
| POWER_SYMBOL | Comma-separated list of power pins/nets found on components in the design. |

### Reported Severity

| Macro | Default |
|---|---|
| DIFF_VCC_SEVERITY | Error |

#### *Error Messages*

STD_SHORT_ERR_DIFF_VCC
STD_ERR_DIFF_VCC

## ecl_non_ecl

Checks to make sure that every ECL gate that drives a non-ECL gate does so through a level shifter. The rule checks each signal in your design to make sure that if it contains at least one ECL output, then the total number of pins on the signal is equal to the total number of ECL input pins.

### Required Data

You must

- Define the ECL_TECH macro (defined in the `cp_global.h`). The default values are 100K,10K,10KH,100E.
- Define the PASSIVE_COMP macro in the `cp_config.h`.The default value (defined in `electrical_checks.h)` is RES.
- Specify the TECH property (using a value specified for ECL_TECH) on the appropriate components on your schematic to identify them as CMOS components.

### Assumptions

The rule assumes that the level shifter will have the TECH property attached to its ECL pins.

### Default Variables

None

### Default Severity

Error

### User Customization Information

You can customize the global macros, variable macro, reported severity, and error messages for this rule.

### Global Macros

| Macro | Description |
|---|---|
| ECL_TECH | Allowed technologies for ECL |

### Variable Macros

| Macro | Description |
|---|---|
| PASSIVE_COMP | Allowed passive components |

### Reported Severity

| Macro | Default |
|---|---|
| ECL_NON_ECL_SEVERITY | Error |

### *Error Messages*

STD_SHORT_ERR_ECL_NON_ECL
STD_ERR_ECL_NON_ECL

## ecl_oe_no_pull_down

Checks that all output/inout instterms of ECL instances are connected to a pull-down resistor. The rule flags an error in the following cases:

- if the instterm is not connected
- if the instterm is directory connected to power/ground
- if the instterm does not have a resistor connected to it
- if the instterm has a pulldown resistor connected to GND and no parallel resistor
- if the instterm has a resistor that is not connected to either VEE or VT.

### Required Data

You must specify a TECH=ECL_TECH property on open emitters of ECL gates in your design.

If you are using parts from a non-Cadence library, you must

- Define the GROUND_SYMBOL macro in the `cp_config.h` file. The default values are GND, GND1, and GND2.
- Define the POWER_SYMBOL macro in the `cp_config.h` file. The default values are VCC, VDD, VEE, VEE1, VEE2, and VSS.
- Define the VT macro in the `cp_config.h` file.

  The default value is VT.
- Define the VEE macro in the `cp_config.h` file.

  The default value is VEE.

- Specify the TECH property (using a value from `cp_global.h`) on the appropriate components on your schematic to identify them as ECL components.
- Specify the property name (RES_PROP_NAME) and value (RES_PROP_VALUE) used specify resistors.

  The default value of RES_PROP_NAME is COMP_TYPE, and the default for RES_PROP_VALUE is RES.

### Assumptions

None

### Default Variables

None

### Default Severity

Error

### *User Customization Information*

You can customize the global macros, reported severity, and error messages for this rule.

### Global Macros

| Macro | Description |
|---|---|
| ECL_TECH | Allowed technologies for ECL |
| GROUND_SYMBOL | Comma-separated list of ground pins/nets found on components in the design. |
| POWER_SYMBOL | Comma-separated list of power pins/nets found on components in the design. |
| VEE | Voltage for ECL components. |
| VT | Threshold voltage for ECL components. |
| RES_PROP_NAME | Property name used to identify resistor. |
| RES_PROP_VALUE | Value of RES_PROP_NAME property. |

### Reported Severity

| Macro | Default |
|---|---|
| ECL_OE_NO_PULL_DOWN_SEVERITY | Error |

#### *Error Messages*

STD_SHORT_ERR_ECL_OE_NO_PULL_DOWN
STD_ERR_ECL_OE_NO_PULL_DOWN_1 STD_ERR_ECL_OE_NO_PULL_DOWN_2
STD_ERR_ECL_OE_NO_PULL_DOWN_3

## hmos_ac

Checks that all HMOS signals that drive AC components are connected to a pull-up resistor.

### Required Data

If you are using parts from a non-Cadence library, you must

- Define the POWER_SYMBOL macro in the `cp_config.h` file. The default values are VCC, VDD, VEE, VEE1, VEE2, and VSS.
- Specify the TECH property (using a value from `cp_global.h`) on the appropriate components on your schematic to identify them as HMOS or AC components.
- Specify the property name (RES_PROP_NAME) and value (RES_PROP_VALUE) used specify resistors.

  The default value of RES_PROP_NAME is COMP_TYPE, and the default for RES_PROP_VALUE is RES.

### Assumptions

None

### Default Variables

None

### Default Severity

Error

### User Customization Information

You can customize the global macros, reported severity, and error messages for this rule.

### Global Macros

| Macro | Description |
|---|---|
| HMOS_TECH | Allowed technologies for HMOS |
| AC_TECH | Allowed technologies for AC |
| POWER_SYMBOL | Comma-separated list of power pins/nets found on components in the design |
| RES_PROP_NAME | Property name used to identify resistor. |
| RES_PROP_VALUE | Value of RES_PROP_NAME property. |

### Reported Severity

| Macro | Default |
|---|---|
| HMOS_AC_SEVERITY | Error |

#### *Error Messages*

STD_SHORT_ERR_HMOS_AC1 STD_SHORT_ERR_HMOS_AC2
STD_ERR_HMOS_AC1
STD_ERR_HMOS_AC2 STD_ERR_HMOS_AC3

## hmos_cmos

Checks that all HMOS signals that drive CMOS components are connected to a pull-up resistor.

### Required Data

If you are using parts from a non-Cadence library, you must do the following:

- Define the POWER_SYMBOL macro in the `cp_config.h` file. The default values are VCC, VDD, VEE, VEE1, VEE2, and VSS.
- Specify the TECH property (using a value from `cp_global.h`) on the appropriate components on your schematic to identify them as HMOS or CMOS components.
- Specify the property name (RES_PROP_NAME) and value (RES_PROP_VALUE) used specify resistors.

  The default value of RES_PROP_NAME is COMP_TYPE, and the default for RES_PROP_VALUE is RES.

### Assumptions

None

### Default Variables

None

### Default Severity

Error

### User Customization Information

You can customize the global macros, reported severity, and error messages for this rule.

### Global Macros

| Macro | Description |
|---|---|
| HMOS_TECH | Allowed technologies for HMOS |
| CMOS_TECH | Allowed technologies for CMOS |
| POWER_SYMBOL | Comma-separated list of power pins/nets found on components in the design |
| RES_PROP_NAME | Property name used to identify resistor |
| RES_PROP_VALUE | Value of RES_PROP_NAME property |

### Reported Severity

| Macro | Default |
|---|---|
| HMOS_CMOS_SEVERITY | Error |

#### *Error Messages*

STD_SHORT_ERR_HMOS_CMOS1 STD_SHORT_ERR_HMOS_CMOS2
STD_ERR_HMOS_CMOS1
STD_ERR_HMOS_CMOS2
STD_ERR_HMOS_CMOS3

## hmos_hc

Checks that all HMOS signals that drive HC components are connected to a pull-up resistor.

### Required Data

If you are using parts from a non-Cadence library, you must

- Define the POWER_SYMBOL macro in the `cp_config.h` file.

  The default values are VCC, VDD, VEE, VEE1, VEE2, and VSS.

- Specify the TECH property (using a value from `cp_global.h`) on the appropriate components on your schematic to identify them as HMOS or HC components.

- Specify the property name (RES_PROP_NAME) and value (RES_PROP_VALUE) used specify resistors.

  The default value of RES_PROP_NAME is COMP_TYPE, and the default for RES_PROP_VALUE is RES.

### Assumptions

None

### Default Variables

None

### Default Severity

Error

### User Customization Information

You can customize the global macros, reported severity, and error messages for this rule.

### Global Macros

| Macro | Description |
|---|---|
| HMOS_TECH | Allowed technologies for HMOS |
| HC_TECH | Allowed technologies for HC |
| POWER_SYMBOL | Comma-separated list of power pins/nets found on components in the design |
| RES_PROP_NAME | Property name used to identify resistor. |
| RES_PROP_VALUE | Value of RES_PROP_NAME property. |

### Reported Severity

| Macro | Default |
|---|---|
| HMOS_HC_SEVERITY | Error |

#### *Error Messages*

STD_SHORT_ERR_HMOS_HC1 STD_SHORT_ERR_HMOS_HC2
STD_ERR_HMOS_HC1
STD_ERR_HMOS_HC2
STD_ERR_HMOS_HC3

## illegal_voltage_power

Checks that the power pin (both on body and from `chips.prt`) of every component in your design is connected to the correct supply voltage.

### Required Data

If you are using parts from a non-Cadence library you must define the following in `cp_config.h`:

- The POWER_SYMBOL macro

  The default values are VCC, VDD, VEE, VEE1, VEE2, and VSS.

- The GROUND_SYMBOL macro

  The default values are GND, GND1, and GND2.

- The DEVICE_VOLT macro
- The SUPPLY_VOLT macro

You also must specify the TECH property as appropriate in the `chips_prt` file.

### Assumptions

None

### Default Variables

None

### Default Severity

Error

### User Customization Information

You can customize the global macros, reported severity, and error messages for this rule.

### Global Macros

| Macro | Description |
|---|---|
| GROUND_SYMBOL | Comma-separated list of ground pins/nets found on components in the design |
| POWER_SYMBOL | Comma-separated list of power pins/nets found on components in the design |

### Global Macros

| Macro | Description |
|---|---|
| SUPPLY_VOLT | Supply voltage |
| DEVICE_VOLT | Device voltage |

### Reported Severity

| Macro | Default |
|---|---|
| ILLEGAL_VOLTAGE_SEVERITY | Error |

#### *Error Messages*

STD_SHORT_ERR_ILLEGAL_VOLTAGE_POWER
STD_ERR_ILLEGAL_VOLTAGE_POWER
STD_SHORT_ERR_ILLEGAL_VOLTAGE_POWER2
STD_ERR_ILLEGAL_VOLTAGE_POWER2
STD_SHORT_ERR_ILLEGAL_VOLTAGE_POWER3
STD_ERR_ILLEGAL_VOLTAGE_POWER3
STD_SHORT_ERR_ILLEGAL_VOLTAGE_POWER4
STD_ERR_ILLEGAL_VOLTAGE_POWER4
STD_SHORT_ERR_ILLEGAL_VOLTAGE_POWER5
STD_ERR_ILLEGAL_VOLTAGE_POWER5

## in_out_count

Reports the number of fanins and fanouts for every net in your design. If a net is connected to any bidirectional pins, the number of bidirectional pins is also reported.

### Required Data

None

### Assumptions

None

### Default Variables

None

### Default Severity

Info

### User Customization Information

You can customize the reported severity and error messages for this rule.

### Reported Severity

| Macro | Default |
|---|---|
| IN_OUT_COUNT_SEVERITY | Info |

#### *Error Messages*

STD_SHORT_ERR_IN_OUT_COUNT STD_ERR_IN_OUT_COUNT_1
STD_ERR_IN_OUT_COUNT_2

## mos_open_inputs

Checks each MOS component in your design to ensure they do not have any open inputs.

### Required Data

If you are using a non-Cadence library, you must specify the TECH property (using a value from `cp_global.h`) on the appropriate components to define them as MOS components.

### Assumptions

None

### Default Variables

None

### Default Severity

Warning

### User Customization Information

You can customize the global macros, reported severity, and error messages for this rule:

### Global Macros

| Macro | Description |
|---|---|
| MOS_TECH | Allowed technologies for MOS |
| POWER_SYMBOL | Comma-separated list of power pins/nets found on components in the design |
| GROUND_SYMBOL | Comma-separated list of ground pins/nets found on components in the design |

### Reported Severity

| Macro | Default |
|---|---|
| MOS_OPEN_INPUTS_SEVERITY | Warning |

#### *Error Messages*

STD_SHORT_ERR_MOS_OPEN_INPUTS STD_ERR_MOS_OPEN_INPUTS

## nmos_ac

Checks that all NMOS signals that drive AC components are connected to a pull-up resistor. It also checks that no NMOS signals attached to AC components are directly connected to power symbols.

### Required Data

If you are using parts from a non-Cadence library, you must

- Define the POWER_SYMBOL macro in the `cp_config.h` file.

  The default values are VCC, VDD, VEE, VEE1, VEE2, and VSS.

- Specify the TECH property (using a value from `cp_global.h`) on the appropriate components on your schematic to identify them as NMOS or AC components.
- Specify the property name (RES_PROP_NAME) and value (RES_PROP_VALUE) used specify resistors.

  The default value of RES_PROP_NAME is COMP_TYPE, and the default for RES_PROP_VALUE is RES.

### Assumptions

None

### Default Variables

None

### Default Severity

Error

### User Customization Information

You can customize the global macros, reported severity, and error messages for this rule.

### Global Macros

| Macro | Description |
|---|---|
| NMOS_TECH | Allowed technologies for NMOS |
| AC_TECH | Allowed technologies for AC |
| POWER_SYMBOL | Comma-separated list of power pins/nets found on components in the design |
| RES_PROP_NAME | Property name used to identify resistor. |
| RES_PROP_VALUE | Value of RES_PROP_NAME property. |

### Reported Severity

| Macro | Default |
|---|---|
| NMOS_AC_SEVERITY | Error |

#### *Error Messages*

STD_SHORT_ERR_NMOS_AC1 STD_SHORT_ERR_NMOS_AC2
STD_ERR_NMOS_AC1
STD_ERR_NMOS_AC2
STD_ERR_NMOS_AC3

## nmos_cmos

Checks that all NMOS signals that drive CMOS components are connected to a pull-up resistor. It also checks that no NMOS signals attached to CMOS components are directly connected to power symbols.

### Required Data

If you are using parts from a non-Cadence library, you must

- Define the POWER_SYMBOL macro in the `cp_config.h` file.

  The default values are VCC, VDD, VEE, VEE1, VEE2, and VSS.

- Specify the TECH property (using a value from `cp_global.h`) on the appropriate components on your schematic to identify them as NMOS or CMOS components.

- Specify the property name (RES_PROP_NAME) and value (RES_PROP_VALUE) used specify resistors.

  The default value of RES_PROP_NAME is COMP_TYPE, and the default for RES_PROP_VALUE is RES.

### Assumptions

None

### Default Variables

None

### Default Severity

Error

### User Customization Information

You can customize the global macros, reported severity, and error messages for this rule.

### Global Macros

| Macro | Description |
|---|---|
| NMOS_TECH | Allowed technologies for NMOS |
| CMOS_TECH | Allowed technologies for CMOS |
| POWER_SYMBOL | Comma-separated list of power pins/nets found on components in the design |
| RES_PROP_NAME | Property name used to identify resistor. |
| RES_PROP_VALUE | Value of RES_PROP_NAME property. |

### Reported Severity

| Macro | Default |
|---|---|
| NMOS_CMOS_SEVERITY | Error |

### *Error Messages*

STD_SHORT_ERR_NMOS_CMOS1 STD_SHORT_ERR_NMOS_CMOS2
STD_ERR_NMOS_CMOS1
STD_ERR_NMOS_CMOS2
STD_ERR_NMOS_CMOS3

## nmos_hc

Checks that all NMOS signals that drive HC components are connected to a pull-up resistor. It also checks that no NMOS signals attached to HC components are directly connected to power symbols.

### Required Data

If you are using parts from a non-Cadence library, you must

- Define the POWER_SYMBOL macro in the `cp_config.h` file. The default values are VCC, VDD, VEE, VEE1, VEE2, and VSS.
- Specify the TECH property (using a value from `cp_global.h`) on the appropriate components on your schematic to identify them as NMOS or HC components.
- Specify the property name (RES_PROP_NAME) and value (RES_PROP_VALUE) used specify resistors.

  The default value of RES_PROP_NAME is COMP_TYPE, and the default for RES_PROP_VALUE is RES.

### Assumptions

None

### Default Variables

None

### Default Severity

Error

### User Customization Information

You can customize the global macros, reported severity, and error messages for this rule.

### Global Macros

| Macro | Description |
|---|---|
| NMOS_TECH | Allowed technologies for NMOS |
| HC_TECH | Allowed technologies for HC |
| POWER_SYMBOL | Comma-separated list of power pins/nets found on components in the design |
| RES_PROP_NAME | Property name used to identify resistor. |
| RES_PROP_VALUE | Value of RES_PROP_NAME property. |

### Reported Severity

| Macro | Default |
|---|---|
| NMOS_HC_SEVERITY | Error |

#### *Error Messages*

STD_SHORT_ERR_NMOS_HC1 STD_SHORT_ERR_NMOS_HC2
STD_ERR_NMOS_HC1
STD_ERR_NMOS_HC2
STD_ERR_NMOS_HC3

## no_drive

Checks that each signal in the design has a drive.

### Required Data

If you are using non-Cadence libraries, you must define the INPUT_LOAD and OUTPUT_LOAD properties in your `chips_prt` file.

### Assumptions

The rule ignores signals included in the POWER_SYMBOL and GROUND_SYMBOL definitions. The rule ignores signals connected to pins that have properties defined by the name-value pair (PIN_PROP_TO_IGN, PIN_PROP_TO_IGN_VALUE). Also, pins that have both, INPUT_LOAD, and OUTPUT_LOAD properties, are assumed to be bidirectional pins. If a pin does not have either property, or if no `chips_prt` file exists for the part, the pin is assumed to be an input pin.

### Default Variables

None

### Default Severity

Error

### User Customization Information

You can customize the variable macros, global macros, reported severity, and error messages for this rule.

### Variable Macros

| Macro | Description |
|---|---|
| PIN_PROP_TO_IGN | Comma-separated list of pin property names |
| PIN_PROP_TO_IGN_ VALUE | Comma-separated list of pin property values |

### Global Macros

| Macro | Description |
|---|---|
| POWER_SYMBOL | Comma-separated list of power pins/nets found on components in the design |
| GROUND_SYMBOL | Comma-separated list of ground pins/nets found on components in the design |

### Reported Severity

| Macro | Default |
|---|---|
| NO_DRIVE_SEVERITY | Error |

#### *Error Messages*

STD_SHORT_ERR_NO_DRIVE
STD_ERR_NO_DRIVE

## no_load

Checks that each signal in the design has a load.

### Required Data

If you are using non-Cadence libraries, you must define the INPUT_LOAD and OUTPUT_LOAD properties in your `chips_prt` file.

### Assumptions

The rule assumes that pins with both INPUT_LOAD and OUTPUT_LOAD properties are assumed to be bidirectional pins. If a pin does not have either property, or if no `chips_prt` file exists for the part, then the pin is assumed to be an input pin.

### Default Variables

None

### Default Severity

Error

### User Customization Information

You can customize the reported severity and error messages for this rule.

### Reported Severity

| Macro | Default |
|---|---|
| NO_LOAD_SEVERITY | Error |

#### *Error Messages*

STD_SHORT_ERR_NO_LOAD
STD_ERR_NO_LOAD

## oc_bidi_connected

Checks that no open-collector/drain and bidirectional outputs on your design are connected to each other.

### Required Data

If you are using non-Cadence libraries, you must specify the TECH and OUTPUT_TYPE properties on the appropriate components in your schematic to identify open-collector/drain components.

### Assumptions

The rule assumes that the default OUTPUT_TYPE property value for MOS components is OC, and OD for CMOS components.

### Default Variables

None

### Default Severity

Error

### User Customization Information

You can customize the global macros, reported severity, and error messages for this rule.

### Global Macros

| Macro | Description |
|---|---|
| MOS_TECH | Allowed technologies for MOS |
| OC_OUTPUT_TYPE | Open-collector outputs |
| CMOS_TECH | Allowed technologies for CMOS |
| OD_OUTPUT_TYPE | Open-drain outputs |

### Reported Severity

| Macro | Default |
|---|---|
| OC_BIDI_CONNECTED_SEVERITY | Error |

#### *Error Messages*

STD_SHORT_ERR_OC_BIDI_CONNECTED STD_ERR_OC_BIDI_CONNECTED

## oc_bidits_connected

Checks that no open-collector/drain and bidirectional tri-state outputs on your design are connected to each other.

### Required Data

If you are using non-Cadence libraries, you must specify the TECH and OUTPUT_TYPE properties on the appropriate components in your schematic to identify open-collector/drain and tri-state components.

### Assumptions

The rule assumes that the default OUTPUT_TYPE property value for MOS components is OC, and is OD for CMOS components.

### Default Variables

None

### Default Severity

Error

### User Customization Information

You can customize the global macros, reported severity, and error messages for this rule.

### Global Macros

| Macro | Description |
|---|---|
| MOS_TECH | Allowed technologies for MOS |
| OC_OUTPUT_TYPE | Open-collector outputs |
| TS_OUTPUT_TYPE | Tri-state outputs |
| OD_OUTPUT_TYPE | Open-drain outputs |
| CMOS_TECH | Allowed technologies for CMOS |

### Reported Severity

| Macro | Default |
|---|---|
| OC_BIDITS_CONNECTED_SEVERITY | Error |

#### *Error Messages*

STD_SHORT_ERR_OC_BIDITS_CONNECTED STD_ERR_OC_BIDITS_CONNECTED

## oc_no_pull_up

Checks that all open-collector/drain terminals are connected to a pull-up resistor.

### Required Data

If you are using parts from a non-Cadence library, you must

- Define the POWER_SYMBOL macro in the `cp_config.h` file. The default values are VCC, VDD, VEE, VEE1, VEE2, and VSS.
- Specify the OUTPUT_TYPE property (using a value from `cp_global.h`) on the components on your schematic to identify them as open-collector/open-drain components.
- Specify the property name (RES_PROP_NAME) and value (RES_PROP_VALUE) used specify resistors.

  The default value of RES_PROP_NAME is COMP_TYPE, and the default for RES_PROP_VALUE is RES.

### Assumptions

None

### Default Variables

None

### Default Severity

Error

### User Customization Information

You can customize the global macros, reported severity, and error messages for this rule.

### Global Macros

| Macro | Description |
|---|---|
| OC_OUTPUT_TYPE | Open-collector outputs |
| OD_OUTPUT_TYPE | Open-drain outputs |
| POWER_SYMBOL | Comma-separated list of power pins/nets found on components in the design |

### Global Macros

| Macro | Description |
|---|---|
| RES_PROP_NAME | Property to identify resistor |
| RES_PROP_VALUE | Value of property RES_PROP_NAME |

### Reported Severity

| Macro | Default |
|---|---|
| OC_NO_PULL_UP_SEVERITY | Error |

#### *Error Messages*

STD_SHORT_ERR_OC_NO_PULL_UP STD_ERR_OC_NO_PULL_UP_1 STD_ERR_OC_NO_PULL_UP_2

## op_vcc_connected

Checks that the output of each component in the design is not connected directly to power.

### Required Data

If you are using parts from a non-Cadence library, you must define the POWER_SYMBOL macro in the `cp_config.h` file. The default values are VCC, VDD, VEE, VEE1, VEE2, and VSS.

### Assumptions

None

### Default Variables

None

### Default Severity

Error

### User Customization Information

You can customize the global macros, reported severity, and error messages for this rule.

### Global Macros

| Macro | Description |
|---|---|
| CONN_PROP_NAME | Allowed list of properties on components for which check will be skipped |
| CONN_PROP_VALUE | Allowed list of values on properties |
| POWER_SYMBOL | Comma-separated list of power pins/nets found on components in the design |

### Reported Severity

| Macro | Default |
|---|---|
| OP_VCC_CONNECTED_SEVERITY | Error |

#### *Error Messages*

STD_SHORT_ERR_OP_VCC_CONNECTED STD_ERR_OP_VCC_CONNECTED

## tech_notech

Checks that all HMOS, NMOS, and TTL signals that drive AC, HC, or CMOS components are connected to a pull-up resistor. It also checks that no HMOS, NMOS, or TTL signals attached to AC, HC, or CMOS are directly connected to power symbols.

### Required Data

If you are using parts from a non-Cadence library, you must

- Define the POWER_SYMBOL macro in the `cp_config.h` file.

  The default values are VCC, VDD, VEE, VEE1, VEE2, and VSS.

- Specify the TECH property (using a value from `cp_global.h`) on the appropriate components on your schematic to identify different technologies.

- Specify the property name (RES_PROP_NAME) and value (RES_PROP_VALUE) used specify resistors.

  The default value of RES_PROP_NAME is COMP_TYPE, and the default for RES_PROP_VALUE is RES.

### Assumptions

None

### Default Variables

None

### Default Severity

Error

### User Customization Information

You can customize the global macros, reported severity, and error messages for this rule.

### Global Macros

| Macro | Description |
|---|---|
| TTL_TECH | Allowed technologies for TTL |
| HMOS_TECH | Allowed technologies for HMOS |
| NMOS_TECH | Allowed technologies for NMOS |
| CMOS_TECH | Allowed technologies for CMOS |
| HC_TECH | Allowed technologies for HC |
| AC_TECH | Allowed technologies for AC |
| RES_PROP_NAME | Property name used to identify resistor |
| RES_PROP_VALUE | Value of RES_PROP_NAME property |

### Reported Severity

| Macro | Default |
|---|---|
| TECH_NOTECH_SEVERITY | Error |

#### *Error Messages*

STD_SHORT_ERR_TECH_NOTECH1 STD_SHORT_ERR_TECH_NOTECH2
STD_ERR_TECH_NOTECH1
STD_ERR_TECH_NOTECH2
STD_ERR_TECH_NOTECH3

## tp_bidi_connected

Checks that no totem-pole outputs and bidirectional outputs on your design are connected to each other.

### Required Data

If you are using non-Cadence libraries, you must specify the TECH and OUTPUT_TYPE properties in the `chips_prt` file.

### Assumptions

The rule assumes that any pin that belongs to the TTL family and does not have an OUTPUT_TYPE property is a totem-pole output.

### Default Variables

None

### Default Severity

Error

### User Customization Information

You can customize the global macros, reported severity, and error messages for this rule.

### Global Macros

| Macro | Description |
|---|---|
| TTL_TECH | Allowed technologies for TTL |

### Reported Severity

| Macro | Default |
|---|---|
| TP_BIDI_CONNECTED_SEVERITY | Error |

#### *Error Messages*

STD_SHORT_ERR_TP_BIDI_CONNECTED STD_ERR_TP_BIDI_CONNECTED

## tp_oc_connected

Checks that no totem-pole outputs and open-collector outputs on your design are connected to each other.

### Required Data

If you are using non-Cadence libraries, you must specify the TECH and OUTPUT_TYPE properties.

### Assumptions

The rule assumes

- Any pin that belongs to the TTL family and does not have an OUTPUT_TYPE property is a totem-pole output.
- The default OUTPUT_TYPE of MOS tech is OC.

### Default Variables

None

### Default Severity

Error

### User Customization Information

You can customize the global macros, reported severity, and error messages for this rule.

### Global Macros

| Macro | Description |
|---|---|
| TTL_TECH | Allowed technologies for TTL |
| OC_OUTPUT_TYPE | Open collector output |
| MOS_TECH | Allowed technologies for MOS |

### Reported Severity

| Macro | Default |
|---|---|
| TP_OC_CONNECTED_SEVERITY | Error |

#### *Error Messages*

STD_SHORT_ERR_TP_OC_CONNECTED STD_ERR_TP_OC_CONNECTED

## tp_tp_connected

Checks that no two totem-pole outputs in your design are connected to each other.

### Required Data

If you are using non-Cadence libraries, you must specify the TECH property to identify totem-pole outputs.

### Assumptions

The rule assumes that any pin that belongs to the TTL family and does not have an OUTPUT_TYPE property is a totem-pole output.

### Default Variables

None

### Default Severity

Error

### User Customization Information

You can customize the global macros, reported severity, and error messages for this rule.

### Global Macros

| Macro | Description |
|---|---|
| TTL_TECH | Allowed technologies for TTL |

### Reported Severity

| Macro | Default |
|---|---|
| TP_TP_CONNECTED_SEVERITY | Error |

#### *Error Messages*

STD_SHORT_ERR_TP_TP_CONNECTED STD_ERR_TP_TP_CONNECTED

## tp_ts_connected

Checks that no totem-pole outputs and tri-state outputs in your design are connected to each other.

### Required Data

If you are using non-Cadence libraries, you must specify the TECH and OUTPUT_TYPE properties to identity tri-state and totem-pole outputs.

### Assumptions

The rule assumes that any pin that belongs to the TTL family and does not have an OUTPUT_TYPE property is a totem-pole output.

### Default Variables

None

### Default Severity

Error

### User Customization Information

You can customize the global macros, reported severity, and error messages for this rule.

### Global Macros

| Macro | Description |
|---|---|
| TTL_TECH | Allowed technologies for TTL |
| TS_OUTPUT_TYPE | Tri-state output |

### Reported Severity

| Macro | Default |
|---|---|
| TP_TS_CONNECTED_SEVERITY | Error |

### *Error Messages*

STD_SHORT_ERR_TP_TS_CONNECTED STD_ERR_TP_TS_CONNECTED

## ts_bidi_connected

Checks that no tri-state outputs and bidirectional outputs (including bidirectional tri-state outputs) in your design are connected to each other.

### Required Data

If you are using non-Cadence libraries, you must specify the OUTPUT_TYPE properties in the `chips_prt` file.

### Assumptions

None

### Default Variables

None

### Default Severity

Error

### User Customization Information

You can customize the global macros, reported severity, and error messages for this rule.

### Global Macros

| Macro | Description |
|---|---|
| TS_OUTPUT_TYPE | Tri-state output |

### Reported Severity

| Macro | Default |
|---|---|
| TS_BIDI_CONNECTED_SEVERITY | Error |

### *Error Messages*

STD_SHORT_ERR_TS_BIDI_CONNECTED STD_ERR_TS_BIDI_CONNECTED

## ts_bidits_connected

Checks that no tri-state outputs and bidirectional tri-state outputs in your design are connected to each other.

### Required Data

If you are using a non-Cadence library, you must specify the OUTPUT_TYPE properties in the `chips_prt` file.

### Assumptions

None

### Default Variables

None

### Default Severity

Error

### User Customization Information

You can customize the global macros, reported severity, and error messages for this rule.

### Global Macros

| Macro | Description |
|---|---|
| TS_OUTPUT_TYPE | Tri-state output |

### Reported Severity

| Macro | Default |
|---|---|
| TS_BIDITS_CONNECTED_SEVERITY | Error |

#### *Error Messages*

STD_SHORT_ERR_TS_BIDITS_CONNECTED
STD_SHORT_ERR_TS_BIDITS_CONNECTED2 STD_ERR_TS_BIDITS_CONNECTED
STD_ERR_TS_BIDITS_CONNECTED2

## ts_oc_connected

Checks that no tri-state outputs and open-collector outputs in your design are connected to each other.

### Required Data

If you are using non-Cadence libraries, you must specify the OUTPUT_TYPE properties on the appropriate components in your schematic.

### Assumptions

The rule assumes that the default OUTPUT_TYPE property value for MOS components is OC.

### Default Variables

None

### Default Severity

Error

### User Customization Information

You can customize the global macros, reported severity, and error messages for this rule.

### Global Macros

| Macro | Description |
|---|---|
| MOS_TECH | Allowed technologies for MOS |
| OC_OUTPUT_TYPE | Open collector outputs |
| TS_OUTPUT_TYPE | Tri-state outputs |

### Reported Severity

| Macro | Default |
|---|---|
| TS_OC_CONNECTED_SEVERITY | Error |

#### *Error Messages*

STD_SHORT_ERR_TS_OC_CONNECTED STD_ERR_TS_OC_CONNECTED

## ttl_ac

Checks that all TTL signals that drive AC components are connected to a pull-up resistor. It also checks that no TTL signals attached to AC components are directly connected to power symbols.

### Required Data

If you are using parts from a non-Cadence library, you must

- Define the POWER_SYMBOL macro in the `cp_config.h` file.

  The default values are VCC, VDD, VEE, VEE1, VEE2, and VSS.
- Specify the TECH property (using a value from `cp_global.h`) on the appropriate components on your schematic to identify them as TTL or AC components.
- Specify the COMP_TYPE=RES property on resistors in your design to designate them.

- Specify the property name (RES_PROP_NAME) and value (RES_PROP_VALUE) used specify resistors.

  The default value of RES_PROP_NAME is COMP_TYPE, and the default for RES_PROP_VALUE is RES.

### Assumptions

None

### Default Variables

None

### Default Severity

Error

### User Customization Information

You can customize the global macros, reported severity, and error messages for this rule.

### Global Macros

| Macro | Description |
|---|---|
| TTL_TECH | Allowed technologies for TTL |
| AC_TECH | Allowed technologies for AC |
| POWER_SYMBOL | Comma-separated list of power pins/nets found on components in the design |
| RES_PROP_NAME | Property to identify resistor |
| RES_PROP_VALUE | Value of property RES_PROP_NAME |

### Reported Severity

| Macro | Default |
|---|---|
| TTL_AC_SEVERITY | Error |

***Error Messages***

STD_SHORT_ERR_TTL_AC1
STD_SHORT_ERR_TTL_AC2
STD_ERR_TTL_AC1
STD_ERR_TTL_AC2
STD_ERR_TTL_AC3

## ttl_cmos

Checks that all TTL signals that drive CMOS components are connected to a pull-up resistor. It also checks that no TTL signals attached to CMOS components are directly connected to power symbols.

### Required Data

If you are using parts from a non-Cadence library, you must

- Define the POWER_SYMBOL macro in the `cp_config.h` file.

  The default values are VCC, VDD, VEE, VEE1, VEE2, and VSS.

- Specify the TECH property (using a value from `cp_global.h`) on the appropriate components on your schematic to identify them as TTL or CMOS components.
- Specify the COMP_TYPE=RES property on resistors in your design to designate them.
- Specify the property name (RES_PROP_NAME) and value (RES_PROP_VALUE) used specify resistors.

  The default value of RES_PROP_NAME is COMP_TYPE, and the default for RES_PROP_VALUE is RES.

### Assumptions

None

### Default Variables

None

### Default Severity

Error

### User Customization Information

You can customize the global macros, reported severity, and error messages for this rule.

### Global Macros

| Macro | Description |
|---|---|
| TTL_TECH | Allowed technologies for TTL |
| CMOS_TECH | Allowed technologies for CMOS POWER_SYMBOL |
| POWER_SYMBOL | Comma-separated list of power signals/nets found on components in the design |
| RES_PROP_NAME | Property to identify resistor |
| RES_PROP_VALUE | Value of property RES_PROP_NAME |

### Reported Severity

| Macro | Default |
|---|---|
| TTL_CMOS_SEVERITY | Error |

#### *Error Messages*

STD_SHORT_ERR_TTL_CMOS1 STD_SHORT_ERR_TTL_CMOS2
STD_ERR_TTL_CMOS1
STD_ERR_TTL_CMOS2
STD_ERR_TTL_CMOS3

## ttl_hc

Checks that all TTL signals that drive HC components are connected to a pull-up resistor. It also checks that no TTL signals attached to HC components are directly connected to power symbols.

### Required Data

If you are using parts from a non-Cadence library, you must

- Define the POWER_SYMBOL macro in the `cp_config.h` file.

  The default values are VCC, VDD, VEE, VEE1, VEE2, and VSS.

- Specify the TECH property (using a value from `cp_global.h`) on the appropriate components on your schematic to identify them as TTL or HC components.

- Specify the property name (RES_PROP_NAME) and value (RES_PROP_VALUE) used specify resistors.

  The default value of RES_PROP_NAME is COMP_TYPE, and the default for RES_PROP_VALUE is RES.

### Assumptions

None

### Default Variables

None

### Default Severity

Error

### User Customization Information

You can customize the global macros, reported severity, and error messages for this rule.

### Global Macros

| Macro | Description |
|---|---|
| TTL_TECH | Allowed technologies for TTL |
| HC_TECH | Allowed technologies for HC |
| POWER_SYMBOL | Comma-separated list of power pins/nets found on components in the design |
| RES_PROP_NAME | Property to identify resistor |

### Global Macros

| Macro | Description |
|---|---|
| RES_PROP_VALUE | Value of property RES_PROP_NAME |

### Reported Severity

| Macro | Default |
|---|---|
| TTL_HC_SEVERITY | Error |

#### *Error Messages*

STD_SHORT_ERR_TTL_HC1
STD_SHORT_ERR_TTL_HC2
STD_ERR_TTL_HC1
STD_ERR_TTL_HC2
STD_ERR_TTL_HC3

## ttl_open_inputs

Checks each TTL component in your design to ensure they do not have any open inputs.

### Required Data

If you are using a non-Cadence library, you must specify the TECH property (using a value from `cp_global.h`) on the appropriate components to define them as TTL components.

You must also define the following in `cp_config.h`:

- The POWER_SYMBOL macro.

  The default values are VCC, VDD, VEE, VEE1, VEE2, and VSS.

- The GROUND_SYMBOL macro.

  The default values are GND, GND1, and GND2.

- The SUPPLY_VOLT macro.

### Assumptions

None

### Default Variables

None

### Default Severity

Warning

### User Customization Information

You can customize the global macros, reported severity, and error messages for this rule.

### Global Macros

| Macro | Description |
|---|---|
| TTL_TECH | Allowed technologies for TTL |
| GROUND_SYMBOL | Comma-separated list of ground pins/nets found on components in the design |
| POWER_SYMBOL | Comma-separated list of power pins/nets found on components in the design |

### Reported Severity

| Macro | Default |
|---|---|
| TTL_OPEN_INPUTS_SEVERITY | Warning |

### *Error Messages*

STD_SHORT_ERR_TTL_OPEN_INPUTS STD_ERR_TTL_OPEN_INPUTS

## undefined_voltage

Checks that all power/ground symbols used in the design have the supply-voltage value specified in the SUPPLY_VOLT macro as well.

### Required Data

If you are using parts from a non-Cadence library, you must define the following in `cp_config.h`:

- The POWER_SYMBOL macro

  The default values are VCC, VDD, VEE, VEE1, VEE2, and VSS.

- The GROUND_SYMBOL macro

  The default values are GND, GND1, and GND2.

- The SUPPLY_VOLT macro

### Assumptions

None

### Default Variables

None

### Default Severity

Error

### User Customization Information

You can customize the global macros, reported severity, and error messages for this rule.

### Global Macros

| Macro | Description |
|---|---|
| SUPPLY_VOLT | Supply voltage |

### Global Macros

| Macro | Description |
|---|---|
| GROUND_SYMBOL | Comma-separated list of ground pins/nets found on components in the design |
| POWER_SYMBOL | Comma-separated list of power pins/nets found on components in the design |

### Reported Severity

| Macro | Default |
|---|---|
| UNDEFINED_VOLTAGE_SEVERITY | Error |

#### *Error Messages*

STD_SHORT_ERR_UNDEFINED_VOLTAGE STD_ERR_UNDEFINED_VOLTAGE

## vcc_gnd_shorted

Checks that power and ground signals in your design are not shorted.

### Required Data

If you are using parts from non-Cadence libraries, you must define the following in the `cp_config.h` file:

- The POWER_SYMBOL macro

  The default values are VCC, VDD, VEE, VEE1, VEE2, and VSS.

- The GROUND_SYMBOL macro

  The default values are GND, GND1, and GND2.

### Assumptions

None

### Default Variables

None

### Default Severity

Error

### User Customization Information

You can customize the global macros, reported severity, and error messages for this rule.

### Global Macros

| Macro | Description |
|---|---|
| GROUND_SYMBOL | Comma-separated list of ground pins/nets found on components in the design |
| POWER_SYMBOL | Comma-separated list of power pins/nets found on components in the design |

### Reported Severity

| Macro | Default |
|---|---|
| VCC_GND_SHORTED_SEVERITY | Error |

#### *Error Messages*

STD_SHORT_ERR_VCC_GND_SHORTED STD_ERR_VCC_GND_SHORTED

## vcc_vcc_shorted

Checks that the power signals in your design are not shorted.

### Required Data

If you are using parts from non-Cadence libraries, you must define the following in the `cp_config.h` file:

- The POWER_SYMBOL macro

  The default values are VCC, VDD, VEE, VEE1, VEE2, and VSS.

- The GROUND_SYMBOL macro

  The default values are GND, GND1, and GND2.

### Assumptions

None

### Default Variables

None

### Default Severity

Error

### User Customization Information

You can customize the global macros, reported severity, and error messages for this rule.

### Global Macros

| Macro | Description |
|---|---|
| POWER_SYMBOL | Comma-separated list of power pins/nets found on components in the design |

### Reported Severity

| Macro | Default |
|---|---|
| VCC_VCC_SHORTED_SEVERITY | Error |

### *Error Messages*

STD_ERR_VCC_GND_SHORTED

## signal_names_with_spaces

Checks if signal names have spaces in them.

### Required Data

If you are using parts from non-Cadence libraries, you must define the following in the `cp_config.h` file:

- The POWER_SYMBOL macro

  The default values are VCC, VDD, VEE, VEE1, VEE2, and VSS.

- The GROUND_SYMBOL macro

  The default values are GND, GND1, and GND2.

### Assumptions

None

### Default Variables

None

### Default Severity

Error

### User Customization Information

You can customize the global macros, reported severity, and error messages for this rule.

### Global Macros

| Macro | Description |
|---|---|
| POWER_SYMBOL | Comma-separated list of power pins/nets found on components in the design |

### Reported Severity

| Macro | Default |
|---|---|
| VCC_VCC_SHORTED_SEVERITY | Error |

### *Error Messages*

STD_ERR_VCC_GND_SHORTED

# B

# Rule Language Reference

## ToolStart Predicates

Forking predicates enable you to spawn off Cadence or third-party tools from within Rules Checker. This ensures that you can use existing tools if you have a common verification engine. You can use Rules Checker as the standard interface for all verifications. With the help of toolStart predicates, you can invoke other verification tools also from the Rules Checker interface.

This section describes the three forking predicates:

- `toolStartPass`
- `toolStartFail`
- `toolStartIgn`

These predicates are invoked on objects. Depending on the type of predicate used, the object (if used) or string is returned.

### Usage for Single Objects

The usage for single objects is:

```
toolStart{Pass/Fail/Ignore}DobjectIds,Call_string)
```

where the three predicates are

- `Pass`

  Returns the objects on which the tool passes; that is, the exit status is 0.

- `Fail`

  Returns the objects on which the tool fails; that is, the exit status is 1.

- `Ign`

The return status of the tool invoked is ignored and it is assumed to have returned Pass.

You construct the call string based on the requirements. The first string in the call string must be `toolName` (with or without the path). Here is a call string example:

```
[path/]toolName + opt_str1 + %s + opt_str2
```

where `%s` is the place holder for *`object_name`*. The name of the object on which the iteration is taking place is printed here.

## Usage for Lists

The usage for lists is as follows:

```
toolStart{pass/Fail/Ignore}(call_string)
```

The significance of `pass/Fail/Ignore` is the same as that of toolStart in the previous section. The user can generate `call_string` on the following lines:

```
[path/]toolName + opt_str1 + concat_name(objList) +
opt_str2
```

`concat_name` is a predicate that concatenates all names in `objList` in order to generate one large list. The tool then runs on the list of objects and returns `call_string` according to its type namely `pass/Fail/Ignore`.

You can use this predicate to return individual objects with the help of foreach.

You can use toolStart in cases where there are no objects. You can also use these predicates when you do not want to do anything on objects.

## Examples of toolStart Predicates

```
toolStartPass("grep 54lsttl /net/sugandh/home/rajeshk/master.lib")
```

This will return non-NULL if the library `54lstt` is listed in `master.lib`.

```
toolStartFail(inst1,"/net/sugandh/home/rajeshk/bin/myScript %s " +
filename(inst1))
```

This will call `myScript`, taking the name of an instance and filename of the instance as arguments. The script processes and exits with status 0 for success, and non-zero for failure. Out of all the instances in the list `inst`, toolStartFail returns those for which `myScript` failed.

```
toolStartIgn(sig1, "AddToListOfSig %s")
```

This calls the script `AddToListOfSig` for all the signals in the list `sig1`. It passes the name of the signal to the script. The script then records the name of the signal in a file for future reference. The exit status of the script is ignored; note `Ign` in `toolStartIgn`.

# Data Types

In the Rules Checker Rule Language, you do not need to specify the type of a variable. Depending on the context, variables can assume values from one of the following basic data types.

| Data Type | Description |
|---|---|
| integer | integer |
| float | floating point number |
| value | character value |
| boolean | true or false |
| dbobject | database object (handle to object in the database) |

Variables whose names start with an object name are automatically considered object variables. An object is defined as part of the environment.

Example

If `inst` and `sig` are declared as objects

- `inst1` and `inst_abc` are object variables of type `inst`
- `sig1` and `sig_20` are object variables of type `sig`

Variables whose names do not start with an object name are considered non-object variables. Non-object variables can be any of the above data types except `dbobject`.

Example

`i`, `j`, and `xyz` are non-object variables.

Internally, all variables have identical storage: the data type of the variable, and a union of the above types.

# Language Constructs

This topic describes Rules Checker Rule Language constructs. The following notation is used in these descriptions:

- An item within curly brackets ({}) indicates 0 or more repetitions of the item.
- An item within curly brackets followed by a plus sign ({}+) indicates at least 1 or more occurrences of the item.
- An item within square brackets ([]) signifies 0 or 1 occurrences of the item.
- A vertical bar (|) denotes the OR of two items.

**Note:** If any of the characters ({}), ([]), (|) form a part of the syntax of the language, they are enclosed within single quotes (‘’).

## Keywords

Keywords are reserved words in the Rules Checker Rule Language. A keyword cannot be used as an identifier, or else the Rules Checker Toolkit compiler flags it as an error.

## Identifiers

Identifiers consist of alphanumeric characters, the underscore (_) and the dollar sign ($). The first character in an identifier cannot be a number (0..9).

## Case Sensitivity

Rules Checker Rule Language is case insensitive.

## Comments

All values between a “/*” and a “*/” are treated as comments. Nested comments are not allowed. All values after characters // until the end of the line are treated as comments.

# Include File/Parameter Files

Rules Checker supports customizing rules by the use of parameters that can be assigned different values. These parameters are stored in an include file that can be included in the

rulefile using include filename statement at the beginning of the rule file. By convention the include file has an extension of .h (such as, myparameterfile.h).

The parameters stored in the include file can be global that are used by more than one rule and local that are used by a single rule. The general structure for an include file is

```
STARTGLOBAL
parameter definitions
ENDGLOBAL

STARTENV <environment name>
STARTRULE <rulename>
parameter definitions
 ENDRULE
ENDENV
```

where parameter definitions can be specified in the following way:

- SEVERITY <*parameter-name*> Fatal|Error|OverSight|Warning|Info
- SHORT <*short-msg-name*> <*short-msg-val*>
- LONG <*long-msg-name*> <*long-msg-val*>
- PARAM <*parameter-name*> <*parameter-val*>

Each of the keywords SEVERITY, SHORT, LONG, PARAM are used to specify whether the parameter is a severity parameter, short message parameter, long message parameter or rule parameter.

STARTGLOBAL ENDGLOBAL section can only have rule parameters specified by PARAM keyword, whereas all four types of parameters can go in STARTRULE ENDRULE section. Global parameters can only be specified in cp_global.h and cp_config.h files. Local parameters can go in any .h file. Also an include file can have multiple STARTGLOBAL ENDGLOBAL or STARTENV ENDENV sections with each STARTENV ENDENV can have multiple STARTRULE ENDRULE sections.

# Rule File Structure

Rule files contain optional include directories that specify the files to be included, use directive specifying the environment, and definitions of rules. Any include path specified must be done before specifying the use directive. A rule file must always have a use directive

```
USE <env_name>;
     env_name := logical|physical|graphical|body
```

## Rule Definitions

Rule definitions contain user-defined rules, described in the Rules Checker Rule Language. The structure for a rule definition is:

```
RuleDefine
{rule_declaration }+
EndRuleDefine
```

where *rule_declaration* uses the following syntax:

```
Rule rule_name
     {condition_declaration message_predicate}+
     EndRule [rule_name]
```

*rule_name* is an identifier. *condition_declaration* and *message_predicate* are described later in this section.

The following statement needs to be put before the first Rule Define in the rule file:

```
Use {environment_name };
```

*environment_name* is the name of the environment (`body`, `graphical`, `logical` or `physical`) in which the rule file must be loaded.

If the rule file does not contain a `Use` statement, an error while compilation will be flagged.

### Basic Objects

Every rule has a basic object variable, which is the first object variable that has not been assigned a value in the condition_declaration section.

For example, in the following rule, `inst1` has not been assigned any values, so it is the basic object variable. Hence, *inst* is the basic object of the rule `basic`.

```
Rule basic
sig1 = sig(inst1) AND
name(sig1) /= "VCC"

    Message()
EndRule
```

When the basic object is a primary object, the basic object determines the number of iterations to be made in the rule.

For example, if `signal` is a primary object, use the following algorithm to count the number of instTerms connected to each signal in the design.

```
for each signal in the design
print the number of instTerms connected to it
```

The condition-message pair for this is

```
val1 = count(instTerm(sig1))
Message(Info,"Number of instterms connected to signal ?sig1 are ?val1");
```

**Note:** The for each loop is built into the execution unit.

In the above example, a message is output for each signal.

When the basic object is a secondary object, the Rules Checker Toolkit compiler adds a predicate that gets the secondary object from a primary object. This primary object determines the number of iterations to be made in the rule, as in the following example:

```
Rule secondary
    body1
    Message(...)
EndRule
```

If `body` is a secondary object, after parsing the rule the Rules Checker Toolkit compiler will traverse the environment to look for a primary object that has a unary predicate named `body`. This search is sequential, so the priority of primary objects is determined by their definition order in the environment file.

For example, if a rule uses the predicate `body(inst)`, where `inst` is a primary object, the compiler creates a data structure representing

```
body1 = body(inst)
```

and the number of iterations now depends on the number of instances in the design.

A rule cannot contain two independent objects. For example, the following is not allowed:

```
Rule err
    inst1 AND sig1

    Message(...)
EndRule
```

### Condition Declarations

Conditions are declared as expressions expressed in terms of predicates and operators of the language. The syntax for a condition (or an expression) is as follows:

*term* {*operator term*}

Syntax for conditions is described throughout this section.

The return value of a condition can be TRUE or FALSE. In case the output of a condition is NULL then its return value is FALSE.

For example,

```
Rule test
    inst1

    Message(...);
EndRule
```

In this case, a message is printed for each instance in the design. If there are 10 instances in the design, then there will be 10 messages. Consider the following rule:

```
Rule RuleX1
    hasNoProperty(inst1,"TEST")

     Message( ... );
EndRule
```

In the above case, if one out of ten instances has the TEST property, `hasNoProperty` will return NULL. Therefore, the condition will be FALSE and no message will be printed for this instance. But for the nine other instances, `hasNoProperty` will return TRUE, resulting in nine messages (one per instance).

To print a message for each instance with the TEST property, you can code the rule as follows:

```
Rule RuleX1
    isNull(hasNoproperty(inst1,"TEST"))
    Message(...);
EndRule
```

In this case, `hasNoProperty` returns NULL for the instance that had the property TEST, and a non-NULL value for the other nine instances. `isNull` returns TRUE only for the case in which `hasNoProperty` returns a NULL, resulting in only one message.

To get one message that lists all instances in the design that do not have the TEST property, write the rule as follows:

```
Rule test_all
    inst1 = hasNoProperty(inst(design1),"TEST")
    Message(...);
EndRule
```

This rule produces one message if there is at least one instance in the design that does not have the TEST property. The difference between this rule (`test_all`) and the `test` rule is the basic design of the rule.

In `test_all`, the rule is evaluated as follows:

```
for each design
check if there is an instance that does not have the TEST property
```

In `test`, instance was the basic object, so the rule is evaluated as follows:

```
for each instance in the design
check if it has the property TEST
```

The number of iterations in both cases is different. `test_all` went through only one iteration (as you can only test one design at a time), while in `test` the number of iterations is equal to the number of instances in the design (10).

Logical operators (AND, OR, XOR) operate on the return values of the conditions. For example, in the following rule

```
Rule test_and
    instTerm1 := output(instTerm(inst1)) AND
    unused(instTerm1)
    Message(..);
EndRule
```

the first condition is TRUE for an instance if it has an output terminal. The second condition is TRUE if the terminal is not connected. Since there is an AND between these conditions, the message is only printed for an instance that has at least one unused output terminal.

### *Operators*

The language has its operators in the form of operator predicates. However, it is sometimes convenient to express operations in terms of operators. All binary operators (such as *, /, ==, AND, OR, and so on), which operate on two operators, use the following syntax:

```
operand1 binary_operator operand2
```

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

| Operator | Description |
|---|---|
| not, isNull, abs, ** | not - negate |
| | isNull - isInputArgumentNull |
| | abs - absolute |
| | ** - exponent |
| *, /, mod, rem | multiply, divide, modulus, remainder |
| +, - | add, subtract |
| ==, /=, <. <=, >, >= | equal to, not equal to, less than, less than or equal to, greater than, greater than or equal to |
| := | assignment |
| and, or, xor | logical operators |

| Operator | Description |
|---|---|
| {item, item, ...} | one of a list of item |

The operator “{}” has a special meaning in the language. Used in conjunction with a “==” operator, it checks for occurrences of the item on the left hand side within the list of items within the curly brackets. For example:

```
term_type(terminal1) == {"Y-PORT", "Z-PORT", W-PORT"}
```

Here if *term_type* is one of Y-PORT, Z_PORT or W-PORT, then the above comparison returns TRUE. You can only include constant literals (23, 14.45, “avc”) within “{” and “}”.

Note that you can use parentheses to replace default precedence. For operators with equal precedence, evaluation takes place from left to right.

This is also true for all predicates with more than one argument. For example, to find the signals that do not have the SAME property and have the same name as an instance, use

```
sig1:= hasNoProperty(sig(design1), “SAME”) AND
matchName(inst(design1), name(sig1))
```

Consider a design that contains two signals that do not have the SAME property and that has five instances, where one of the two signals has the same name as an instance. Because the second argument of the predicate *matchName* is a list with two elements, there will be two entries per instance in the output list. Therefore, there will be ten entries in the output list; nine out of the ten entries will be NULL and one will have the instance whose name matched the signal name.

The `isNull` (condition) operator checks if the condition has returned all NULLs. For example:

```
isNull(hasProperty(inst(design1), “RES”))
```

returns TRUE if no instance in the design has the RES property.

### *Bound and Unbound Variables*

Variables represent values determined during rule execution. The variable can be either bound or unbound to a value. If a variable is bound, it has a substitution that is valid through the rest of the rule. A variable is unbound when what the variable stands for is not yet known. For example, in the following condition

```
instTerm2 := input(connected(output(instTerm1)))
```

the variable `instTerm1` is unbound to start with. If the design contains an output pin, then `instTerm1` will be bound to that pin. Similarly, `instTerm2` is not bound until an input pin

connected to an output pin (in this case, `instTerm2`) is found. If such a pin is found, then `instTerm2` becomes bound to that pin.

### *List Operations*

The `foreach` construct lets you evaluate a condition on individual elements of a list. For example, you can use it to search for a property on every element of a list, or you can use it to compare two lists.

The syntax of the `foreach` construct is

`foreach(`*`arg1`*`, `*`cond1`*`)`

This returns a list of items in *`arg1`* that satisfy the condition *`cond1`*.

The following example shows how to use `foreach` to check each element of a list (`sig1`, which contains all signals in `design1`) for a property (TEST), and returns a list (`sig2`) of the elements that have the TEST property attached.

```
sig1 := sig(design1) AND
sig2 := foreach(sig1, hasproperty(sig1, "TEST")
```

The following example shows how to create a list (`newlist`) that contains the elements of `list1` that are not in `list2`.

```
newlist := (foreach(list1, isNull(matchName(list1,list2))))
```

In this example, every element in `list1` is checked against the condition `isNull(matchName(list1, list2))`. Because `foreach` checks each element in `list1` individually, `matchName` also considers `list1` as one element. `matchName` checks the element from `list1` against each element of `list2`, and if a pair of elements does not match, `matchName` returns a NULL, which satisfies the above condition, and the element is added to `newlist`.

The following example determines whether two lists (`list1`, `list2`) are identical.

```
count(list1) == count(list2) AND
list3 := list1 AND
list4 := foreach (list1,
    nth(list2, index(list3, list1)) == list1) AND
count (list4)== count(list1)
```

This example can be described in the following steps.

1. Check if `list1` and `list2` have the same number of elements
2. Create a third list (`list4`) that contains those elements of `list1` that have a corresponding match in `list2` at the same position.

3. Check if `list4` has the same number of elements as the other lists. To reach this step, `list1` and `list2` must have the same number of elements, so `list4` only needs to be compared against `list1` or `list2`.

See "User-Created and Predicate-Created Lists" on page 291 for additional information on using user-defined and predicate-created lists.

### *Message Predicates*

Message predicates are used for reporting violations and specifying objects to be highlighted in the design.

The syntax of the predicate is as follows.

```
message_predicate ::= Message(message_class_name, hilite_object_list,
"short_message_Value"[, "long_message_Value"]*)
hilite_object_list := hilite_object_list object_name
object_name := an object variable defined in the current condition / short message/
long message section
```

The predefined message classes are `Info`, `Oversight`, `Error`, `Fatal`, and `Warning`.

The predicate requires a minimum of three arguments. The first is the severity of the message and the second is the highlighted list and the third is the short message value, which is enclosed within double quotes. If a long message is to be printed, it can be included after the short message. The long message is printed in the `cp.msg` file. A newline character is automatically printed after both the messages.

The values of variables or expressions can be printed by prefixing the variable names with a question mark (?). The Message predicate treats variables that are objects in a special way. It always prints the name of the object if the variable represents an object. Thus "?inst1" results in the name of the instance being printed.

Predicate calls cannot be used in a short or long message.

The following are examples of message predicates:

```
Message(Error, inst1 "Instance ?inst1 under error")
Message(Warning, pin1, "No drive on port","Input port  ?pin1 on:", "macro
?inst1 have no source")
```

You can print the? character in the message by using the escape character '\' before it, as in the following example:

```
Message(Hint, "What's in a name \?")
```

To print a backslash (\) or double quotes (”) through the message predicate, escape using backslashes \\ or \ respectively.

# Programming Examples

This section contains hints and examples to assist you with creating rules in the Rules Checker Rule Language. For additional examples, examine the source files for the rules supplied with the Rules Checker Toolkit. You can find these in `<install_dir>/tools/checkplus_exp/concept/rules_source`.

## Counting Elements

The following rule, `count_inst`, is an example of how to count elements at the design level.

```
Rule count_inst

    var1 := count(inst(design1)) AND
    str1 := "Gates"

    Message(COUNT_INST_SEVERITY, design1,
             STD_SHORT_ERR_COUNT_INST,
             STD_ERR_COUNT_INST);
EndRule
```

In the above rule, `inst(design1)` returns a list of all instances in the design (`design1`). `count`() then counts the elements in this list, and this value is assigned to `var1`.

`str1` is assigned a value of “Gates,” which is used in the message file. (All of the counting rules supplied with Rules Checker use the same message predicates, so “Gates” helps to clarify the messages produced by this rule.)

## Finding Unconnected Elements

The following rule, `unconnected_instance`, is an example of how to find unconnected elements in your design.

```
Rule unconnected_instance

    val1 := count(unused(instTerm(inst1))) AND
    val2 := count(instTerm(inst1)) AND
    val2 /= 0 AND
    val1 == val2

    Message(UNCONNECTED_INSTANCE_SEVERITY, inst1,
             STD_SHORT_ERR_UNCONNECTED_INSTANCE,
             STD_ERR_UNCONNECTED_INSTANCE);
EndRule
```

In the above rule, `val1` is assigned the number of unconnected instance terminals on `inst1`. `val2` is assigned the number of instance terminals on `inst1`. Then the values of `val1` and `val2` are compared. If `val1` and `val2` are equal and `val2` is not equal to zero, this means all instance terminals are unused, and the instance is unconnected in the design.

Note that `inst1` is the primary object in the design, so the rule checks every instance in the design.

## Using Strings and Lists

The following rule, `power_group1`, provides an example of how to create and perform basic operations on strings and lists. This rule checks that reassigned power pins (specified by the POWER_GROUP property) on instances are reassigned to global nets.

```
Rule power_group1
        str :=GetAllSubStrings(SkipWhiteSpaces
            (getHierProperty(inst1,"POWER_GROUP")),
            "=",{";",""}) AND
        str2 :=GetAllSubStrings(SkipWhiteSpaces
            (getHierProperty(inst1,"POWER_GROUP")),
            {";",""},{";",""}) AND
        str3 := foreach(str, isNull(isglobal
            (findsig(str)))) AND
        (
            count(str3) > 0 OR
            sig1 := isnotglobal(findsig(str))
        )
        Message(POWER_GROUP1_SEVERITY, inst1,
                    STD_SHORT_POWER_GROUP1,
                    STD_ERR_POWER_GROUP1);
EndRule power_group1
```

In the above rule, `str` is a list of all new power pin assignments. This list is created by using `getHierProperty(inst1)` to create a list that contains the values of all POWER_GROUP properties attached to every instance in the design. `SkipWhiteSpaces` removes any blank spaces in the list, then `getAllSubStrings` extracts all strings between “=” and “;”, and strings between “=” and the end of a line, and creates a list (`str`) from each of these strings.

For example, the following property on an instance:

```
POWER_GROUP = VCC=NEW_VCC;GND=NEW_GND;
```

adds the following elements to `str`:

```
NEW_VCC NEW_GND
```

`str2` is created similarly, except that it extracts substrings preceded by “;” or the beginning of a line, instead of substrings preceded by =, as in `str`. For example, the above POWER_GROUP property adds the following elements to `str2`:

```
VCC=NEW_VCC GND=NEW_GND
```

`str3` uses the `foreach` construct to create a list of the non-global and non-existent signals in `str`. `findsig(str)` returns NULL on signals in `str` that do not exist, and `isglobal(findsig(str))` returns NULL on each non-global signal in `str`. `isNull` is used as a condition to add these signals to the list.

Next, the rule checks if `str3` has any elements. If so, then information on each of the elements in this list is sent to the message predicate. The rule also checks if any of the elements in `str` are non-global, and sends this information to the message predicate.

### User-Created and Predicate-Created Lists

The Rules Checker evaluation engine treats user-created and predicate-created lists differently. When evaluating predicate-created lists, the engine evaluates the list as separate elements. However, user-created lists are evaluated as single elements.

For example, consider the following three lists with elements VCC and GND, where `list1` and `list2` are created by a predicate and `list3` is created by the following statement in a header file:

```
PARAM list3 "{VCC, GND}"
```

Because `list1` and `list2` are predicate-created, the `matchName(list1, list2)` checks the following combinations:

```
(VCC, VCC), (VCC, GND), (GND, VCC), (GND, GND)
```

which returns VCC, NULL, NULL, GND. Similarly, `isNull(matchName(list1, list2))` returns VCC, GND.

However, `matchName(list1, list3)` checks the following combinations:

```
(VCC, {VCC,GND}), (GND, {VCC, GND})
```

which returns VCC, GND. In this case, the `isNull(matchName(list1, list3))` returns NULL, as both `matchName` operations return non-NULL values.

# Built-In Predicates

This section describes the built-in predicates in Rules Checker. These predicates do not appear in any of the environment files, and can be used with any of the basic objects. The predicates are listed in alphabetical order.

### append

`append(`*`list1`*`, list2)`

Appends the contents of *`list2`* to the end of *`list1`*. You can use this on a list only once. For repeated operation on *`list1`* use it within the foreach statement.

### areDatesEqual

`areDatesEqual(`*`str1,str2`*`)`

Returns *`str1`* if *`str1`* in date format is more current than *`str2`* in date format. Returns NULL if false, or if *`str1`* or *`str2`* are not in valid date format.

The following strings are acceptable date formats:

- Oct 10 10:10:10 2020
- Mon Oct 10 10:10:10 2010
- Mon Oct 10 10:10:10 PST8PDT 2010
- Oct 10 10:10:10 PST8PDT 2010

### areFilesSame

`areFilesSame(`*`str1,str2`*`)`

`areFilesSame(`*`str1,str2,opt`* `)`

Returns `str1` if contents of file `str1` and file `str2` are equal. `opt` can use the values `i` and/or `w`, where `i` makes the search case-insensitive and `w` strips white spaces before comparing.

### car

`car(`*`list`*`)`

Returns the first element in the list. If *`list`* is nil, then nil is returned. *`list`* can have both object and non-object types.

### cdr

`cdr(`*`list`*`)`

Returns the list without the first element. Returns nil if the *list* contains only one element or no elements. *list* can have both object and non-object types.

### ceil

`ceil(val1,val2)`

Returns the ceil of *val1* divided by *val2*.

### concat

`concat(list1,list2, ...)`

Concatenates a copy of all the lists passed and returns the result. Multiplicity of elements in the resulting list is not checked.

### count

`count(condition)`

Returns the number of elements in *condition*.

### fexists

`fexists(file)`

Returns True if *file* exists.

### fSearch

`fSearch(file,str)`

`fSearch(file,str,opt)`

Returns the number of times the string *str* is found in the file *file*. *opt* can use the values `i` and/or `w`, where `i` makes the search case-insensitive and `w` strips white spaces before comparing.

### fsubmatch

`fsubmatch(value)`

Returns a list of file names if *value* matches part of the file name. The substring preceding the last "/" is used as the search directory. Otherwise it returns NULL.

### getAllSubstrings

`getAllSubstring(`*`string1, startToken, endToken`*`)`

Returns all substrings of `string1` that occur between `startToken` and `endToken`.

**Note:** *`StartToken`* and *`endToken`* are not part of the returned string.

For example, `getAllSubstrings("VCC=VEE;VDD=VSS;", "=",";")` returns a list containing VEE and VSS.

### getChars

`getchars(`*`string`*`)`

Returns all characters of a *`string`*. For example,

```
getChars (NEEDS)
```

returns `N`, `E`, `E`, `D`, `S`.

### getFileSubStrings

`getFileSubStrings(`*`file,starttoken,endtoken`*`)`

Returns a list of strings from *`file`*, where *`starttoken`* and *`endtoken`* delimit the strings to be returned.

### getAccess

`getAccess(`*`filename`*`)`

Returns date of last access of *`filename`* in the following format:

```
Mon DD HH:MM:SS YYYY
```

### getChange

`getChange(`*`filename`*`)`

Returns date of last change to *`filename`* in the following format:

```
Mon DD HH:MM:SS YYYY
```

### getModify

`getModify(`*`filename`*`)`

Returns date of last modification to *filename* in the following format:

```
Mon DD HH:MM:SS YYYY
```

### getSubstring

```
getSubstring(string1, startToken, endToken)
```

Returns the first substring of *string1* that occurs between *startToken* and *endToken*.

**Note:** S*tartToken* and *endToken* are not part of the returned string.

For example, getSubstring("VCC=VEE;VDD=VSS;", "=",";") returns VEE.

### index

```
index (list,data)
```

Returns the index of the data element in *list*. If *data* is not found in the list, it returns nil. *list* can have both object and non-types. *data* must be of the same type as that of *list* and it must be a single element.

### inRange

```
inRange(value, range)
```

Returns true if *value* is within *range*. Specify *range* as `minValue:maxValue`. For example, use “1:10” or “1...10” to specify the range between 1 and 10, inclusive.

### intersection

```
intersection(list1, list2)
```

Returns the intersection of the two lists. If the intersection is nil then nil is returned. *list1* and *list2* can have both object and non-object types. Both arguments must be of the same type.

### isDirec

```
isDirec(string)
```

Returns TRUE if *string* is a directory name.

### isExecutable

`isExecutable(`*`string`*`)`

Returns TRUE if *`string`* is a file and the user has execute permissions. Otherwise, returns FALSE.

### isLater

`isLaterDate(`*`str1`*`,`*`str2`*`)`

Returns *`str1`* if *`str1`* in date format is equal to *`str2`* in date format. Returns NULL if no match, or if *`str1`* or *`str2`* are not in valid date format.

The following strings are acceptable date formats:

- Oct 10 10:10:10 2020
- Mon Oct 10 10:10:10 2010
- Mon Oct 10 10:10:10 PST8PDT 2010
- Oct 10 10:10:10 PST8PDT 2010

### isReadable

`isReadable(`*`string`*`)`

Returns TRUE if *`string`* is a file and the user has read permissions. Otherwise, returns FALSE.

### isWriteable

`isWriteable(`*`string`*`)`

Returns TRUE if *`string`* is a file and the user has write permissions. Otherwise, returns FALSE.

### last

`last(`*`list`*`)`

Returns the last element of the list. If *`list`* is nil, then it returns nil. *`list`* can have both object and non-object types.

### lsDir

`lsDir(`*`dirName`*`)`

`lsDir(`*`dirName`*`,`*`extn`*`)`

Returns a list of files and directories in the *`dirName`* directory. If *`extn`* is specified, returns only file names with extension *`extn`*. Otherwise returns NULL if *`dirName`* is not a valid directory name.

### matchExp

`matchExp(`*`string,regex`*`)`

Checks if *`string`* fits into the regular expression *`regex`* (for example,

for matching all signals whose names contain string TP use,

matchExp(name(sig1), “[tT] [pP]”))).

**Note:** Allegro Design Entry HDL Rules Checker supports the following functions for simple regular expressions: re_comp() and re_exec().

### matchName

`matchName(`*`val1`*`,`*`val2`*`)`

Returns *`val1`* if *`val1`* and *`val2`* match.

### matchNoName

matchNoName(*`val1`*,*`val2`*)

Returns *`val1`* if *`val1`* and *`val2`* do not match.

### max

`max(`*`condition`*`)`

Returns the maximum value of the elements in *`condition`*. Arguments must be of integer or float type. If you are using a predicate that returns a string value (such as `getProperty`) as an argument to `max`, you must use the `toFloat` or `toInt` predicate to convert the string (for example, `max(toFloat(getProperty(sig1,SIGNAL_PROPERTY_NAME)))`).

### median

*median (list)*

Finds the mathematical median of values in the list. All the values in the list must be integers or floating points.

### min

min(*condition*)

Returns the minimum value of the elements in *condition*. Arguments must be of integer or float type. If you are using a predicate that returns a string value (such as `getProperty`) as an argument to `min`, you must use the `toFloat` or `toInt` predicate to convert the string (for example, `min(toFloat(getProperty(sig1,SIGNAL_PROPERTY_NAME)))`).

### mode

*mode (list)*

Finds the mathematical mode of values in the list. All the values in the list must be integers or floating points,

### nth

nth(*list*,*n*)

Returns the *n*th element, where *n* is an integer. Returns nil if *list* is shorter than *n* elements. If *n* is more than the length of the list then an evaluation error is reported. Specify `n=1` for the first element. The list can have both object and non-object types.

### outStrToFile

*outStrToFile( <file name> , <mode>, <string to output)*

Writes a string onto the file specified by the file name. The file is opened in the mode that you specify.

*<mode>:*

- "w"/"w+" - write: Creates an empty file for output operations. If a file with the same name already exists, its contents are discarded and the file is treated as a new empty file.

- "a"/"a+" - append: Opens file for output at the end of a file. Output operations always write data at the end of the file, expanding it. Repositioning operations (fseek, fsetpos, rewind) are ignored. The file is created if it does not exist.

The plus sign (+ ) in a mode opens the file for both read and write operations. However, the `outStrToFile` is meant only for output. Therefore, you can omit the + sign.

### overlap

`overlap(`*`string`*`,`*`string`*`)`

Returns TRUE if the bounding boxes of the strings (specified in the form "((x,y),(x,y))") overlap. Returns FALSE otherwise.

### remove

`remove(`*`list1`*`,`*`list2`*`)`

Returns a list containing the elements of *`list1`* that are not in *`list2`*. If there are no such elements, then it returns nil. *`list1`* and *`list2`* can have both object and non-object types. Both arguments must be of the same type.

### remove_i

`remove_i(`*`list`*`,`*`i`*`)`

Returns a list containing the elements of *`list`* except for the *`i`*th element, where *`i`* is an integer. If the remaining list is nil, then it returns nil. If *`i`* is more than the length of the list then an evaluation error is reported. Specify `i=1` for the first element. *`list`* can have both object and non-object types.

### sizeof

`sizeof(`*`file`*`)`

Returns the size (in bytes) of *`file`*. Issues a warning and returns 0 if *`file`* does not exist.

### skipWhiteSpaces

`skipWhiteSpaces(`*`string`*`)`

Strips spaces, tabs and linefeeds from *`string`*, unless they are enclosed within single quotes (for example, '\t' is not stripped), and returns the resulting string.

### sort

`sort(list)`

Sorts the list in ascending order according to the type: integer, float, boolean, or string. If the list consists of boolean elements (TRUE, FALSE, and so on) the predicate places the FALSE before TRUE.

### strlen

`strlen(string)`

Returns the length of *string*.

### sum

`sum(condition)`

Returns the sum of the values of the elements in *condition*.

### toFloat

`toFloat(value)`

Converts *value* to a float and returns the value.

### toInt

`toInt(value)`

Converts *value* to an integer and returns the value.

### toString

`toString (value)`

Converts *value* to string. *value* must be an integer, float, boolean, or string.

### union

`union(list1,list2)`

Returns the union of the two lists. If both lists are nil, then it returns nil. Any duplicates in the resulting list are removed. *list1* and *list2* can have both object an non-object types. *list1* and *list2* must be of the same type.

### unique

`unique(`*list*`)`

Returns a *list* containing the elements of *list*, with no duplicates. *list* can have both object and non-object types.

# Body Environment Predicates

This section describes the predicates available for creating rules in the body environment.

The body environment uses the following primary objects:

- design is the complete body file.
- note is a note in the body file.
- prop is a property in the body file.
- bodypin is a physical pin in the body file.
- seg is a line segment in the body file.
- arc is an arc in the body file.
- PhysPackType is a primitive in the `chips_prt` file.

## Object Design Predicates

This section describes the object design predicates available in the body environment. These predicates operate on the entire design. The predicates are listed in alphabetical order.

### arc

`arc(`*`design`*`)`

Returns a list of all arcs in the design.

### bbox

`bbox(`*`design`*`)`

Returns the bounding box of the body file.

### bodypin

`bodypin(`*`design`*`)`

Returns a list of all bodypins in the body file.

### file

`file(`*`design`*`)`

Returns `<`*`cell`*`>/<`*`view`*`>/symbol.css`.

### leftX

`leftX(`*`design`*`)`

Returns the lower-left x value of the bounding box of the body file.

### library

`library(`*`design`*`)`

Returns the name of the library in which the body file resides.

### logicPinNames

`logicPinNames(`*`design,version,substring`*`)`

Returns a list of all pin names on all pages of the logic body file with version *`version`* and a substring *`substring`* in the value.

### lowerY

`lowerY(`*`design`*`)`

Returns the lower-left y value of the bounding box of the body file.

### name

`name(`*`design`*`)`

Returns the name of the body file.

### note

`note(`*`design`*`)`

Returns a list of all notes in the body file.

### path

`path(`*`design`*`)`

Returns the library of the design directory that contains the body file.

### physPackType

`physPackType(`*`design`*`)`

Returns a list of all primitive names from the `chips.prt` file corresponding to the body file.

### prop

`prop(`*`design`*`)`

Returns a list of all properties in the body file.

### rightX

`rightX(`*`design`*`)`

Returns the upper-right x value of the bounding box of the body file.

### toolstartFail

`toolstartFail(`*`design`*`,`*`call_string`*`)`

Returns the design on which tool invoked in *`call_string`* fails, that is, the exit status is 1.

### toolstartIgnore

`toolstartignore(`*`design`*`,`*`call_string`*`)`

Returns the design ignoring return-status of tool invoked in *`call_string`*.

### toolstartPass

`toolstartPass(`*`design,call_string`*`)`

Returns the design on which tool invoked in *`call_string`* passes, that is the exit status is 0.

### seg

`seg(`*`design`*`)`

Returns a list of all segments in the body file.

### upperY

`upperY(`*`design`*`)`

Returns the upper right y-value of the bounding box of the body file.

### version

`version(`*`design`*`)`

Returns the version number of the body file.

## Object Note Predicates

This section describes the object note predicates available in the body environment. These predicates operate on notes in the body file. The predicates are listed in alphabetical order.

### bbox

`bbox(note)`

Returns the bounding box of the note.

### color

`color(note)`

Returns the color of the note.

### designName

`designName(note)`

Returns the name of the body drawing.

### distance

`distance(note, bbox)`

Finds the distance between the note and the bounding box specified as the second argument.

### filename

`filename(note)`

Returns `<cell>/<view>/symbol.css`.

### just

`just(note)`

Returns the justification of the note.

### leftX

`leftX(note)`

Returns the lower-left x value of the bounding box of the note.

### lowerY

`lowerY(note)`

Returns the lower-left y value of the bounding box of the note.

### matchName

`matchName(note, string)`

Returns the note if its name (that is, text) in the schematic matches the second argument.

### matchNoName

`matchNoName(note, string)`

Returns the note if its name (that is, text) in the schematic does not match the second argument.

### name

`name(note)`

Returns the text of the note.

### path

`path(note)`

Returns the library of the design directory that contains the note.

### orient

`orient(note)`

Returns the orientation of the note.

### overlap

`overlap(`*`note, value`*`)`

Returns the note if it overlaps with the bounding box specified as the second argument, which should be in the format ((x,y),(x,y)).

### rightX

`rightX(`*`note`*`)`

Returns the upper-right x value of the bounding box of the note.

### size

`size(`*`note`*`)`

Returns the font size of the text of the note.

### upperY

`upperY(`*`note`*`)`

Returns the upper right y-value of the bounding box of the note.

## Object Property Predicates

This section describes the object property predicates available in the body environment. These predicates operate on properties in the body file. The predicates are listed in alphabetical order.

### bbox

`bbox(`*`prop`*`)`

Returns the bounding box of the visible portion of the property.

### bodypin

`bodypin(`*`prop`*`)`

Returns the bodypin associated with the property.

### color

`color(`*`prop`*`)`

Returns the color of the property.

### designName

`designName(`*`prop`*`)`

Returns the name of the body drawing.

### distance

`distance(`*`prop`*`, `*`bbox`*`)`

Finds the distance between the property and the bounding box specified as the second argument in the format *`((x1,y1),(x2,y2))`*.

### filename

`filename(`*`prop`*`)`

Returns `<`*`cell`*`>/<`*`view`*`>/symbol.css`.

### just

`just(`*`prop`*`)`

Returns the justification of the property.

### leftX

`leftX(`*`prop`*`)`

Returns the lower-left x value of the bounding box of the property.

### lowerY

`lowerY(`*`prop`*`)`

Returns the lower-left y-value of the bounding box of the property.

### matchName

`matchName(`*`prop`*`,`*`string`*`)`

Returns the property if its name in the schematic matches the second argument.

### matchNoName

`matchNoName(`*`prop`*`,`*`string`*`)`

Returns the property if its name in the schematic does not match the second argument.

### name

`name(`*`prop`*`)`

Returns the name of the property.

### nameInvis

`nameInvis(`*`prop`*`)`

Returns the property if its name is invisible in the schematic.

### nameOrValueVis

`nameOrValueVis(`*`prop`*`)`

Returns the property if its name or value is visible in the schematic.

### nameVis

`nameVis(`*`prop`*`)`

Returns the property if its name is visible in the schematic.

### orient

`orient(`*`prop`*`)`

Returns the orientation of the property.

### overlap

`overlap(`*`prop,bbox`*`)`

Returns the property if it overlaps with the bounding box specified as the second argument, which needs to be in the format ((x,y),(x,y)).

### path

`path(`*`prop`*`)`

Returns the library of the design that contains the property.

### rightX

`rightX(`*`prop`*`)`

Returns the upper-right x value of the bounding box of the property.

### segs

`segs(`*`prop`*`)`

Returns the list of segments (which constitute a wire) to which the property is attached.

### size

`size(`*prop*`)`

Returns the font size of the text of the property.

### upperY

`upperY(`*prop*`)`

Returns the upper right y-value of the bounding box of the property.

### value

`value(`*prop*`)`

Returns the value of the property.

### valueInvis

`valueInvis(`*prop*`)`

Returns the property if its value is invisible in the schematic.

### valueVis

`valueVis(`*prop*`)`

Returns the property if its value is visible in the schematic.

## Object Bodypin Predicates

This section describes the object bodypin predicates available in the body environment. These predicates operate on body pins in the body file. The predicates are listed in alphabetical order.

### designName

`designName(`*`bodypin`*`)`

Returns the name of the body drawing.

### dir

`dir(`*`bodypin`*`)`

Returns the direction (EAST, WEST, NORTH, SOUTH) of the bodypin.

### fileName

`fileName(`*`bodypin`*`)`

Returns `<`*`cell`*`>/<`*`view`*`>/symbol.css`.

### getProperty

`getProperty(`*`bodypin,string`*`)`

Gets the value of the property specified as the second argument.

### getPropertyFloat

`getPropertyFloat(`*`bodypin,string`*`)`

Gets the floating-point value of the property specified as the second argument.

### getPropertyInt

`getPropertyInt(`*`bodypin,string`*`)`

Gets the integer value of the property specified as the second argument.

### hasProperty

`hasProperty(`*`bodypin,string`*`)`

Returns the bodypin if the property specified as the second argument exists on it.

### inout

`inout(`*`bodypin`*`)`

Case pin_dir_check option is ON (default):

Returns the bodypin if BIDIRECTIONAL property is there on the bodypin.

Case pin_dir_check option is OFF:

Returns the bodypin if BIDIRECTIONAL or INPUT_LOAD and OUTPUT_LOAD property is present.

### inOrInout

`inOrInout(`*`bodypin`*`)`

Case pin_dir_check option is ON (default):

Returns the bodypin if INPUT_LOAD or BIDIRECTIONAL property is there on the bodypin.

Case pin_dir_check option is OFF:

Returns the bodypin if none of the properties (BIDIRECTIONAL, INPUT_LOAD,OUTPUT_LOAD) is present or INPUT_LOAD or BIDIRECTIONAL or both INPUT_LOAD and OUTPUT_LOAD property is present.

### input

`input(`*`bodypin`*`)`

Case pin_dir_check option is ON (default):

Returns the bodypin if INPUT_LOAD property is there on the bodypin.

Case pin_dir_check option is OFF:

Returns the bodypin if none of the properties (BIDIRECTIONAL, INPUT_LOAD,OUTPUT_LOAD) is present or if INPUT_LOAD property is only present.

### matchName

`matchName(`*`bodypin,string`*`)`

Returns the bodypin if its name in the schematic matches the second argument.

### matchNoName

`matchNoName(`*`bodypin, string`*`)`

Returns the bodypin if its name in the schematic does not match the second argument.

### matchProperty

`matchProperty(`*`bodypin,string,string`*`)`

Returns the bodypin if the property matched as the second argument exists on it, and its value matches the third argument.

### name

`name(`*`bodypin`*`)`

Returns the name of the bodypin.

### outOrInout

`outOrInout(`*`bodypin`*`)`

Case pin_dir_check option is ON (default):

Returns the bodypin if OUTPUT_LOAD or BIDIRECTIONAL property is there on the bodypin.

Case pin_dir_check option is OFF:

Returns the bodypin if OUTPUT_LOAD or BIDIRECTIONAL or both INPUT_LOAD and OUTPUT_LOAD property is present.

### output

`output(`*`bodypin`*`)`

Case pin_dir_check option is ON (default):

Returns the bodypin if OUTPUT_LOAD property is there on the bodypin.

Case pin_dir_check option is OFF:

Returns the bodypin if OUTPUT_LOAD property is there on the bodypin.

### PassThru

`PassThru(`*`bodypin`*`)`

Returns the bodypin if it is a pass-thru pin.

### path

`path(`*`bodypin`*`)`

Returns the path of the design library that contains the bodypin.

### prop

`prop(`*`bodypin`*`)`

Returns a list of the properties attached to the body pin.

### xcoord

`xcoord(`*`bodypin`*`)`

Returns the x-coordinate of the bodypin.

### ycoord

`ycoord(`*`bodypin`*`)`

Returns the y-coordinate of the bodypin.

## Object Segment Predicates

This section describes the predicates available for the object segment in the body environment. These predicates operate on segments in the body file. These predicates are listed in alphabetical order.

### bbox

`bbox(`*`seg`*`)`

Returns the bounding box of the segment.

### color

`color(`*`seg`*`)`

Returns the color of the segment.

### designName

`designName(`*`seg`*`)`

Returns the name of the body drawing.

### distance

`distance(`*`seg, value`*`)`

Finds the distance between the segment and the box specified as the second argument.

### fileName

`fileName(`*`seg`*`)`

Returns *<cell>/<view>*`/symbol.css`.

### getProperty

`getProperty(`*`seg, string`*`)`

Gets the value of the property specified as the second argument.

### getPropertyFloat

`getPropertyFloat(`*`seg, string`*`)`

Gets the floating-point value of the property specified as the second argument.

### getPropertyInt

`getProperty(`*`seg, string`*`)`

Gets the integer value of the property specified as the second argument.

### hasProperty

`hasProperty(`*`seg,string`*`)`

Returns the segment if the property specified as the second argument exists on it.

### horizontal

`horizontal(`*`seg`*`)`

Returns the segment if it is horizontal.

### leftX

`leftX(`*`seg`*`)`

Returns the lower left x-coordinates of the bounding box of the segment.

### lowerY

`lowerY(`*`seg`*`)`

Returns the lower left y-coordinate of the bounding box of the segment.

### matchName

`matchName(`*`seg,string`*`)`

Returns the segment if its name in the schematic matches the second argument.

### matchNoName

`matchNoName(`*`seg,string`*`)`

Returns the segment if its name in the schematic does not match the second argument.

### matchProperty

`matchProperty(`*`seg,string,string`*`)`

Returns the segment in case the property specified as the second argument exists on it and matches the third argument.

### name

`name(`*`seg`*`)`

Returns the coordinates of the segment.

### overlap

`overlap(`*`seg,string`*`)`

Returns the segment if it overlaps with the bounding box specified as the second argument, which needs to be in the form ((x,y),(x,y)).

### path

`path(`*`seg`*`)`

Returns the path of the design library that contains the segment.

### rightX

`rightX(`*`seg`*`)`

Returns the upper right x-coordinate of the bounding box of the segment.

### upperY

`upperY(`*`seg`*`)`

Returns the upper right y-coordinate of the bounding box of the segment.

### vertical

`vertical(`*seg*`)`

Returns the segment if it is vertical.

### wireprop

`wireprop(`*seg*`)`

Returns the list of properties on the wire to which the segment belongs.

## Object Arc Predicates

This section describes the predicates for object arc available in the body environment. These predicates operate on arcs in the body file. The predicates are listed in alphabetical order.

### bbox

`bbox(arc)`

Returns the bounding box of the arc.

### color

`color(arc)`

Returns the color of the arc.

### designName

`designName(arc)`

Returns the name of the body drawing.

### distance

`distance(arc,value)`

Finds the distance between the arc and the bounding box specified as the second argument in the format `((x1,y1), (x2,y2))`.

### fileName

`fileName(arc)`

Returns `<cell>/<view>/symbol.css`.

### leftX

`leftX(arc)`

Returns the lower left x-coordinate of the bounding box of the arc.

### lowerY

`lowerY(`*`arc`*`)`

Returns the lower left y-coordinate of the bounding box of the arc.

### name

`name(`*`arc`*`)`

Returns the name of the arc.

### overlap

`overlap(`*`arc,value`*`)`

Returns the segment if it overlaps the bounding box specified as the second argument, which needs to be in the format ((x,y),(x,y)).

### radius

`radius(`*`arc`*`)`

Returns the radius of the arc.

### rightX

`rightX(`*`arc`*`)`

Returns the upper right x-coordinate of the bounding box of the arc.

### upperY

`upperY(`*`arc`*`)`

Returns the upper right y-coordinate of the bounding box of the arc.

### xcenter

`xcenter(`*`arc`*`)`

Returns the x-coordinate of the center of the arc.

### ycenter

`ycenter(`*`arc`*`)`

Returns the y-coordinate of the center of the arc.

## Object PhysPackType Predicates

This section describes the predicates available for object PhysPackType in the body environment. These predicates operate on primitives listed in the `chips_prt` file corresponding to the body file.The predicates are listed in alphabetical order.

### designName

`designName(`*`physPackType`*`)`

Returns the name of the body drawing.

### fileName

`fileName(`*`physPackType`*`)`

Returns *`<cell>/<view>`*`/symbol.css`.

### getAllPropNames

`getAllPropNames(`*`physPackType`*`)`

Returns a list of the property names in the `chips_prt` file.

### getPinsOfTerm

`getPinsOfTerm(`*`physPackType, string`*`)`

Returns the pin numbers of the term of the primitive specified as the second argument.

### getProperty

`getProperty(`*`physPackType, string`*`)`

Returns the value of the property (specified as the second argument) if it exists on *`physPackType`*.

### getPropertyFloat

`getProperty(`*`seg, string`*`)`

Gets the floating-point value of the property specified as the second argument.

### getPropertyInt

`getProperty(`*`seg, string`*`)`

Gets the integer value of the property specified as the second argument.

### getSectOfPin

`getSectOfPin(`*`physPackType,string`*`)`

Returns the section number in which the pin number specified as the second argument occurs on the primitive specified in the first argument.

### getSectPins

`getSectPins(`*`physPackType,string`*`)`

Returns the list of pin names that occur in the section specified as the second argument.

### getTermAllPropNames

`getTermAllPropNames(`*`physPackType, string`*`)`

Returns a list of all properties that exists on the term of the primitive specified as the second argument.

### getTermOfPin

`getTermOfPin(`*`physPackType,string`*`)`

Returns the name of the term for the pin number specified in the second argument for the primitive specified as the first argument.

### getTermProperty

`getTermProperty(`*`physPackType, string, string`*`)`

Returns the value of the property specified as the third argument if it exists on the term specified as the second argument.

### hasTermProperty

`hasTermProperty(`*`physPackType,string,string`*`)`

Returns *physPackType* if the property specified as the third argument on the term specified as the second argument.

### matchName

`matchName(`*physPackType, value*`)`

Returns the primitive if its name in the schematic matches the second argument.

### matchNoName

`matchNoName(`*physPackType, value*`)`

Returns the primitive if its name in the schematic does not match the second argument.

### matchProperty

`matchProperty(`*physPackType, value, value*`)`

Returns the primitive if the property specified as the second argument exists on it and has a value that matches the third argument.

### name

`name(`*physPackType*`)`

Returns the name of the primitive from the `chips_prt` file.

### numSection

`numSection(`*physPackType*`)`

Returns the number of sections in the primitive.

### path

`path(`*physPackType*`)`

Returns the path of the design library that contains the primitive.

### term

`term(`*physPackType*`)`

Returns a list of term names.

## Miscellaneous Predicates

The following predicates are available in the body environment.

### mapToPhysPackName

`mapToPhysPackName(`*`value`*`)`

Returns the values corresponding to *`value`* in the `chips_prt` file.

### overlap

`overlap(`*`value, value`*`)`

The first and second arguments are strings in the format *`(x1,y1),(x2,y2)`*. This predicate returns true if the bounding boxes specified in the two arguments overlap.

# Graphical Environment Predicates

This section describes the predicates available for creating rules in the graphical environment.

The graphical environment uses the following primary objects:

- *design* is the Schematic view used as the base object to ensure that each selected rule is executed once per logic drawing
- *seg* is a segment (of a wire) in the drawing.
- *note* is a note in the drawing.
- *inst* is an instance in the drawing.
- *prop* is a property in the drawing.
- *body* is a unique body type used in the drawing. This is a subset of inst objects in the drawing.
- *pin* is a pin attached to an instance in the drawing.
- *bodypin* is pin attached to a body (not an instance) used in the drawing.
- *wire* is a wire (composed of segments) in the drawing.
- *PhysPackType* is a physical component in the drawing.

## Object Design Predicates

This section describes the object design predicates available in the graphical environment. These predicates operate on the design's entire logic drawing. The predicates are listed in alphabetical order.

### body

`body(`*`design`*`)`

Returns a list of all bodies whose instances occur in the design.

### fileName

`file(`*`design`*`)`

Returns *<cell>/<view>*`page` *<pagenum>*`.csb`.

### inst

`inst(`*`design`*`)`

Returns a list of all instances in the design.

### library

`library(`*`design`*`)`

Returns the name of the library or work file in which the design resides.

### name

`name(`*`design`*`)`

Returns the name of the design.

### note

`note(`*`design`*`)`

Returns a list of all notes in the design.

### path

`path(`*`design`*`)`

Returns the path of the design.

### prop

`prop(`*`design`*`)`

Returns a list of all properties in the design. This list does not contain properties found on bodies of instances in the design.

### toolstartFail

`toolstartFail(`*`design,call_string`*`)`

Returns the design on which tool invoked in *`call_string`* fails, that is, the exit status is 1.

### toolstartIgnore

`toolstartignore(`*`design,call_string`*`)`

Returns the design ignoring return-status of tool invoked in *`call_string`*.

### toolstartPass

`toolstartPass(`*`design,call_string`*`)`

Returns the design on which tool invoked in *`call_string`* passes, that is, the exit status is 0.

### seg

`seg(`*`design`*`)`

Returns a list of all segments in the design.

### wire

`wire(`*`design`*`)`

Returns a list of all wires (composed of all the segments) in the design.

## Object Segment Predicates

This section describes the object segment predicates available in the graphical environment. These predicates operate on segments in the logic drawing. The predicates are listed in alphabetical order.

### baseName

`baseName(`*`seg`*`)`

Returns the name of the signal, skipping the macros at the end.

### bbox

`bbox(`*`seg`*`)`

Returns the bounding box of the segment.

### color

`color(`*`seg`*`)`

Returns the color of the segment.

### designName

`designName(`*`seg`*`)`

Returns the name of the logic drawing that contains the segment.

### distance

`distance(`*`seg, value`*`)`

Finds the distance between the segment and the box specified as the second argument.

### fileName

`fileName(`*`seg`*`)`

Returns `<`*`cell`*`>/<`*`view`*`>page <`*`pagenum`*`>.csb`.

### getProperty

`getProperty(`*`seg, string`*`)`

Gets the value of the property specified as the second argument.

### getPropertyFloat

`getPropertyFloat(`*`seg, string`*`)`

Gets the floating-point value of the property specified as the second argument.

### getPropertyInt

`getPropertyInt(`*`seg, string`*`)`

Gets the integer value of the property specified as the second argument.

### hasProperty

`hasProperty(`*`seg,string`*`)`

Returns the segment if the property specified as the second argument exists on it.

### horizontal

`horizontal(`*`seg`*`)`

Returns the segment if it is horizontal.

### leftX

`leftX(`*`seg`*`)`

Returns the lower left x-coordinates of the bounding box of the segment.

### lowerY

`lowerY(`*`seg`*`)`

Returns the lower left y-coordinate of the bounding box of the segment.

### matchName

`matchName(seg,string)`

Returns the segment if its name in the schematic matches the second argument.

### matchNoName

`matchNoName(seg,string)`

Returns the segment if its name does not match the second argument.

### matchNoType

`matchNoType(seg,string)`

Returns the segment in case its name does not contain the second argument.

### matchProperty

`matchProperty(seg,string,string)`

Returns the segment in case the property specified as the second argument exists on it and matches the third argument.

### matchType

`matchType(seg,string)`

Returns the segment if its name contains the second argument.

### name

`name(seg)`

Returns the name of the segment.

### overlap

`overlap(seg,string)`

Returns the segment if it overlaps with the bounding box specified as the second argument, which needs to be in the form ((x,y),(x,y)).

### path

`path(seg)`

Returns the path of the design library that contains the segment.

### rightX

`rightX(seg)`

Returns the upper right x-coordinate of the bounding box of the segment.

### scalar

`scalar(seg)`

Returns the segment if its signal is a scalar.

### sigma

`sigma(list)`

Finds the standard deviation of values in the list. All the values in the list must be integers or floating points.

### signalWidth

`signalWidth(seg)`

Returns the bit width of the signal associated with the segment. (Returns 1 for scalar signals.)

### startIndex

`startIndex(seg)`

Returns the start index of the signal associated with the segment.

### stopIndex

`stopIndex(seg)`

Returns the stop index of the signal associated with the segment.

### upperY

`upperY(`*`seg`*`)`

Returns the upper right y-coordinate of the bounding box of the segment.

### variance

`sigma(`*`list`*`)`

Finds the mathematical variance of values in the list. All the values in the list must be integers or floating points.

### vector

`vector(`*`seg`*`)`

Returns the segment if its signal is a vector.

### vertical

`vertical(`*`seg`*`)`

Returns the segment if it is vertical.

### wire

`wire(`*`seg`*`)`

Returns the wire of the segment.

### wireprop

`wireprop(`*`seg`*`)`

Returns the list of properties on the wire to which the segment belongs.

### x1

`x1(`*`seg`*`)`

Returns the starting x-coordinate of the segment.

### x2

`x2(`*seg*`)`

Returns the ending x-coordinate of the segment.

### y1

`y1(`*seg*`)`

Returns the starting y-coordinate of the segment.

### y2

`y2(`*seg*`)`

Returns the ending y-coordinate of the segment.

## Object Note Predicates

This section describes the object note predicates available in the graphical environment. These predicates operate on every note in the logic drawing. The predicates are listed in alphabetical order.

### bbox

`bbox(note)`

Returns the bounding box of the note.

### color

`color(note)`

Returns the color of the note.

### designName

`designName(note)`

Returns the name of the logic drawing that contains the segment.

### distance

`distance(note, bbox)`

Finds the distance between the note and the bounding box specified as the second argument.

### fileName

`fileName(note)`

Returns *<cell>/<view>*`page` *<pagenum>*`.csb`.

### just

`just(note)`

Returns the justification of the note.

### leftX

`leftX(note)`

Returns the lower-left X value of the bounding box of the note.

### lowerY

`lowerY(note)`

Returns the lower-left Y value of the bounding box of the note.

### matchName

`matchName(note, string)`

Returns the note if its name in the schematic matches the second argument.

### matchNoName

`matchNoName(note, string)`

Returns the note if its name in the schematic does not match the second argument.

### name

`name(note)`

Returns the text of the note.

### orient

`orient(note)`

Returns the orientation of the note.

### overlap

`overlap(note, value)`

Returns the note if it overlaps with the bounding box specified as the second argument, which should be in the format ((x,y),(x,y)).

### path

`path(note)`

Returns the path of the design library that contains the note.

### rightX

`rightX(note)`

Returns the upper-right X value of the bounding box of the note.

### size

`size(note)`

Returns the font size of the text of the note.

### upperY

`upperY(note)`

Returns the upper-right Y value of the bounding box of the note.

## Object Instance Predicates

This section describes the object instance predicates available in the graphical environment. These predicates operate on every instance in the logic drawing. The predicates are listed in alphabetical order.

### bbox

`bbox(`*`inst`*`)`

Returns the bounding box of the instance.

### body

`body(`*`inst`*`)`

Returns the body of the instance.

### checkSection

`checkSection(`*`inst`*`)`

Returns the value of Section on the instance.

### color

`color(`*`inst`*`)`

Returns the color of the instance.

### designname

`designname(`*`inst`*`)`

Returns the name of the current design.

### distance

`distance(`*`inst, value`*`)`

Finds the distance between the instance and the box specified as the second argument.

### fileName

`fileName(`*`instance`*`)`

Returns *<cell>/<view>*`page` *<pagenum>*`.csb`.

### getProperty

`getProperty(`*`inst, string`*`)`

Gets the value of the property specified as the second argument.

### getPropertyFloat

`getPropertyFloat(`*`inst, string`*`)`

Gets the floating-point value of the property specified as the second argument.

### getPropertyInt

`getPropertyInt(`*`inst, string`*`)`

Gets the integer value of the property specified as the second argument.

### hasProperty

`hasProperty(`*`inst,string`*`)`

Returns the instance if the property specified as the second argument exists on it.

### leftX

`leftX(`*`inst`*`)`

Returns the lower left x-coordinates of the bounding box of the instance.

### lowerY

`lowerY(`*`inst`*`)`

Returns the lower left y-coordinate of the bounding box of the instance.

### matchName

`matchName(inst,string)`

Returns the instance if its name in the schematic matches the second argument.

### matchNoName

`matchNoName(inst,string)`

Returns the instance if its name in the schematic does not match the second argument.

### matchProperty

`matchProperty(inst,string,string)`

Returns the instance in case the property specified as the second argument exists on it and matches the third argument.

### name

`name(inst)`

Returns the name (that is, value Path Property) of the instance.

### orient

`orient(inst)`

Returns the orientation of the instance.

### overlap

`overlap(inst,string)`

Returns the instance if it overlaps with the bounding box specified as the second argument, which needs to be in the form ((x,y),(x,y)).

### path

`path(inst)`

Returns the path of the design library that contains the instance.

### physPackType

`physPackType(`*`inst`*`)`

Returns a list of all packaging of the instance from the `chips_prt` file.

### pin

`pin(`*`inst`*`)`

Returns the pin list of the instance.

### prop

`prop(`*`inst`*`)`

Returns a list of properties on the instance.

### rightX

`rightX(`*`isnt`*`)`

Returns the upper right x-coordinate of the bounding box of the instance.

### toolstartFail

`toolstartFail(`*`inst,call_string`*`)`

Returns the instance on which tool invoked in *`call_string`* fails that is the exit status is 1.

### toolstartIgnore

`toolstartignore(`*`inst,call_string`*`)`

Returns the instance ignoring return-status of tool invoked in *`call_string`*.

### toolstartPass

`toolstartPass(`*`inst,call_string)`*

Returns the instance on which tool invoked in *`call_string`* passes, that is the exit status is 0.

### unnamed

`unnamed(`*`inst`*`)`

Returns the instance if it has no PATH property.

### upperY

`upperY(`*`inst`*`)`

Returns the upper right y-coordinate of the bounding box of the instance.

## Object Property Predicates

This section describes the object property predicates available in the graphical environment. These predicates operate on properties in the logic drawing. The predicates are listed in alphabetical order.

### bbox

`bbox(`*`prop`*`)`

Returns the bounding box of the visible portion of the property.

### bodypin

`bodypin(`*`prop`*`)`

Returns the bodypin associated with the property.

### body

`body(`*`prop`*`)`

Returns the body attached to the property.

### color

`color(`*`prop`*`)`

Returns the color of the property.

### designname

`designname(`*`prop`*`)`

Returns the name of the current design.

### distance

`distance(`*`prop`*`, `*`bbox`*`)`

Finds the distance between the property and the bounding box specified as the second argument.

### fileName

`fileName(prop)`

Returns `<cell>/<view>page <pagenum>.csb`.

### inst

`inst(prop)`

Returns the instance attached to the property.

### just

`just(prop)`

Returns the justification of the property.

### leftX

`leftX(prop)`

Returns the lower-left x value of the bounding box of the property.

### lowerY

`lowerY(prop)`

Returns the lower left y-value of the bounding box of the property.

### matchName

`matchName(prop,string)`

Returns the property if its name in the schematic matches the second argument.

### matchNoName

`matchNoName(prop,string)`

Returns the property if its name in the schematic does not match the second argument.

### name

`name(prop)`

Returns the name of the property.

### nameInvis

`nameInvis(prop)`

Returns the property if its name is invisible in the schematic.

### nameOrValueVis

`nameOrValueVis(prop)`

Returns the property if its name or value is visible in the schematic.

### nameVis

`nameVis(prop)`

Returns the property if its name is visible in the schematic.

### orient

`orient(prop)`

Returns the orientation of the property.

### overlap

`overlap(prop,bbox)`

Returns the property if it overlaps with the bounding box specified as the second argument, which needs to be in the format ((x,y),(x,y)).

### path

`path(prop)`

Returns the path of the design that contains the property.

### pin

`pin(`*prop*`)`

Returns the pin associated with the property.

### rightX

`rightX(`*prop*`)`

Returns the upper-right x value of the bounding box of the property.

### segs

`segs(`*prop*`)`

Returns the list of segments (which constitutes a wire) to which the property is attached.

### size

`size(`*prop*`)`

Returns the font size of the text of the property.

### upperY

`upperY(`*prop*`)`

Returns the upper right y-value of the bounding box of the property.

### value

`value(`*prop*`)`

Returns the value of the property.

### valueInvis

`valueInvis(`*prop*`)`

Returns the property if its value is invisible in the schematic.

### valueVis

`valueVis(`*`prop`*`)`

Returns the property if its value is visible in the schematic.

## Object Body Predicates

This section describes the object body predicates available in the graphical environment. These predicates operate on every body used in the design. The predicates are listed in alphabetical order.

### bodypin

`bodypin(`*`body`*`)`

Returns the list of pins on the body.

### designname

`designname(`*`body`*`)`

Returns the name of the current design.

### fileName

`fileName(`*`body`*`)`

Returns *`<cell>/<view>`*`symbol.css`.

### getProperty

`getProperty(`*`body,string`*`)`

Gets the value of the property specified by the second argument.

### getPropertyFloat

`getProperty(`*`body,string`*`)`

Gets the floating-point value of the property specified by the second argument.

### getPropertyInt

`getPropertyInt(`*`body,string`*`)`

Gets the integer value of the property specified by the second argument.

### hasProperty

`hasProperty(`*`body, string`*`)`

Returns the body if the property specified as the second argument exists on it.

### inst

`inst(`*`body`*`)`

Returns a list with the instances of the body in the design.

### logicExists

`logicExists(`*`body`*`)`

Returns true if the body is a hierarchical drawing (that is, if `logic.1.1` or `logic_bn.1.1` exists for it).

### matchName

`matchName(`*`body,string`*`)`

Returns the body if its name in the schematic matches the second argument.

### matchNoName

`matchNoName(`*`body, string`*`)`

Returns the body if its name in the schematic does not match the second argument.

### matchProperty

`matchProperty(`*`body, string,string`*`)`

Returns the body if the property (specified as the second argument) has a value that matches the third argument.

### name

`name(`*`body`*`)`

Returns the name of the body.

### path

`path(`*`body`*`)`

Returns the path of the library of the body.

### prop

`prop(`*`body`*`)`

Returns a list of the properties on the body.

## Object Pin Predicates

This section describes the object pin predicates available in the graphical environment. These predicates operate on all pins attached to instances in the logic drawing. The predicates are listed in alphabetical order.

### bodypin

`bodypin(`*`pin`*`)`

Returns the bodypin of the pin.

### designname

`designname(`*`pin`*`)`

Returns the name of the current design.

### dir

`dir(`*`pin`*`)`

Returns the direction (EAST, WEST, NORTH, SOUTH) of the pin relative to the body.

### fileName

`file(`*`pin`*`)`

Returns `<`*`cell`*`>/<`*`view`*`>page <`*`pagenum`*`>.csb`.

### getProperty

`getProperty(`*`pin,string`*`)`

Gets the value of the property specified as the second argument.

### getPropertyFloat

`getPropertyFloat(`*`pin,string`*`)`

Gets the floating-point value of the property specified as the second argument.

### getPropertyInt

`getPropertyInt(`*`pin,string`*`)`

Gets the integer value of the property specified as the second argument.

### hasProperty

`hasProperty(`*`pin,string`*`)`

Returns the pin if the property specified as the second argument exists on it.

### inout

`inout(`*`pin`*`)`

Case pin_dir_check option is ON (default):

Returns the pin if BIDIRECTIONAL property is there on the pin.

Case pin_dir_check option is OFF:

Returns the pin if BIDIRECTIONAL or INPUT_LOAD and OUTPUT_LOAD property is present.

### inOrInout

`inOrInout(`*`pin`*`)`

Case pin_dir_check option is ON (default):

Returns the pin if INPUT_LOAD or BIDIRECTIONAL property is there on the pin.

Case pin_dir_check option is OFF:

Returns the pin if none of the properties (BIDIRECTIONAL, INPUT_LOAD,OUTPUT_LOAD) is present or INPUT_LOAD or BIDIRECTIONAL or both INPUT_LOAD and OUTPUT_LOAD property is present.

### input

`input(`*`pin`*`)`

Case pin_dir_check option is ON (default):

Returns the pin if INPUT_LOAD property is there on the pin.

Case pin_dir_check option is OFF:

Returns the pin if none of the properties (BIDIRECTIONAL, INPUT_LOAD,OUTPUT_LOAD) is present or if INPUT_LOAD property is only present.

### inst

`inst(`*`pin`*`)`

Returns the instance to which the pin belongs.

### matchName

`matchName(`*`pin,string`*`)`

Returns the pin if its name in the schematic matches the second argument.

### matchNoName

`matchNoName(`*`pin, string`*`)`

Returns the pin if its name in the schematic does not match the second argument.

### matchProperty

`matchProperty(`*`pin,string,string`*`)`

Returns the pin if the property matched as the second argument exists on it, and its value matches the third argument.

### name

`name(`*`pin`*`)`

Returns the name of the pin.

### outOrInout

`outOrInout(`*`pin`*`)`

Case *pin_dir_check* option is ON (default):

Returns the pin if OUTPUT_LOAD or BIDIRECTIONAL property is there on the pin.

Case pin_dir_check option is OFF:

Returns the pin if OUTPUT_LOAD or BIDIRECTIONAL or both INPUT_LOAD and OUTPUT_LOAD property is present.

### output

`output(`*`pin`*`)`

Case pin_dir_check option is ON (default):

Returns the pin if OUTPUT_LOAD property is there on the pin.

Case pin_dir_check option is OFF:

Returns the pin if OUTPUT_LOAD property is there on the pin.

### path

`path(`*`pin`*`)`

Returns the path of the design library that contains the pin.

### prop

`prop(`*`pin`*`)`

Returns a list of the properties attached to the pin.

### seg

`seg(`*`pin`*`)`

Returns the segments connected to the pin.

### xcoord

`xcoord(`*`pin`*`)`

Returns the x-coordinate of the pin.

## ycoord

`ycoord(`*`pin`*`)`

Returns the y-coordinate of the pin.

## Object Bodypin Predicates

This section describes the object bodypin predicates available in the graphical environment. These predicates operate once on each bodypin used in the logic drawing. The predicates are listed in alphabetical order.

### body

`body(`*`bodypin`*`)`

Returns the body of the bodypin.

### designname

`designname(`*`bodypin`*`)`

Returns the name of the current design.

### dir

`dir(`*`bodypin`*`)`

Returns the direction (EAST, WEST, NORTH, SOUTH) of the bodypin.

### fileName

`fileName(`*`bodypin`*`)`

Returns *<cell>/<view>*`symbol.css`.

### getProperty

`getProperty(`*`bodypin,string`*`)`

Gets the value of the property specified as the second argument.

### getPropertyFloat

`getPropertyFloat(`*`bodypin,string`*`)`

Gets the floating-point value of the property specified as the second argument.

### getPropertyInt

`getPropertyInt(`*`bodypin,string`*`)`

Gets the integer value of the property specified as the second argument.

### hasProperty

`hasProperty(`*`bodypin,string`*`)`

Returns the bodypin if the property specified as the second argument exists on it.

### inout

`inout(`*`bodypin`*`)`

Case pin_dir_check option is ON (default):

Returns the bodypin if BIDIRECTIONAL property is there on the bodypin.

Case pin_dir_check option is OFF:

Returns the bodypin if BIDIRECTIONAL or INPUT_LOAD and OUTPUT_LOAD property is present.

### inOrInout

`inOrInout(`*`bodypin`*`)`

Case pin_dir_check option is ON (default):

Returns the bodypin if INPUT_LOAD or BIDIRECTIONAL property is there on the bodypin.

Case pin_dir_check option is OFF:

Returns the bodypin if none of the properties (BIDIRECTIONAL, INPUT_LOAD,OUTPUT_LOAD) is present or INPUT_LOAD or BIDIRECTIONAL or both INPUT_LOAD and OUTPUT_LOAD property is present.

### input

`input(`*`bodypin`*`)`

Case pin_dir_check option is ON (default):

Returns the bodypin if INPUT_LOAD property is there on the bodypin.

Case pin_dir_check option is OFF:

Returns the bodypin if none of the properties (BIDIRECTIONAL, INPUT_LOAD,OUTPUT_LOAD) is present or if INPUT_LOAD property is only present.

### matchName

`matchName(`*`bodypin,string`*`)`

Returns the bodypin if its name in the schematic matches the second argument.

### matchNoName

`matchNoName(`*`bodypin, string`*`)`

Returns the bodypin if its name in the schematic does not match the second argument.

### matchProperty

`matchProperty(`*`bodypin,string,string`*`)`

Returns the bodypin if the property matched as the second argument exists on it, and its value matches the third argument.

### name

`name(`*`bodypin`*`)`

Returns the name of the bodypin.

### outOrInout

`outOrInout(`*`bodypin`*`)`

Case pin_dir_check option is ON (default):

Returns the bodypin if OUTPUT_LOAD or BIDIRECTIONAL property is there on the bodypin.

Case pin_dir_check option is OFF:

Returns the bodypin if OUTPUT_LOAD or BIDIRECTIONAL or both INPUT_LOAD and OUTPUT_LOAD property is present.

### output

`output(`*`bodypin`*`)`

Case pin_dir_check option is ON (default):

Returns the bodypin if OUTPUT_LOAD property is there on the bodypin.

Case pin_dir_check option is OFF:

Returns the bodypin if OUTPUT_LOAD property is there on the bodypin.

### PassThru

`PassThru(`*`bodypin`*`)`

Returns the bodypin if it is a pass-thru pin.

### path

`path(`*`bodypin`*`)`

Returns the path of the library of the body that contains the bodypin.

### prop

`prop(`*`bodypin`*`)`

Returns a list of the properties attached to the body pin.

## Object Wire Predicates

This section describes the object wire predicates available in the graphical environment. These predicates operate on all wires in the logic drawing. The predicates are listed in alphabetical order.

### baseName

`baseName(`*`wire`*`)`

Returns the name of the signal, omitting the macros at the end.

### designname

`designname(`*`wire`*`)`

Returns the name of the current design.

### fileName

`fileName(`*`wire`*`)`

Returns *<cell>/<view>page <pagenum>.csb.*

### getProperty

`getProperty(`*`wire, string`*`)`

Gets the value of the property specified as the second argument.

### getPropertyFloat

`getPropertyFloat(`*`wire, string`*`)`

Gets the floating-point value of the property specified as the second argument.

### getPropertyInt

`getPropertyInt(`*`wire, string`*`)`

Gets the integer value of the property specified as the second argument.

### hasProperty

`hasProperty(`*`wire,string`*`)`

Returns the wire if the property specified as the second argument exists on it.

### matchName

`matchName(`*`wire,string`*`)`

Returns the wire if its name in the schematic matches the second argument.

### matchNoName

`matchNoName(`*`wire,string`*`)`

Returns the wire if its name in the schematic does not match the second argument.

### matchNoType

`matchNoType(`*`wire,string`*`)`

Returns the wire if its name does not match the second argument.

### matchProperty

`matchProperty(`*`wire,string,string`*`)`

Returns the wire if the property specified as the second argument exists on it and matches the third argument.

### matchType

`matchType(`*`wire,string`*`)`

Returns the wire if its name contains the second argument.

### name

`name(`*`wire`*`)`

Returns the name of the wire.

### pageNumbers

`pagenumbers( design, wire)`

Returns the page numbers of the design in which the wire exists.

### path

`path(wire)`

Returns the path of the design library that contains the wire.

### prop

`prop(wire)`

Returns the list of properties on the wire.

### scalar

`scalar(wire)`

Returns the wire if its signal is a scalar.

### seg

`seg(wire)`

Returns the list of segments that comprise the wire.

### signalWidth

`signalWidth(wire)`

Returns the bit width of the signal associated with the wire. (Returns 1 for scalar signals.)

### startIndex

`startIndex(wire)`

Returns the start index of the signal associated with the wire.

### stopIndex

`stopIndex(`*`wire`*`)`

Returns the stop index of the signal associated with the wire.

### vector

`vector(`*`wire`*`)`

Returns the wire if its signal is a vector.

## Object PhysPackType Predicates

This section describes the object PhysPackType predicates available in the graphical environment. These predicates operate once on each primitive listed in the `chips_prt` file corresponding to the design.The predicates are listed in alphabetical order.

### designname

`designname(`*`physPackType`*`)`

Returns the name of the design.

### fileName

`fileName(`*`physPackType`*`)`

Returns *<cell>/<view>*`page` *<pagenum>*`.csb`.

### getAllPropNames

`getAllPropNames(`*`physPackType`*`)`

Returns a list of the property names in the `chips_prt` file.

### getPinsOfTerm

`getPinsOfTerm(`*`physPackType, string`*`)`

Returns the pin numbers of the term of the primitive specified as the second argument.

### getProperty

`getProperty(`*`physPackType, string`*`)`

Returns the value of the property (specified as the second argument) if it exists on the primitive.

### getPropertyFloat

`getPropertyFloat(`*`physPackType, string`*`)`

Returns the floating-point value of the property (specified as the second argument) if it exists on the primitive.

### getPropertyInt

`getPropertyInt(`*`physPackType, string`*`)`

Returns the integer value of the property (specified as the second argument) if it exists on the primitive.

### getSectOfPin

`getSectOfPin(`*`physPackType,string`*`)`

Returns the section number in which the pin number specified as the second argument occurs in the primitive specified by the first argument.

### getSectPins

`getSectPins(`*`physPackType,string`*`)`

Returns the list of pin names that occur in the section specified as the second argument.

### getTermAllPropNames

`getTermAllPropNames(`*`physPackType,string`*`)`

Returns a list of all properties that exist on the term of the primitive specified as the second argument.

### getTermOfPin

`getTermOfPin(`*`physPackType,string`*`)`

Returns the name of the term for the pin number specified in the second argument for the primitive specified as the first argument.

### getTermProperty

`getTermProperty(`*`physPackType,string,string`*`)`

Returns the value of the property specified as the third argument if it exists on the term specified as the second argument.

### hasProperty

`hasProperty(`*`physPackType,string`*`)`

Returns the primitive if the property specified as the second argument exists on it.

### hasTermProperty

`hasTermProperty(`*`physPackType,string,string`*`)`

Returns the primitive if value of the property specified as the third argument exists on the term specified as the second argument.

### matchName

`matchName(`*`physPackType, value`*`)`

Returns the primitive if its name in the schematic matches the second argument.

### matchNoName

`matchNoName(`*`physPackType, value`*`)`

Returns the primitive if its name in the schematic does not match the second argument.

### matchProperty

`matchProperty(`*`physPackType, value, value`*`)`

Returns the primitive if the property specified as the second argument exists on it and has a value that matches the third argument.

### name

`name(`*`physPackType`*`)`

Returns the primitive name in the `chips_prt` file.

### numSection

`numSection(`*`physPackType`*`)`

Returns the number of sections in the primitive.

### path

`path(`*`physPackType`*`)`

Returns the path of the design library that contains the primitive.

### term

`term(`*`physPackType`*`)`

Returns a list of term names.

## Miscellaneous Predicates

The following predicates are available in the body environment.

### mapToPhysPackName

`mapToPhysPackName(`*`value`*`)`

Returns the values corresponding to *`value`* in the `chips_prt` file.

### overlap

`overlap(`*`value, value`*`)`

Returns True if the bounding boxes specified in the two arguments overlap. The first and second arguments are strings in the format *`(x1,y1),(x2,y2)`*.

# Logical Environment Predicates

This section describes the predicates available for creating rules in the logical environment.

The logical environment uses the following primary objects:

- design is the complete compiled drawing (all pages).
- inst is an instance of a body in the drawing.
- sig is a signal corresponding to a logical signal in the design.
- term is a pin attached to a body (not an instance) in the design.
- instTerm is a pin attached to an instance of a body.
- body is a body type (from a unique listing of instances) used in the design.

## Object Design Predicates

This section describes the object design predicates available in the logical environment. These predicates operate on the entire compiled design. The predicates are listed in alphabetical order.

### body

`body(`*`design`*`)`

Returns a list of all part types in *`design`*.

### getAllPropNames

`getAllPropNames(`*`design`*`)`

Returns a list of all properties attached to *`design`*.

### getDuplicateProps

`getDuplicateProps(`*`design`*`)`

Returns a list of properties that have more than one occurrence on *`design`*.

### getProperty

`getProperty(`*`design`*`, `*`property`*`)`

Returns the value of *`property`* if *`property`* exists on the design. If there are multiple occurrences of *`property`* in the design, the predicate returns a list of all values, and a message with the property names and design name is printed in the `cp.lst` file. Otherwise it returns NULL.

### getPropertyFloat

`getPropertyFloat(`*`design`*`, `*`property`*`)`

Returns the floating point value of *`property`* if *`property`* exists on the design. If there are multiple occurrences of *`property`* in the design, the predicate returns a list of all floating point values and a message with the property names and design name is printed in the `cp.lst` file. Otherwise it returns NULL.

### getPropertyInt

`getPropertyInt(`*`design`*`, `*`property`*`)`

Returns the integer value of *`property`* if *`property`* exists on the design. If there are multiple occurrences of *`property`* in the design, the predicate returns a list of all integer values and a message with the property names and design name is printed in the `cp.lst` file. Otherwise it returns NULL.

### getPropertyFloatWith Warn

`getPropertyFloatWithWarn(`*`design`*`, `*`property`*`)`

Returns the floating point value of *`property`* if *`property`* exists on the design. If the predicate finds multiple occurrences of *`property`*, it returns a list of all floating point values and a message with the property names and design name is printed in the `cp.lst` file. Otherwise it returns NULL and prints a message in the `cp.lst` file.

### getPropertyIntWith Warn

`getPropertyIntWithWarn(`*`design`*`, `*`property`*`)`

Returns the integer value of *`property`* if *`property`* exists on the design. If the predicate finds multiple occurrences of *`property`*, it returns a list of all integer values, and a message with the property names and design name is printed in the `cp.lst` file. Otherwise it returns NULL and prints a message in the `cp.lst` file.

### getPropertyWith Warn

`getPropertyWithWarn(`*`design`*`, `*`property`*`)`

Returns the value of *`property`* if *`property`* exists on the design. If the predicate finds multiple occurrences of *`property`*, it returns a list of all values, and a message with the property names and design name is printed in the `cp.lst` file. Otherwise it returns NULL and prints a message in the `cp.lst` file.

### hasNoProperty

`hasNoProperty(`*`design`*`, `*`property`*`)`

Returns the design object if *`property`* does not exist on it. Otherwise it returns NULL.

### hasProperty

`hasProperty(`*`design`*`, `*`property`*`)`

Returns the design object if *`property`* exists on it. Otherwise, it returns NULL.

### hasPropertyWith Warn

`hasPropertyWithWarn(`*`design`*`, `*`property`*`)`

Returns the design object if *`property`* exists on it. Otherwise it returns NULL and prints a warning message in the `cp.lst` file.

### inst

`inst(`*`design`*`)`

Returns a list of all bodies (instances) in *`design`*.

### instTerm

`instTerm(`*`design`*`)`

Returns a list of all pins in *`design`*.

### matchProperty

`matchProperty(`*`design`*`, `*`property`*`, `*`value`*`)`

Returns the design object if *`property`* has a value of *`value`*. If there are multiple occurrences of *`property`* in the design, the predicate returns a list of all values, and a message with the property names and design name is printed in the `cp.lst` file.

### name

`name(`*`design`*`)`

Returns the name of *`design`*.

### sig

`sig(`*`design`*`)`

Returns a list of all nets (signals) in *`design`*.

### term

`term(`*`design`*`)`

Returns a list of all terminals in *`design`*.

### toolstartFail

`toolstartFail(`*`design,call_string`*`)`

Returns the design on which tool invoked in *`call_string`* fails, that is the exit status is 1.

### toolstartIgnore

`toolstartignore(`*`design,call_string`*`)`

Returns the design ignoring return-status of tool invoked in *`call_string`*.

### toolstartPass

`toolstartPass(`*`design,call_string`*`)`

Returns the design on which tool invoked in *`call_string`* passes, that is the exit status is 0.

## Object Instance Predicates

This section describes the object instance predicates available in the logical environment. These predicates operate on each instance of a body on your schematic. The predicates are listed in alphabetical order.

### body

`body(`*`inst`*`)`

Returns the body or part type (whichever is appropriate) of *`inst`*.

### getAllPropNames

`getAllPropNames(`*`inst`*`)`

Returns a list of all properties attached to *`inst`*.

### getDuplicateProps

`getDuplicateProps(`*`inst`*`)`

Returns a list of properties that have more than one occurrence on *`inst`*.

### getHierProperty

`getHierProperty(`*`inst, property`*`)`

Returns the value of *`property`* if *`property`* is attached to *`inst`*; if not, it returns the value of *`property`* if *`property`* is attached to the part type of *`inst`*. If the predicate finds multiple occurrences of *`property`*, it returns a list of all values, and a message with the property names and instance name is printed in the `cp.lst` file. If *`property`* is not attached to either of these objects, it returns NULL.

### getHierPropertyFloat

`getHierPropertyFloat(`*`inst, property`*`)`

Returns the floating point value of *`property`* if *`property`* is attached to *`inst`*; if not, it returns the floating point value of *`property`* if *`property`* is attached to the part type of *`inst`*. If the predicate finds multiple occurrences of *`property`*, it returns a list of all values and a message with the property names and instance name is printed in the `cp.lst` file. If *`property`* is not attached to either of these objects, it returns NULL.

### getHierPropertyInt

`getHierPropertyInt(`*`inst`*`, `*`property`*`)`

Returns the integer value of *property* if *property* is attached to *inst*; if not, it returns the integer value of *property* if *property* is attached to the part type of *inst*. If the predicate finds multiple occurrences of *property*, it returns a list of all integer values, and a message with the property names and instance name is printed in the `cp.lst` file. If *property* is not attached to either of these objects, it returns NULL

### getHierPropertyFloatWithWarn

`getHierPropertyWithWarn(`*`inst`*`, `*`property`*`)`

Returns the floating point value of *property* if *property* is attached to *inst*; if not, it returns the floating point value of *property* if *property* is attached to the part type of *inst*. If the predicate finds multiple occurrences of *property*, it returns a list of all integer values, and a message with the property names and instance name is printed in the `cp.lst` file. If *property* is not attached to either of these objects, it returns NULL and prints a message in the `cp.lst` file.

### getHierPropertyIntWithWarn

`getHierPropertyIntWithWarn(`*`inst`*`, `*`property`*`)`

Returns the integer value of *property* if *property* is attached to *inst*; if not, it returns the integer value of *property* if *property* is attached to the part type of *inst*. If the predicate finds multiple occurrences of *property*, it returns a list of all integer values and a message with the property names and instance name is printed in the `cp.lst` file. If *property* is not attached to either of these objects, it returns NULL and prints a message in the `cp.lst` file.

### getHierPropertyWithWarn

`getHierPropertyWithWarn(`*`inst`*`, `*`property`*`)`

Returns the value of *property* if *property* is attached to *inst*; if not, it returns the value of *property* if *property* is attached to the part type of *inst*. If the predicate finds multiple occurrences of *property*, it returns a list of all values and a message with the property names and instance name is printed in the `cp.lst` file. If *property* is not attached to either of these objects, it returns NULL and prints a message in the `cp.lst` file.

### hasHierProperty

`hasHierProperty(`*`inst`*`, `*`property`*`)`

Returns the instance object if *`property`* is attached to *`inst`*; if not, it returns the instance object if *`property`* is attached to the part type of *`inst`*. If *`property`* is not attached to any of these objects, it returns NULL.

### hasHierProperty WithWarn

`hasHierPropertyWithWarn(`*`inst`*`, `*`property`*`)`

Returns the instance object if *`property`* is attached to *`inst`*; if not, it returns the instance object if *`property`* is attached to the part type of *`inst`*. If *`property`* is not attached to either of these objects, it returns NULL and prints a message in the `cp.lst` file.

### hasNoBody

`hasNoBody(`*`inst`*`)`

Returns the instance object if *`inst`* does not have a body. Mainly used to check for flag bodies.

### hasNoHierProperty

`hasNoHierProperty(`*`inst`*`, `*`property`*`)`

Returns the instance object if *`property`* is not attached to *`inst`*. If it is attached, it returns the instance object if *`property`* is not attached to the part type of *`inst`*. If *`property`* is attached to either of these objects, it returns NULL.

### instTerm

`instTerm(`*`inst`*`)`

Returns a list of all pins on *`inst`*.

### matchHierProperty

`matchHierProperty(`*`inst`*`, `*`property`*`, `*`value`*`)`

Returns the instance object if *`property`* with value *`value`* attached to *`inst`*; if not, it returns the instance object if *`property`* with value *`value`* attached to the part type of *`inst`*. If the predicate finds multiple occurrences of *`property`*, it returns a list of all values,

and a message with the property names and instance name printed in the `cp.lst` file. If *`property`* with value *`value`* is not attached to either of these objects, it returns NULL.

### matchName

`matchName(`*`inst`*`, `*`string`*`)`

Returns the instance object if *`string`* matches the schematic or hierarchical name of *`inst`*. Otherwise, it returns NULL.

### matchNHierProperty

`matchNHierProperty(`*`inst`*`, `*`property`*`, `*`value`*`)`

Returns *`inst`* if *`property`* with a value that contains substring *`value`* is found on (in order) the instance or the body of the instance. If *`property`* with value *`value`* is not attached to either of these objects, it returns NULL.

### matchNoHier Property

`matchNoHierProperty(`*`inst`*`, `*`property`*`, `*`value`*`)`

Returns the instance object if it does not have a *`property`* property with value *`value`*.

### matchNoName

`matchNoName(`*`inst`*`, `*`string`*`)`

Returns the instance object if there is no match between string and the schematic or hierarchical name of *`inst`*. Otherwise, else it returns NULL.

### name

`name(`*`inst`*`)`

Returns the name of *`inst`*.

### sig

`sig(`*`inst`*`)`

Returns a list of all signals (nets) connected to the *`inst`* through its pins. Equivalent to *`sig(instTerm(inst))`* but more efficient.

### toolstartFail

`toolstartFail(`*`inst,call_string`*`)`

Returns the instance on which tool invoked in *`call_string`* fails, that is the exit status is 1.

### toolstartIgnore

`toolstartignore(`*`inst,call_string`*`)`

Returns the instance ignoring return-status of tool invoked in *`call_string`*.

### toolstartPass

`toolstartPass(`*`inst,call_string`*`)`

Returns the instance on which tool invoked in *`call_string`* passes, that is the exit status is 0.

## Object Signal Predicates

This section describes the object signal predicates available in the logical environment. These predicates operate on signals (nets) in your compiled design. The predicates are listed in alphabetical order.

### getAllPropNames

`getAllPropNames(`*`signal`*`)`

Returns a list of all properties attached to *`signal`*.

### getDuplicateProps

`getDuplicateProps(`*`signal`*`)`

Returns a list of properties that have more than one occurrence on *`signal`*.

### getProperty

`getProperty(`*`signal, property`*`)`

Returns the value of *`property`* if it exists on *`signal`*. If there are multiple occurrences of *`property`* on *`signal`*, the predicate returns a list of all values, and a message with the property names and signal name is printed in the `cp.lst` file. Otherwise, it returns NULL.

### getPropertyFloat

`getPropertyFloat(`*`signal, property`*`)`

Returns the floating point value of *`property`* if it exists on *`signal`*. If there are multiple occurrences of *`property`* on *`signal`*, the predicate returns a list of all floating point values and a message with the property names and signal name is printed in the `cp.lst` file. Otherwise, it returns NULL.

### getPropertyInt

`getPropertyInt(`*`signal, property`*`)`

Returns the integer value of *`property`* if it exists on *`signal`*. If there are multiple occurrences of *`property`* on *`signal`*, the predicate returns a list of all integer values and a message with the property names and signal name is printed in the `cp.lst` file. Otherwise, it returns NULL.

### getPropertyFloatWith Warn

`getPropertyFloatWithWarn(`*`signal`*`, `*`property`*`)`

Returns the floating point value of *`property`* if it exists on *`signal`*. Otherwise, it returns NULL, and prints an error message in the `cp.lst` file.

### getPropertyIntWith Warn

`getPropertyIntWithWarn(`*`signal`*`, `*`property`*`)`

Returns the integer value of *`property`* if it exists on *`signal`*. Otherwise, it returns NULL, and prints an error message in the `cp.lst` file.

### getPropertyWith Warn

`getPropertyWithWarn(`*`signal`*`, `*`property`*`)`

Returns the value of *`property`* if it exists on *`signal`*. Otherwise, it returns NULL and prints an error message in the `cp.lst` file.

### getSupplyVolt

`getSupplyVolt (`*`signal`*`, `*`value`*`)`

Returns the supply voltage for the signal.

### hasProperty

`hasProperty(`*`signal`*`, `*`property`*`)`

Returns the signal object if *`property`* exists on *`signal`*. Otherwise, it returns NULL.

### hasPropertyWith Warn

`hasPropertywithWarn(`*`signal`*`, `*`property`*`)`

Returns the signal object if *`property`* exists on *`signal`*. Otherwise, it returns NULL and prints an error in the `cp.lst` file.

### hasSynonyms

`hasSynonyms(`*`signal`*`)`

Returns the synonym names if *signal* has synonyms.

### inst

`inst(`*signal*`)`

Returns a list of all instances of the pins on *signal*. Equivalent to `inst(`*instTerm(signal)*`)`, but more efficient.

### instTerm

`instTerm(`*signal*`)`

Returns a list of all pins on *signal*.

### isGlobal

`isGlobal(`*signal*`)`

Returns the signal if *signal* is global. Otherwise, it returns NULL.

### isInterface

`isInterface(`*signal*`)`

Returns the signal if *signal* is an interface signal. Otherwise, it returns NULL.

### isLocal

`isLocal(`*signal*`)`

Returns the signal if *signal* is a local signal. Otherwise, it returns NULL.

### isNotGlobal

`isNotGlobal(`*signal*`)`

Returns the signal if *signal* is not a global signal. Otherwise, it returns NULL.

### isNotInterface

`isNotInterface(`*signal*`)`

Returns the signal if *signal* is not an interface signal. Otherwise, it returns NULL.

### matchName

`matchName(`*signal*`, `*string*`)`

Returns the signal object if *string* matches the schematic or hierarchical name of *signal*. Otherwise, it returns NULL.

### matchNoName

`matchNoName(`*signal*`, `*string*`)`

Returns the signal object if *string* does not match the schematic or hierarchical name of *signal*. Otherwise, it returns NULL.

### matchProperty

`matchProperty(`*signal*`, `*property*`, `*value*`)`

Returns the signal object if *property* has a value of *value*. If there are multiple occurrences of *property* on the signal, the predicate returns a list of all values and a message with the property names and signal name is printed in the `cp.lst` file. Otherwise, it returns NULL.

### name

`name(`*signal*`)`

Returns the name of *signal*.

### named

`named(`*signal*`)`

Returns the signal if *signal* is a named net.

### NoOfPages

`NoOfPages(`*signal*`)`

Returns the number of pages on which *signal* occurs.

### shorted

`shorted(`*`signal`*`, `*`value1`*`, `*`value2`*`)`

Returns the signal if it (or its synonyms) matches both *`value1`* and *`value2`*. Otherwise, it returns NULL. This can be used when checking for a short between any two signals.

### term

`term(`*`signal`*`)`

Returns a list of all terminals on *`signal`*.

### unnamed

`unnamed(`*`signal`*`)`

Returns the signal if *`signal`* is an unnamed signal.

## Object Terminal Predicates

This section describes the object terminal predicates available in the logical environment. These predicates operate on the interface terminals types of a drawing (the body pins attached the body of a drawing). The predicates are listed in alphabetical order.

### getProperty

`getProperty(`*`term`*`, `*`property`*`)`

Returns the value of *`property`* if it exists on *`term`*. If there are multiple occurrences of *`property`* on the object terminal, the predicate returns a list of all values, and a message with the property names and object terminal names is printed in the `cp.lst` file.Otherwise, it returns NULL.

### getPropertyFloat

`getPropertyFloat(`*`term`*`, `*`property`*`)`

Returns the floating point value of *`property`* if it exists on *`term`*. If there are multiple occurrences of *`property`* on the object terminal, the predicate returns a list of all floating point values and a message with the property names and object terminal names is printed in the `cp.lst` file.Otherwise, it returns NULL.

### getPropertyInt

`getPropertyInt(`*`term`*`, `*`property`*`)`

Returns the integer value of *`property`* if it exists on *`term`*. If there are multiple occurrences of *`property`* on the object terminal, the predicate returns a list of all integer values and a message with the property names and object terminal names is printed in the `cp.lst` file.Otherwise, it returns NULL.

### getPropertyFloatWith Warn

`getPropertyFloatWithWarn(`*`term`*`, `*`property`*`)`

Returns the floating point value of *`property`* if it exists on *`term`*. If the predicate finds multiple occurrences of *`property`*, it returns a list of all floating point values and a message with the property names and terminal name is printed in the `cp.lst` file. Otherwise, it returns NULL, and prints an error message in the `cp.lst` file.

### getPropertyIntWith Warn

`getPropertyIntWithWarn(`*`term`*`, `*`property`*`)`

Returns the integer value of *`property`* if it exists on *`term`*. If the predicate finds multiple occurrences of *`property`*, it returns a list of all integer values and a message with the property names and terminal name is printed in the `cp.lst` file. Otherwise, it returns NULL, and prints an error message in the `cp.lst` file.

### getPropertyWith Warn

`getPropertyWithWarn(`*`term`*`, `*`property`*`)`

Returns the value of *`property`* if it exists on *`term`*. If the predicate finds multiple occurrences of *`property`*, it returns a list of all values and a message with the property names and terminal name is printed in the `cp.lst` file. Otherwise, it returns NULL, and prints an error message in the `cp.lst` file.

### hasProperty

`hasProperty(`*`term`*`, `*`property`*`)`

Returns the terminal object if *`property`* exists on *`term`*. Otherwise, it returns NULL.

### hasPropertyWith Warn

`hasPropertyWithWarn(`*`term`*`, `*`property`*`)`

Returns the terminal object if *`property`* exists on *`term`*. Otherwise, it returns NULL and prints an error in the `cp.lst` file.

### inOrInOut

`inOrInOut(`*`term`*`)`

Case *`pin_dir_check`* option is ON (default):

Returns the term if INPUT_LOAD or BIDIRECTIONAL property is there on the term.

Case *`pin_dir_check`* option is OFF:

Returns the term if none of the properties (BIDIRECTIONAL, INPUT_LOAD,OUTPUT_LOAD) is present or INPUT_LOAD or BIDIRECTIONAL or both INPUT_LOAD and OUTPUT_LOAD property is present.

### inout

`inout(term)`

Case *pin_dir_check* option is ON (default):

Returns the term if BIDIRECTIONAL property is there on the term.

Case *pin_dir_check* option is OFF:

Returns the term if BIDIRECTIONAL or INPUT_LOAD and OUTPUT_LOAD property is present.

### input

`input(term)`

Case *pin_dir_check* option is ON (default):

Returns the term if INPUT_LOAD property is there on the term.

Case *pin_dir_check* option is OFF:

Returns the term if none of the properties (BIDIRECTIONAL, INPUT_LOAD,OUTPUT_LOAD) is present or if INPUT_LOAD property is only present.

### instTerm

`instTerm(term)`

Returns a list of all pins connected to the signal that is connected to *term*.

### matchName

`matchName(term, string)`

Returns the terminal object if *string* matches the schematic or hierarchical name of *term*. Otherwise, it returns NULL.

### matchNoName

`matchNoName(term, string)`

Returns the terminal object if *string* does not match the schematic of hierarchical name of *term*. Otherwise, it returns NULL.

### matchProperty

`matchProperty(term, property, value)`

Returns the terminal object if *property* has a value of *value*. If there are multiple occurrences of *property* on the terminal, the predicate returns a list of all values and a message with the property names and terminal name is printed in the `cp.lst` file. Otherwise, it returns NULL.

### name

`name(term)`

Returns the name of *term*.

### outOrInOut

`outOrInOut(term)`

Case *pin_dir_check* option is ON (default):

Returns the term if OUTPUT_LOAD or BIDIRECTIONAL property is there on the term.

Case *pin_dir_check* option is OFF:

Returns the term if OUTPUT_LOAD or BIDIRECTIONAL or both INPUT_LOAD and OUTPUT_LOAD property is present.

### output

`output(term)`

Case *pin_dir_check* option is ON (default):

Returns the * if OUTPUT_LOAD property is there on the term.

Case *pin_dir_check* option is OFF:

Returns the term if OUTPUT_LOAD property is there on the term.

## Object Instance Terminal Predicates

This section describes the object instance terminal predicates available in the logical environment. These predicates operate on all instance terminals in the compiled design. The predicates are listed in alphabetical order.

### connected

`connected(`*`instTerm`*`)`

Returns a list of instTerms connected to *`instTerm`*.

### endpoint

`endpoint(`*`instTerm, string, string`*`)`

Returns a list of instTerms connected to the signal(s) specified as the third argument or to an instTerm of a component not specified in the second argument. Skips components specified in the second argument, which is a name-value pair (for example, “COMP_TYPE RES”).

### getAllPropNames

`getAllPropNames(`*`instTerm`*`)`

Returns a list of all properties attached to *`instTerm`*.

### getCap

`getCap(`*`instTerm`*`, `*`property`*`)`

Returns the INPUT_CAP or the OUTPUT_CAP value of *`instTerm`*, depending on the value (INPUT_CAP or OUTPUT_CAP) of the second parameter.

### getDeviceVolt

`getDeviceVolt(`*`instTerm`*`, `*`value`*`)`

Returns the device voltage corresponding to the technology of *`instTerm`* and the specified *`value`* (VCC, VEE, and so on). Device voltage information is defined in the DEVICE_VOLT macro within `cp_config.h`.

### getDuplicateProps

`getDuplicateProps(`*`instTerm`*`)`

Returns a list of properties that have more than one occurrence on *`instTerm`*.

### getHierProperty

`getHierProperty(`*`instTerm, property`*`)`

Returns the first value of *`property`* found on (in order) *`instTerm`*, pin, instance to which *`instTerm`* is attached, and the body of the instance. If the predicate finds multiple occurrences of *`property`*, it returns a list of all values and a message with the property names and instance terminal name is printed in the `cp.lst` file. If property does not exist on any of these four elements, it returns NULL.

### getHierPropertyFloat

`getHierPropertyFloat(`*`instTerm, property`*`)`

Returns the first floating point value of *`property`* found on (in order) *`instTerm`*, pin, instance to which *`instTerm`* is attached, and the body of the instance. If the predicate finds multiple occurrences of *`property`*, it returns a list of all floating point values and a message with the property names and instance terminal name is printed in the `cp.lst` file. If property does not exist on any of these four elements, it returns NULL.

### getHierPropertyInt

`getHierPropertyInt(`*`instTerm, property`*`)`

Returns the first integer value of *`property`* found on (in order) *`instTerm`*, pin, instance to which *`instTerm`* is attached, and the body of the instance. If the predicate finds multiple occurrences of *`property`*, it returns a list of all integer values and a message with the property names and instance terminal name is printed in the `cp.lst` file. If property does not exist on any of these four elements, it returns NULL.

### getHierPropertyFloatWithWarn

`getHierPropertyFloatWithWarn(`*`instTerm, property`*`)`

Returns the first floating point value of *`property`* found on (in order) *`instTerm`*, pin, instance to which *`instTerm`* is attached, and the body of the instance. If the predicate finds multiple occurrences of *`property`*, it returns a list of all floating point values and a message with the property names and instance terminal name is printed in the `cp.lst` file. If property

does not exist on any of these four elements, it returns NULL and prints an error message in the `cp.lst` file.

### getHierPropertyIntWithWarn

`getHierPropertyIntWithWarn(`*`instTerm`*`, `*`property`*`)`

Returns the first integer value of *`property`* found on (in order) *`instTerm`*, pin, instance to which *`instTerm`* is attached, and the body of the instance. If the predicate finds multiple occurrences of *`property`*, it returns a list of all integer values and a message with the property names and instance terminal name is printed in the `cp.lst` file. If property does not exist on any of these four elements, it returns NULL and prints an error message in the `cp.lst` file.

### getHierPropertyWithWarn

`getHierPropertyWithWarn(`*`instTerm`*`, `*`property`*`)`

Returns the first value of *`property`* found on (in order) *`instTerm`*, pin, instance to which *`instTerm`* is attached, and the body of the instance. If the predicate finds multiple occurrences of *`property`*, it returns a list of all values and a message with the property names and instance terminal name is printed in the `cp.lst` file. If property does not exist on any of these four elements, it returns NULL and prints an error message in the `cp.lst` file.

### hasHierProperty

`hasHierProperty(`*`instTerm`*`, `*`property`*`)`

Returns *`instTerm`* if *`property`* is attached to any one of the following: *`instTerm`*, pin, instance, or body. Otherwise, it returns NULL.

### hasHierPropertyWithWarn

`hasHierPropertyWithWarn(`*`instTerm`*`, `*`property`*`)`

Returns *`instTerm`* if *`property`* is attached to any one of the following: *`instTerm`*, pin, instance, or body. Otherwise, it returns NULL and prints an error message in the `cp.lst` file.

### hasNoHierProperty

`hasNoHierProperty(`*`instTerm`*`, `*`property`*`)`

Returns *instTerm* if *property* is not attached to any one of the following: *instTerm*, pin, instance, or body. Otherwise, it returns NULL.

### inOrInOut

`inOrInOut(`*instTerm*`)`

Case *pin_dir_check option* is ON (default):

Returns the instTerm if INPUT_LOAD or BIDIRECTIONAL property is there on the instTerm.

Case *pin_dir_check* option is OFF:

Returns the instTerm if none of the properties (BIDIRECTIONAL, INPUT_LOAD,OUTPUT_LOAD) is present or INPUT_LOAD or BIDIRECTIONAL or both INPUT_LOAD and OUTPUT_LOAD property is present.

### inout

`inout(`*instTerm*`)`

Case *pin_dir_check* option is ON (default):

Returns the instTerm if BIDIRECTIONAL property is there on the instTerm.

Case *pin_dir_check* option is OFF:

Returns the instTerm if BIDIRECTIONAL or INPUT_LOAD and OUTPUT_LOAD property is present.

### input

`input(`*instTerm*`)`

Case *pin_dir_check* option is ON (default):

Returns the instTerm if INPUT_LOAD property is there on the instTerm.

Case *pin_dir_check* option is OFF:

Returns the instTerm if none of the properties (BIDIRECTIONAL, INPUT_LOAD,OUTPUT_LOAD) is present or if INPUT_LOAD property is only present.

### input_load_h

`input_load_h(`*`instTerm`*`)`

Returns the value of the 1 state INPUT_LOAD property on *`instTerm`*.

### input_load_l

`input_load_l(`*`instTerm`*`)`

Returns the value of the 0 state INPUT_LOAD property on *`instTerm`*.

### inst

`inst(`*`instTerm`*`)`

Returns the instance to which *`instTerm`* belongs.

### matchHierProperty

`matchHierProperty(`*`instTerm`*`, `*`property`*`, `*`value`*`)`

Returns *`instTerm`* if *`property`* with value *`value`* is found on (in order) *`instTerm`*, pin, instance to which *`instTerm`* is attached, and the body of the instance. If the predicate finds multiple occurrences of *`property`*, it returns a list of all values, and a message with the property names and instance terminal name is printed in the `cp.lst` file. If property with value *`value`* does not exist on any of these four elements, it returns NULL.

### matchName

`matchName(`*`instTerm`*`, `*`string`*`)`

Returns the instTerm object if the schematic or hierarchical name of *`instTerm`* matches *`string`*.

### matchNHierProperty

`matchNHierProperty(`*`instTerm`*`, `*`property`*`, `*`value`*`)`

Returns *`instTerm`* if *`property`* with a value that contains the substring *`value`* is found on (in order) the instance terminal, pin, instance to which *`instTerm`* is attached, or the body of the instance. If *`property`* with value *`value`* is not attached to any of these objects, it returns NULL.

### matchNoName

`matchNoName(`*`instTerm`*`, `*`string`*`)`

Returns the instTerm object if the schematic name or the hierarchical name of *`instTerm`* name does not match *`string`*.

### matchNoNHier Property

`matchNoNHierProperty(`*`instTerm`*`, `*`property`*`, `*`value`*`)`

Returns *`instTerm`* if *`property`* with value that contains substring *`value`* is not attached to any one of the following: *`instTerm`*, pin, instance, or body. Otherwise, it returns NULL.

### name

`name(`*`instTerm`*`)`

Returns the canonical name of *`instTerm`*.

### outOrInOut

`outOrInOut(`*`instTerm`*`)`

Case *`pin_dir_check`* option is ON (default):

Returns the instTerm if OUTPUT_LOAD or BIDIRECTIONAL property is there on the instTerm.

Case *`pin_dir_check`* option is OFF:

Returns the instTerm if OUTPUT_LOAD or BIDIRECTIONAL or both INPUT_LOAD and OUTPUT_LOAD property is present.

### output

`output(`*`instTerm`*`)`

Case *`pin_dir_check`* option is ON (default):

Returns the instTerm if OUTPUT_LOAD property is there on the *`instTerm`*.

Case *`pin_dir_check`* option is OFF:

Returns the instTerm if OUTPUT_LOAD property is there on the *`instTerm`*.

### output_load_h

`output_load_h(`*`instTerm`*`)`

Returns the value of the 1 state OUTPUT_LOAD property on *`instTerm`*.

### output_load_l

`output_load_l(`*`instTerm`*`)`

Returns the value of the 0 state OUTPUT_LOAD property on *`instTerm`*.

### printProp

`printProp(`*`instTerm`*`, `*`property`*`)`

Prints the logical part name and the pin name together with the name and value of *`property`*. This predicate is used for printing purposes in the message only.

### sig

`sig(`*`instTerm`*`)`

Returns the signal to which *`instTerm`* is connected.

### term

`term(`*`instTerm`*`)`

Returns the terminals connected to the signal to which *`instTerm`* is connected. Equivalent to *`term(sig(instTerm))`* but more efficient.

### unused

`unused(`*`instTerm`*`)`

Returns *`instTerm`* if it is not connected to any other instTerms.

### used

`used(`*`instTerm`*`)`

Returns *`instTerm`* if it is connected to any other instTerm.

## Object Body Predicates

This section describes the object body predicates available in the logical environment. These predicates operate on each unique body type in the design. The predicates are listed in alphabetical order.

### getAllPropNames

`getAllPropNames(`*`body`*`)`

Returns a list of all properties attached to *`body`*.

### getDuplicateProps

`getDuplicateProps(`*`body`*`)`

Returns a list of properties that have more than one occurrence on *`body`*.

### getProperty

`getProperty(`*`body`*`, `*`property`*`)`

Returns the value of *`property`*, if it exists on *`body`*. If there are multiple occurrences of *`property`* on *`body`*, the predicate returns a list of all values, and a message with the property names and body name is printed in the `cp.lst` file.Otherwise, it returns NULL.

### getPropertyFloat

`getPropertyFloat(`*`body`*`, `*`property`*`)`

Returns the floating point value of *`property`*, if it exists on *`body`*. If there are multiple occurrences of *`property`* on *`body`*, the predicate returns a list of all floating point values and a message with the property names and body name is printed in the `cp.lst` file.Otherwise, it returns NULL.

### getPropertyInt

`getPropertyInt(`*`body`*`, `*`property`*`)`

Returns the integer value of *`property`*, if it exists on *`body`*. If there are multiple occurrences of *`property`* on *`body`*, the predicate returns a list of all integer values and a message with the property names and body name is printed in the `cp.lst` file.Otherwise, it returns NULL.

### getPropertyFloatWithWarn

`getPropertyFloatWithWarn(`*`body`*`, `*`property`*`)`

Returns the floating point value of *`property`* if it exists on *`body`*. If the predicate finds multiple occurrences of *`property`*, it returns a list of all floating point values and a message with the property names and body name is printed in the `cp.lst` file. Otherwise, it returns NULL and prints an error message in the `cp.lst` file.

### getPropertyIntWithWarn

`getPropertyIntWithWarn(`*`body`*`, `*`property`*`)`

Returns the integer value of *`property`* if it exists on *`body`*. If the predicate finds multiple occurrences of *`property`*, it returns a list of all integer values and a message with the property names and body name is printed in the `cp.lst` file. Otherwise, it returns NULL and prints an error message in the `cp.lst` file.

### getPropertyWithWarn

`getPropertyWithWarn(`*`body`*`, `*`property`*`)`

Returns the value of *`property`* if it exists on *`body`*. If the predicate finds multiple occurrences of *`property`*, it returns a list of all values and a message with the property names and body name is printed in the `cp.lst` file. Otherwise, it returns NULL and prints an error message in the `cp.lst` file.

### hasProperty

`hasProperty(`*`body`*`, `*`property`*`)`

Returns the body object if *`property`* exists on *`body`*. Otherwise, it returns NULL.

### hasPropertywithWarn

`hasPropertywithWarn(`*`body`*`, `*`property`*`)`

Returns the body object if *`property`* exists on *`body`*. Otherwise, it returns NULL and prints an error in the `cp.lst` file.

### inst

`inst(`*`body`*`)`

Returns the instance list of *body*.

### libName

`libName(`*body*`)`

Returns the name of the library to which *body* belongs.

### matchName

`matchName(`*body*`, `*String*`)`

Returns the body object if its schematic or hierarchical name matches *string*.

### matchNoName

`matchNoName(`*body*`, `*string*`)`

Returns the body object if its schematic or hierarchical name does not match *string*.

### matchProperty

`matchProperty(`*body*`, `*property*`, `*value*`)`

Returns the body object if it has a *property* property with value *value*. If there are multiple occurrences of *property* on the body, the predicate returns a list of all values and a message with the property names and body name is printed in the `cp.lst` file.

### name

`name(`*body*`)`

Returns the name of *body*.

## Miscellaneous Predicates

The following predicates are available in the Logical environment.

### findSig

`findSig(`*`string`*`)`

Finds a list of all signals whose synonym name matches *`string`*. Note that this can be a lengthy process for large designs.

### getVolt

`getVolt(`*`string`*`, `*`string`*`, `*`string`*`)`

Returns the component voltage corresponding to the technology passed as first argument and the specified power signal specified as second argument from the component voltage information which is passed as the third argument and is defined in the DEVICE_VOLT parameter in `cp_global.h`

# Physical Environment Predicates

This section describes the predicates available for creating rules in the physical environment.

The physical environment uses the following primary objects:

- body is a component or part type in a library
- design is the complete packaged netlist all pages)
- inst is an instance of a body in the drawing
- instTerm is a pin attached to an instance of a body
- packinst is a packaged instance in the design
- pin is a pin attached to a packaged instance
- sig is a signal corresponding to a logical signal in the design (note that signals can be composed of multiple segments and wires)
- term is a pin attached to a body (not an instance) used in the design.

## Object Body Predicates

This section describes the object body predicates available in the physical environment. These predicates operate on each body type in the packaged design. The predicates are listed in alphabetical order.

### getAllPropNames

`getAllPropNames(`*`body`*`)`

Returns a list of all properties attached to *`body`*.

### getDuplicateProps

`getDuplicateProps(`*`body`*`)`

Returns a list of properties that have more than one occurrence on *`body`*.

### getProperty

`getProperty(`*`body, property`*`)`

Returns the value of *`property`*, if it exists on *`body`*. If there are multiple occurrences of *`property`* on *`body`*, the predicate returns a list of all values, and a message with the property names and body name is printed in the `cp.lst` file.Otherwise, it returns NULL.

### getPropertyFloat

`getPropertyFloat(`*`body, property`*`)`

Returns the floating point value of *`property`*, if it exists on *`body`*. If there are multiple occurrences of *`property`* on *`body`*, the predicate returns a list of all floating-point values and a message with the property names and body name is printed in the `cp.lst` file. Otherwise, it returns NULL.

### getPropertyInt

`getPropertyInt(`*`body, property`*`)`

Returns the integer value of *`property`*, if it exists on *`body`*. If there are multiple occurrences of *`property`* on *`body`*, the predicate returns a list of all integer values, and a message with the property names and body name is printed in the `cp.lst` file.Otherwise, it returns NULL.

### getPropertyFloatWithWarn

`getPropertyFloatWithWarn(`*`body`*`, `*`property`*`)`

Returns the floating point value of *`property`* if it exists on *`body`*. If the predicate finds multiple occurrences of *`property`*, it returns a list of all floating point values and a message with the property names and body name is printed in the `cp.lst` file. Otherwise, it returns NULL and prints an error message in the `cp.lst` file.

### getPropertyIntWithWarn

`getPropertyIntWithWarn(`*`body`*`, `*`property`*`)`

Returns the integer value of *`property`* if it exists on *`body`*. If the predicate finds multiple occurrences of *`property`*, it returns a list of all integer values and a message with the property names and body name is printed in the `cp.lst` file. Otherwise, it returns NULL and prints an error message in the `cp.lst` file.

### getPropertyWithWarn

`getPropertyWithWarn(`*`body`*`, `*`property`*`)`

Returns the value of *`property`* if it exists on *`body`*. If the predicate finds multiple occurrences of *`property`*, it returns a list of all values and a message with the property names and body name is printed in the `cp.lst` file. Otherwise, it returns NULL and prints an error message in the `cp.lst` file.

### hasNoProperty

`hasNoProperty(`*`body`*`, `*`property`*`)`

Returns the body object if *`property`* does not exist on *`body`*. Otherwise, it returns NULL.

### hasProperty

`hasProperty(`*`body`*`, `*`property`*`)`

Returns the body object if *`property`* exists on *`body`*. Otherwise, it returns NULL.

### hasPropertywithWarn

`hasPropertywithWarn(`*`body`*`, `*`property`*`)`

Returns the body object if *property* exists on *body*. Otherwise, it returns NULL and prints an error in the `cp.lst` file.

### inst

`inst(`*body*`)`

Returns the instance list of *body*.

### libName

`libName(`*body*`)`

Returns the name of the library to which *body* belongs.

### matchName

`matchName(`*body*`, `*String*`)`

Returns the body object if its schematic name or hierarchical name matches *string*.

### matchNoName

`matchNoName(`*body*`, `*string*`)`

Returns the body object if its schematic name or hierarchical name does not match *string*.

### matchProperty

`matchProperty(`*body*`, `*property*`, `*value*`)`

Returns the body object if it has a *property* property with *value* value. If there are multiple occurrences of *property* on the body, the predicate returns a list of all values, and a message with the property names and body name is printed in the `cp.lst` file.

### name

`name(`*body*`)`

Returns the name of *body*.

### packinst

`packinst(`*`body`*`)`

Returns a list of packaged instances of *`body`*.

### pin

`pin(`*`body`*`)`

Returns a list of pins of *`body`*.

### term

`term(`*`body`*`)`

Returns a list of terminals of *`body`*.

## Object Design Predicates

This section describes the object design predicates available in the physical environment. These predicates operate on the entire packaged design. The predicates are listed in alphabetical order.

### body

`body(`*`design`*`)`

Returns a list of all bodies in *`design`*.

### getAllPropNames

`getAllPropNames(`*`design`*`)`

Returns a list of all properties attached to *`design`*.

### getDuplicateProps

`getDuplicateProps(`*`design`*`)`

Returns a list of properties that have more than one occurrence on *`design`*.

### getProperty

`getProperty(`*`design`*`, `*`property`*`)`

Returns the value of *`property`* if *`property`* exists on the design. If there are multiple occurrences of *`property`* in the design, the predicate returns a list of all values, and a message with the property names and design name is printed in the `cp.lst` file. Otherwise it returns NULL.

### getPropertyFloat

`getPropertyFloat(`*`design`*`, `*`property`*`)`

Returns the floating point value of *`property`* if *`property`* exists on the design. If there are multiple occurrences of *`property`* in the design, the predicate returns a list of all floating point values, and a message with the property names and design name is printed in the `cp.lst` file. Otherwise it returns NULL.

### getPropertyInt

`getPropertyInt(`*`design`*`, `*`property`*`)`

Returns the integer value of *`property`* if *`property`* exists on the design. If there are multiple occurrences of *`property`* in the design, the predicate returns a list of all integer values, and a message with the property names and design name is printed in the `cp.lst` file. Otherwise it returns NULL.

### getPropertyFloatWith Warn

`getPropertyFloatWithWarn(`*`design`*`, `*`property`*`)`

Returns the floating point value of *`property`* if *`property`* exists on the design. If the predicate finds multiple occurrences of *`property`*, it returns a list of all values and a message with the property names and design name is printed in the `cp.lst` file. Otherwise it returns NULL and prints a message in the `cp.lst` file.

### getPropertyIntWithWarn

`getPropertyIntWithWarn(`*`design`*`, `*`property`*`)`

Returns the integer value of *`property`* if *`property`* exists on the design. If the predicate finds multiple occurrences of *`property`*, it returns a list of all integer values and a message with the property names and design name is printed in the `cp.lst` file. Otherwise it returns NULL and prints a message in the `cp.lst` file.

### getPropertytWithWarn

`getPropertyWithWarn(`*`design`*`, `*`property`*`)`

Returns the value of *`property`* if *`property`* exists on the design. If the predicate finds multiple occurrences of *`property`*, it returns a list of all values and a message with the property names and design name is printed in the `cp.lst` file. Otherwise it returns NULL and prints a message in the `cp.lst` file.

### hasNoProperty

`hasNoProperty(`*`design`*`, `*`property`*`)`

Returns the design object if *`property`* does not exist on it. Otherwise it returns NULL.

### hasProperty

`hasProperty(`*`design`*`, `*`property`*`)`

Returns the design object if *`property`* exists on it. Otherwise, it returns NULL.

### hasPropertyWithWarn

`hasPropertyWithWarn(`*`design`*`, `*`property`*`)`

Returns the design object if *`property`* exists on it. Otherwise it returns NULL and prints a warning message in the `cp.lst` file.

### inst

`inst(`*`design`*`)`

Returns a list of all instances in *`design`*.

### instTerm

`instTerm(`*`design`*`)`

Returns a list of all instance terminals in *`design`*.

### matchProperty

`matchProperty(`*`design`*`, `*`property`*`, `*`value`*`)`

Returns the design object if *`property`* has a value of *`value`*. If there are multiple occurrences of *`property`* in the design, the predicate returns a list of all values and a message with the property names and design name is printed in the `cp.lst` file. Otherwise, it returns NULL.

### name

`name(`*`design`*`)`

Returns the name of *`design`*.

### packinst

`packinst(`*`design`*`)`

Returns a list of all packaged instances in *design*.

### pin

`pin(`*design*`)`

Returns a list of all pins of *design*.

### sig

`sig(`*design*`)`

Returns a list of all nets (signals) in *design*.

### term

`term(`*design*`)`

Returns a list of all terminals of *design*.

### ToolstartFail

`toolstartFail(`*design,call_string*`)`

Returns the design on which tool invoked in *call_string* fails, that is the exit status is 1.

### toolstartIgnore

`toolstartignore(`*design,call_string*`)`

Returns the design ignoring return-status of tool invoked in *call_string*.

### toolstartPass

`toolstartPass(`*design,call_string*`)`

Returns the design on which tool invoked in *call_string* passes, that is the exit status is 0.

## Object Instance Predicates

This section describes the object instance predicates available in the physical environment. These predicates operate on all logical instances in the packaged design. The predicates are listed in alphabetical order.

### body

`body(`*`inst`*`)`

Returns the body or part type (whichever is appropriate) of *`inst`*.

### convert

`convert(`*`inst`*`)`

Returns a string containing information about the location of the instance in the drawing

### getAllPropNames

`getAllPropNames(`*`inst`*`)`

Returns a list of all properties attached to *`inst`*.

### getDuplicateProps

`getDuplicateProps(`*`inst`*`)`

Returns a list of properties that have more than one occurrence on *`inst`*.

### getHierPropertyFloat

`getHierPropertyFloat(`*`inst`*`, `*`property`*`)`

Returns the floating point value of *`property`* if *`property`* is attached to *`inst`*; if not, it returns the value of *`property`* if *`property`* is attached to the part type of *`inst`*. If the predicate finds multiple occurrences of *`property`*, it returns a list of all floating point values and a message with the property names and instance name is printed in the `cp.lst` file. If *`property`* is not attached to either of these objects, it returns NULL.

### getHierProperty

`getHierProperty(`*`inst`*`, `*`property`*`)`

Returns the integer value of *property* if *property* is attached to *inst*; if not, it returns the integer value of *property* if *property* is attached to the part type of *inst*. If the predicate finds multiple occurrences of *property*, it returns a list of all integer values, and a message with the property names and instance name is printed in the `cp.lst` file. If *property* is not attached to either of these objects, it returns NULL.

### getHierPropertyFloatWithWarn

`getHierPropertyFloatWithWarn(`*inst*`, `*property*`)`

Returns the floating point value of *property* if *property* is attached to *inst*; if not, it returns the floating point value of *property* if *property* is attached to the part type of *inst*. If the predicate finds multiple occurrences of *property*, it returns a list of all floating point values, and a message with the property names and instance name is printed in the `cp.lst` file. If *property* is not attached to either of these objects, it returns NULL and prints a message in the `cp.lst` file.

### getHierPropertyIntWithWarn

`getHierPropertyWithWarn(`*inst*`, `*property*`)`

Returns the integer value of *property* if *property* is attached to *inst*; if not, it returns the integer value of *property* if *property* is attached to the part type of *inst*. If the predicate finds multiple occurrences of *property*, it returns a list of all integer values and a message with the property names and instance name is printed in the `cp.lst` file. If *property* is not attached to either of these objects, it returns NULL and prints a message in the `cp.lst` file.

### getHierPropertyFloatWithWarn

`getHierPropertyFloatWithWarn(`*inst*`, `*property*`)`

Returns the floating-point value of *property* if *property* is attached to *inst*; if not, it returns the floating-point value of *property* if *property* is attached to the part type of *inst*. If the predicate finds multiple occurrences of *property*, it returns a list of all floating-point values and a message with the property names and instance name is printed in the `cp.lst` file. If *property* is not attached to either of these objects, it returns NULL and prints a message in the `cp.lst` file.

### getHierPropertyIntWithWarn

`getHierPropertyIntWithWarn(`*inst*`, `*property*`)`

Returns the integer value of *property* if *property* is attached to *inst*; if not, it returns the integer value of *property* if *property* is attached to the part type of *inst*. If the predicate finds multiple occurrences of *property*, it returns a list of all integer values and a message with the property names and instance name is printed in the `cp.lst` file. If *property* is not attached to either of these objects, it returns NULL and prints a message in the `cp.lst` file.

### getHierPropertyWithWarn

`getHierPropertyWithWarn(`*inst*`, `*property*`)`

Returns the value of *property* if *property* is attached to *inst*; if not, it returns the value of *property* if *property* is attached to the part type of *inst*. If the predicate finds multiple occurrences of *property*, it returns a list of all values and a message with the property names and instance name is printed in the `cp.lst` file. If *property* is not attached to either of these objects, it returns NULL and prints a message in the `cp.lst` file.

### hasHierProperty

`hasHierProperty(`*inst*`, `*property*`)`

Returns the instance object if *property* is attached to *inst*; if not, it returns the instance object if *property* is attached to the part type of *inst*. If *property* is not attached to any of these objects, it returns NULL.

### hasHierProperty WithWarn

`hasHierPropertyWithWarn(`*inst*`, `*property*`)`

Returns the instance object if *property* is attached to *inst*; if not, it returns the instance object if *property* is attached to the part type of *inst*. If *property* is not attached to either of these objects, it returns NULL and prints a message in the `cp.lst` file.

### hasNoHierProperty

`hasNoHierProperty(`*inst*`, `*property*`)`

Returns the instance object if *property* is not attached to *inst* or part type of inst.

### hasNoBody

`hasNoBody(`*inst*`)`

Returns the instance object if the *inst* does not have a body. Mainly used to check for flag bodies.

### hasNoHierProperty

`hasNoHierProperty(`*inst*`, `*property*`)`

Returns the instance object if *property* is not attached to *inst*. If it is attached, it returns the instance object if *property* is not attached to the part type of *inst*. If *property* is attached to either of these objects, it returns NULL.

### instTerm

`instTerm(`*inst*`)`

Returns a list of all pins on *inst*.

### matchHierProperty

`matchHierProperty(`*inst*`, `*property*`, `*value*`)`

Returns the instance object if *property* with value *value* is attached to *inst*; if not, it returns the instance object if *property* with value *value* is attached to the part type of *inst*. If the predicate finds multiple occurrences of *property*, it returns a list of all values and a message with the property names and instance name is printed in the `cp.lst` file. If *property* with value *value* is not attached to either of these objects, it returns NULL.

### matchName

`matchName(`*inst*`, `*string*`)`

Returns the instance object if its schematic name or its hierarchical name matches *string*. Otherwise, it returns NULL.

### matchNHierProperty

`matchNHierProperty(`*inst*`, `*property*`, `*value*`)`

Returns *inst* if *property* with a value that contains substring *value* is found on (in order) the instance or the body of the instance. If *property* with value *value* is not attached to either of these objects, it returns NULL.

### matchNoHierProperty

`matchNoHierProperty(`*`inst`*`, `*`property`*`, `*`value`*`)`

Returns the instance object if it does not have a *`property`* property with value *`value`*.

### matchNoName

`matchNoName(`*`inst`*`, `*`string`*`)`

Returns the instance object if its schematic name or hierarchical name does not match *`string`*. Otherwise, else it returns NULL.

### name

`name(`*`inst`*`)`

Returns the name of *`inst`*.

### packinst

`packinst(`*`inst`*`)`

Returns the packaged instance to which *`inst`* belongs.

### toolstartFail

`toolstartFail(`*`inst,call_string`*`)`

Returns the instance on which tool invoked in *`call_string`* fails, that is the exit status is 1.

### toolstartIgnore

`toolstartignore(`*`inst,call_string`*`)`

Returns the instance ignoring return-status of tool invoked in *`call_string`*.

### toolstartPass

`toolstartPass(`*`inst,call_string)`*

Returns the instance on which tool invoked in *`call_string`* passes, that is the exit status is 0

## sig

`sig(`*`inst`*`)`

Returns a list of all signals (nets) connected to the *`inst`* through its pins. Equivalent to *`sig(instTerm(inst))`* but more efficient.

## Object Instance Terminal Predicates

This section describes the object instance terminal predicates available in the physical environment. These predicates operate on all instTerm attached to instances in the design. The predicates are listed in alphabetical order.

### connected

`connected(`*`instTerm`*`)`

Returns a list of instTerm connected to *`instTerm`*.

### convert

`convert(`*`instTerm`*`)`

Returns a string that contains information about the page number of the drawing containing the instance of the *`instTerm.`*

### getAllPropNames

`getAllPropNames(`*`instTerm`*`)`

Returns a list of all properties attached to *`instTerm`*.

### getCap

`getCap(`*`instTerm, property`*`)`

Returns the INPUT_CAP or the OUTPUT_CAP value of *`instTerm`*, according to the value (INPUT_CAP or OUTPUT_CAP) of the second parameter.

### getDeviceVolt

`getDeviceVolt(`*`instTerm, value`*`)`

Returns the device voltage corresponding to the technology of *`instTerm`* and the specified *`value`* (VCC, VEE, and so on). Device voltage information is defined in the DEVICE_VOLT macro within *`cp_config.h`*.

### getDuplicateProps

`getDuplicateProps(`*`instTerm`*`)`

Returns a list of properties that have more than one occurrence on *instTerm*.

### getHierProperty

`getHierProperty(`*instTerm*`, `*property*`)`

Returns the first value of *property* found on (in order) *instTerm*, pin, instance to which *instTerm* is attached, and the body of the instance. If the predicate finds multiple occurrences of *property*, it returns a list of all values, and a message with the property names and instance terminal name is printed in the `cp.lst` file. If property does not exist on any of these four elements, it returns NULL.

### getHierPropertyFloat

`getHierPropertyFloat(`*instTerm*`, `*property*`)`

Returns the first floating-point value of *property* found on (in order) *instTerm*, pin, instance to which *instTerm* is attached, and the body of the instance. If the predicate finds multiple occurrences of *property*, it returns a list of all floating-point values and a message with the property names and instance terminal name is printed in the `cp.lst` file. If property does not exist on any of these four elements, it returns NULL.

### getHierPropertyInt

`getHierPropertyInt(`*instTerm*`, `*property*`)`

Returns the first integer value of *property* found on (in order) *instTerm*, pin, instance to which *instTerm* is attached, and the body of the instance. If the predicate finds multiple occurrences of *property*, it returns a list of all integer values and a message with the property names and instance terminal name is printed in the `cp.lst` file. If property does not exist on any of these four elements, it returns NULL.

### getHierPropertyFloatWithWarn

`getHierPropertyFloatWithWarn(`*instTerm*`, `*property*`)`

Returns the first floating-point value of *property* found on (in order) *instTerm*, pin, instance to which *instTerm* is attached, and the body of the instance. If the predicate finds multiple occurrences of *property*, it returns a list of all floating-point values and a message with the property names and instance terminal name is printed in the `cp.lst` file. If property does not exist on any of these four elements, it returns NULL and prints an error message in the `cp.lst` file.

### getHierPropertyIntWithWarn

`getHierPropertyIntWithWarn(`*`instTerm`*`, `*`property`*`)`

Returns the first integer value of *`property`* found on (in order) *`instTerm`*, pin, instance to which *`instTerm`* is attached, and the body of the instance. If the predicate finds multiple occurrences of *`property`*, it returns a list of all integer values and a message with the property names and instance terminal name is printed in the `cp.lst` file. If property does not exist on any of these four elements, it returns NULL and prints an error message in the `cp.lst` file.

### getHierPropertyWithWarn

`getHierPropertyWithWarn(`*`instTerm`*`, `*`property`*`)`

Returns the first value of *`property`* found on (in order) *`instTerm`*, pin, instance to which *`instTerm`* is attached, and the body of the instance. If the predicate finds multiple occurrences of *`property`*, it returns a list of all values, and a message with the property names and instance terminal name is printed in the `cp.lst` file. If property does not exist on any of these four elements, it returns NULL and prints an error message in the `cp.lst` file.

### hasHierProperty

`hasHierProperty(`*`instTerm`*`, `*`property`*`)`

Returns *`instTerm`* if *`property`* is attached to any one of the following: *`instTerm`*, pin, instance, or body. Otherwise, it returns NULL.

### hasHierPropertyWithWarn

`hasHierPropertyWithWarn(`*`instTerm`*`, `*`property`*`)`

Returns *`instTerm`* if *`property`* is attached to any one of the following: *`instTerm`*, pin, instance, or body. Otherwise, it returns NULL and prints an error message in the `cp.lst` file.

### hasNoHierName

`hasNoHierName(`*`instTerm`*`, `*`value`*`)`

Returns *`instTerm`* if *`value`* is a substring of the *instTerm* or *instance* name.

### hasNoHierProperty

`hasNoHierProperty(`*`instTerm`*`, `*`property`*`)`

Returns *`instTerm`* if *`property`* is not attached to any one of the following: *`instTerm`*, pin, instance, or body. Otherwise, it returns NULL.

### inOrInOut

`inOrInOut(`*`instTerm`*`)`

Case *`pin_dir_check`* option is ON (default):

Returns the instTerm if INPUT_LOAD or BIDIRECTIONAL property is there on the instTerm.

Case *`pin_dir_check`* option is OFF:

Returns the instTerm if none of the properties (BIDIRECTIONAL, INPUT_LOAD,OUTPUT_LOAD) is present or INPUT_LOAD or BIDIRECTIONAL or both INPUT_LOAD and OUTPUT_LOAD property is present.

### inout

`inout(`*`instTerm`*`)`

Case *`pin_dir_check`* option is ON (default):

Returns the instTerm if BIDIRECTIONAL property is there on the instTerm.

Case *`pin_dir_check`* option is OFF:

Returns the instTerm if BIDIRECTIONAL or INPUT_LOAD and OUTPUT_LOAD property is present.

### input

`input(`*`instTerm`*`)`

Case *`pin_dir_check`* option is ON (default):

Returns the *`instTerm`* if INPUT_LOAD property is there on the instTerm.

Case *`pin_dir_check`* option is OFF:

Returns the instTerm if none of the properties (BIDIRECTIONAL, INPUT_LOAD,OUTPUT_LOAD) is present or if INPUT_LOAD property is only present.

### input_load_h

`input_load_h(`*`instTerm`*`)`

Returns the value of the 1 state INPUT_LOAD property on *`instTerm`*.

### input_load_l

`input_load_l(`*`instTerm`*`)`

Returns the value of the 0 state INPUT_LOAD property on *`instTerm`*.

### inst

`inst(`*`instTerm`*`)`

Returns the instance to which *`instTerm`* belongs.

### instHasBody

`instHasBody(`*`instTerm`*`)`

Returns *`instTerm`* if the instance it is connected to does not have a body.

### matchHierProperty

`matchHierProperty(`*`instTerm`*`, `*`property`*`, `*`value`*`)`

Returns *`instTerm`* if *`property`* with value *`value`* is found on (in order) *`instTerm`*, pin, instance to which *`instTerm`* is attached, and the body of the instance. If the predicate finds multiple occurrences of *`property`*, it returns a list of all values and a message with the property names and instance terminal name is printed in the `cp.lst` file. If property with value *`value`* does not exist on any of these four elements, it returns NULL.

### matchNHierProperty

`matchNHierProperty(`*`instTerm`*`, `*`property`*`, `*`value`*`)`

Returns *`instTerm`* if *`property`* with a value that contains the substring *`value`* is found on (in order) the instance terminal, pin, instance to which *`instTerm`* is attached, or the body

of the instance. If *property* with value *value* is not attached to any of these objects, it returns NULL.

### matchName

`matchName(`*instTerm*`, `*string*`)`

Returns the instTerm object if its schematic name or hierarchical name matches *string*.

### matchNoName

`matchNoName(`*instTerm*`, `*string*`)`

Returns the instTerm object if its schematic name or its hierarchical name does not match *string*.

### matchNoNHierProperty

`matchNoNHierProperty(`*instTerm*`, `*property*`, `*value*`)`

Returns *instTerm* if *property* with value that contains substring *value* is not attached to any one of the following: *instTerm*, pin, instance, or body. Otherwise, it returns NULL.

### name

`name(`*instTerm*`)`

Returns the canonical name of *instTerm*.

### outOrInOut

`outOrInOut(`*instTerm*`)`

Case *pin_dir_check* option is ON (default):

Returns the instTerm if OUTPUT_LOAD or BIDIRECTIONAL property is there on the instTerm.

Case *pin_dir_check* option is OFF:

Returns the instTerm if OUTPUT_LOAD or BIDIRECTIONAL or both INPUT_LOAD and OUTPUT_LOAD property is present.

### output

`output(`*`instTerm`*`)`

Case *`pin_dir_check`* option is ON (default):

Returns the *`instTerm`* if OUTPUT_LOAD property is there on the instTerm.

Case *`pin_dir_check`* option is OFF:

Returns the instTerm if OUTPUT_LOAD property is there on the instTerm.

### output_load_h

`output_load_h(`*`instTerm`*`)`

Returns the value of the 1 state OUTPUT_LOAD property on *`instTerm`*.

### output_load_l

`output_load_l(`*`instTerm`*`)`

Returns the value of the 0 state OUTPUT_LOAD property on *`instTerm`*.

### pin

`pin(`*`instTerm`*`)`

Returns the pin that is connected to *`instTerm`*.

### printProp

`printProp(`*`instTerm`*`, `*`property`*`)`

Prints the logical part name and the pin name together with the name and value of *`property`*. This predicate is used for printing purposes in the message only.

### sig

`sig(`*`instTerm`*`)`

Returns the signal to which *`instTerm`* is connected.

### term

`term(`*`instTerm`*`)`

Returns the terminals connected to the signal to which *`instTerm`* is connected. Equivalent to *`term(sig(instTerm))`* but more efficient.

### unused

`unused(`*`instTerm`*`)`

Returns *`instTerm`* if it is not connected to any other instTerms.

### used

`used(`*`instTerm`*`)`

Returns *`instTerm`* if it is connected to any other instTerms.

## Object Packaged Instance Predicates

This section describes the object packaged instance predicates available in the physical environment. These predicates operate on packaged instances in the design. The predicates are listed in alphabetical order.

### body

`body(`*`packinst`*`)`

Returns the part type of the *`packinst`*.

### inst

`inst(`*`packinst`*`)`

Returns a list of instances in *`packinst`*.

### instTerm

`instTerm(`*`packinst`*`)`

Returns a list of instance terminals in *`packinst`*.

### matchName

`matchName(`*`packinst, string`*`)`

Returns *`packinst`* if its schematic name or hierarchical name matches *`string`*.

### matchNoName

`matchName(`*`packinst, string`*`)`

Returns *`packinst`* if its schematic name or hierarchical name does not match *`string`*.

### name

`name(`*`packinst`*`)`

Returns the name of the packaged instance *`packinst`*.

### power

`power(packinst, property)`

Determines the value of *property* from the *chips.prt* file, then multiplies it by (*used_sections*/*total_sections*) to determine the power requirements of *packinst*.

### totalSections

`totalSections(packinst)`

Returns (as an integer) the total number of sections in *packinst*.

### usedSections

`usedSections(packinst)`

Returns (as an integer) the number of used sections in *packinst*.

## Object Pin Predicates

This section describes the object pin predicates available in the physical environment. These predicates operate on pins of packaged instances in the design. The predicates are listed in alphabetical order.

### matchName

`matchName(`*`pin, string`*`)`

Returns the pin object if its schematic name or its hierarchical name matches *`string.`* Otherwise, it returns NULL.

### matchNoName

`matchNoName(`*`pin, string`*`)`

Returns the pin object if its schematic name or its hierarchical name does not match *`string.`* Otherwise, it returns NULL.

### name

`name(`*`pin`*`)`

Returns the name of *`pin`*.

### section

`section(`*`pin`*`)`

Returns the section of the packaged instance where *`pin`* is located.

### term

`term(`*`pin`*`)`

Returns the terminal associated with *`pin`*.

## Object Signal Predicates

This section describes the object signal predicates available in the physical environment. These predicates operate on signals (nets) in the design. The predicates are listed in alphabetical order.

### instTerm

`instTerm(sig)`

Returns the instance pins connected to the given signal.

### getAllPropNames

`getAllPropNames(`*`sig`*`)`

Returns a list of all properties attached to *`sig`*.

### getDuplicateProps

`getDuplicateProps(`*`sig`*`)`

Returns a list of properties that have more than one occurrence on *`sig`*.

### getProperty

`getProperty(`*`sig, property`*`)`

Returns the value of the *`property`* if it exists on *`sig`*. If there are multiple occurrences of *`property`* on *`sig`*, the predicate returns a list of all values, and a message with the property names and signal name is printed in the `cp.lst` file.Otherwise, it returns NULL.

### getPropertyFloat

`getPropertyFloat(`*`sig, property`*`)`

Returns the floating-point value of the *`property`* if it exists on *`sig`*. If there are multiple occurrences of *`property`* on *`sig`*, the predicate returns a list of all floating-point values and a message with the property names and signal name is printed in the `cp.lst` file.Otherwise, it returns NULL.

### getPropertyInt

`getPropertyInt(`*`sig`*`, `*`property`*`)`

Returns the integer value of the *property* if it exists on *sig*. If there are multiple occurrences of *property* on *sig*, the predicate returns a list of all integer values and a message with the property names and signal name is printed in the `cp.lst` file.Otherwise, it returns NULL.

### getPropertyFloatWithWarn

`getPropertyFloatWithWarn(`*`sig`*`, `*`property`*`)`

Returns the floating-point value of property if it exists on sig. If the predicate finds multiple occurrences of *property*, it returns a list of all floating-point values and a message with the property names and signal name is printed in the `cp.lst` file. Otherwise, it returns NULL and prints a message in the `cp.lst` file.

### getPropertyIntWithWarn

`getPropertyIntWithWarn(`*`sig`*`, `*`property`*`)`

Returns the integer value of property if it exists on sig. If the predicate finds multiple occurrences of *property*, it returns a list of all integer values and a message with the property names and signal name is printed in the `cp.lst` file. Otherwise, it returns NULL and prints a message in the `cp.lst` file.

### getPropertyWithWarn

`getPropertyWithWarn(`*`sig`*`, `*`property`*`)`

Returns the value of property if it exists on sig. If the predicate finds multiple occurrences of *property*, it returns a list of all values and a message with the property names and signal name is printed in the `cp.lst` file. Otherwise, it returns NULL and prints a message in the `cp.lst` file.

### getSupplyVolt

`getSupplyVolt (`*`sig`*`, `*`value`*`)`

Returns the supply voltage corresponding to *sig* and *value* (VCC, VDD, and so on). Supply voltage information is defined in the SUPPLY_VOLT macro within *cp_config.h*.

### hasProperty

`hasProperty(sig, property)`

Returns the signal object if *property* exists on it.

### hasPropertyWithWarn

`hasPropertywithWarn(sig, property)`

Returns the signal object if *property* exists on it. Otherwise, it returns NULL and prints a message in the `cp.lst` file.

### matchName

`matchName(sig, string)`

Returns the signal object if its schematic name or its hierarchical name matches *sig*. Otherwise, it returns NULL.

### matchProperty

`matchProperty(sig, property, value)`

Returns the signal object if the *property* property with value *value* exists on *sig*. If there are multiple occurrences of *property* on the signal, the predicate returns a list of all values, and a message with the property names and signal name is printed in the `cp.lst` file.

### name

`name(sig)`

Returns the name of *sig*. This name is the canonical name used by CAE Views.

### named

`named(sig)`

Returns the signal if *sig* is named.

### NoOfPages

`NoOfPages(sig)`

Returns the number of pages that contain *sig*.

### pin

`pin(`*sig*`)`

Returns the list of pins attached to *sig*.

### unnamed

`unnamed(`*sig*`)`

Returns the signal if *sig* is unnamed.

## Object Terminal Predicates

This section describes the object terminal predicates available in the physical environment. These predicates operate on unique pin types attached to bodies used in the design. The predicates are listed in alphabetical order.

### getAllPropNames

`getAllPropNames(`*`term`*`)`

Returns a list of all properties attached to *`term`*.

### getDuplicateProps

`getDuplicateProps(`*`term`*`)`

Returns a list of properties that have more than one occurrence on *`term`*.

### getProperty

`getProperty(`*`term`*`, `*`property`*`)`

Returns the value of *`property`* if it exists on *`term`*. If there are multiple occurrences of *`property`* on the terminal, the predicate returns a list of all values and a message with the property names and terminal name is printed in the `cp.lst` file. Otherwise, it returns NULL.

### getPropertyFloat

`getProperty(`*`term`*`, `*`property`*`)`

Returns the floating-point value of *`property`* if it exists on *`term`*. If there are multiple occurrences of *`property`* on the terminal, the predicate returns a list of all floating-point values and a message with the property names and terminal name is printed in the `cp.lst` file. Otherwise, it returns NULL.

### getPropertyInt

`getPropertyInt(`*`term`*`, `*`property`*`)`

Returns the integer value of *`property`* if it exists on *`term`*. If there are multiple occurrences of *`property`* on the terminal, the predicate returns a list of all integer values and a message with the property names and terminal name is printed in the `cp.lst` file. Otherwise, it returns NULL.

### getPropertyFloatWithWarn

`getPropertyFloatWithWarn(`*`term`*`, `*`property`*`)`

Returns the floating-point value of *`property`* if it exists on *`term`*. If the predicate finds multiple occurrences of *`property`*, it returns a list of all floating-point values and a message with the property names and terminal name is printed in the `cp.lst` file. Otherwise, it returns NULL, and prints an error message in the `cp.lst` file.

### getPropertyIntWithWarn

`getPropertyIntWithWarn(`*`term`*`, `*`property`*`)`

Returns the integer value of *`property`* if it exists on *`term`*. If the predicate finds multiple occurrences of *`property`*, it returns a list of all integer values and a message with the property names and terminal name is printed in the `cp.lst` file. Otherwise, it returns NULL, and prints an error message in the `cp.lst` file.

### getPropertyWithWarn

`getPropertyWithWarn(`*`term`*`, `*`property`*`)`

Returns the value of *`property`* if it exists on *`term`*. If the predicate finds multiple occurrences of *`property`*, it returns a list of all values, and a message with the property names and terminal name is printed in the `cp.lst` file. Otherwise, it returns NULL, and prints an error message in the `cp.lst` file.

### hasProperty

`hasProperty(`*`term`*`, `*`property`*`)`

Returns the terminal object if *`property`* exists on *`term`*. Otherwise, it returns NULL.

### hasPropertyWithWarn

`hasPropertyWithWarn(`*`term`*`, `*`property`*`)`

Returns the terminal object if *`property`* exists on *`term`*. Otherwise, it returns NULL and prints an error in the `cp.lst` file.

### inOrInOut

`inOrInOut(`*`term`*`)`

Case *pin_dir_check* option is ON (default):

Returns the term if INPUT_LOAD or BIDIRECTIONAL property is there on the term.

Case *pin_dir_check* option is OFF:

Returns the term if none of the properties (BIDIRECTIONAL, INPUT_LOAD,OUTPUT_LOAD) is present or INPUT_LOAD or BIDIRECTIONAL or both INPUT_LOAD and OUTPUT_LOAD property is present.

### inout

`inout(term)`

Case *pin_dir_check* option is ON (default):

Returns the term if BIDIRECTIONAL property is there on the term.

Case *pin_dir_check* option is OFF:

Returns the term if BIDIRECTIONAL or INPUT_LOAD and OUTPUT_LOAD property is present.

### input

`input(term)`

Case *pin_dir_check* option is ON (default):

Returns the term if INPUT_LOAD property is there on the term.

Case *pin_dir_check* option is OFF:

Returns the term if none of the properties (BIDIRECTIONAL, INPUT_LOAD,OUTPUT_LOAD) is present or if INPUT_LOAD property is only present.

### matchName

`matchName(term, string)`

Returns the terminal object its schematic name or its hierarchical name matches *string.* Otherwise, it returns NULL.

### matchNoName

`matchNoName(term, string)`

Returns the terminal object if its schematic name or its hierarchical name do not match *string*. Otherwise, it returns NULL.

### matchProperty

`matchProperty(term, property, value)`

Returns the terminal object if *property* has a value of *value*. If there are multiple occurrences of *property* on the terminal, the predicate returns a list of all values, and a message with the property names and terminal name is printed in the `cp.lst` file. Otherwise, it returns NULL.

### name

`name(term)`

Returns the name of *term*.

### outOrInOut

`outOrInOut(term)`

Case *pin_dir_check* option is ON (default):

Returns the term if OUTPUT_LOAD or BIDIRECTIONAL property is there on the term.

Case *pin_dir_check* option is OFF:

Returns the term if OUTPUT_LOAD or BIDIRECTIONAL or both INPUT_LOAD and OUTPUT_LOAD property is present.

### output

`output(term)`

Case *pin_dir_check* option is ON (default):

Returns the term if OUTPUT_LOAD property is there on the term.

Case *pin_dir_check* option is OFF:

Returns the term if OUTPUT_LOAD property is there on the term.

### pin

`pin(`*`term`*`)`

Returns the pin that is connected to *`term`*.

## Miscellaneous Predicates

The following predicates are available in the Physical environment.

### findSig

`findSig(`*`string`*`)`

Finds a list of all signals whose synonym name matches *`string`*.

**Note:** This can be a lengthy process for large designs

# C

# Dialog Box and Menu Help

## Design Entry HDL Rules Checker

This is the main window for Rules Checker. In this window, you can do the following:

- Specify a design for running Rules Checker. For details, see Specifying a Design on page 56.
- Select rules you want to run in your design. For details, see Selecting Rules on page 57.
- Carry out other functions of Rules Checker by selecting appropriate options from the menu bar.
- Customize rule parameters and messages.

### Function

This dialog box has four window tabs:

- Body Rules tab - lets you specify the body rules you want to run on the design. Displays the list of rule files in the current path for the selected environment.
- Graphical Rules tab - lets you specify the graphical rules you want to run on the screen. Displays the list of rule files in the current path for the selected environment.
- Logical Rules tab - lets you specify the logical rules you want to run on the screen. Displays the list of rule files in the current path for the selected environment.

- Physical Rules tab - lets you specify the logical rules you want to run on the screen. Displays the list of rule files in the current path for the selected environment.

| | |
|---|---|
| **Design** | Enter the design name. Entering libname is mandatory for all. You can use the Browse button to enter the design name.<br><br>For Logical and Physical environments, enter libname.cellname. Entering cellname is also mandatory.<br><br>For Graphical environment, enter libname.cellname(viewname)page_no.<br><br>For Body environment, enter libname.cellname(viewname). |
| **Run** | Click this button to run the selected rules on your design. Brings up the Rules Checker Progress dialog box. |
| **View Markers** | Click this button to launch Markers in Allegro Design Entry HDL with the marker file, for the current run loaded. You can highlight the design violations in Allegro Design Entry HDL. If Allegro Design Entry HDL is not up, an error message is displayed. |
| **View Files** | Click this button to bring up the Choose File dialog box. You can specify the files you want to view. |

# Rules Checker Toolkit

This dialog box lets you create your own rule files.

| | |
|---|---|
| **Look in** | This browser lets you select an existing rule-file that you want to change. The selected rule appears in the *File name* field. |
| **File name** | Enter the name of the rule file you want to create or change. |
| **Edit** | Click this button to launch the text editor for creating a new rule or changing an existing rule. You can define the Editor you want to use. |
| **Compile** | Click this button to compile the rule-file you have created using the Editor. |

# View Open

Use this dialog box to open a component or design for checking.

| | |
|---|---|
| **Library** | Identifies the library whose components you want to list in the scroll box. |
| **Cell** | If you know the name of the component or design, type it in this box. |
| **View** | Specifies the Schematic (logical) or Symbol (symbolic representation) based on the view you want to open. |
| **Version** | Displays the version number of the schematic or symbol representation (the default is 1). This field does not identify drawing revision. |
| **Page** | Displays the page number of a schematic (the default is 1). |
| **Open** | Opens the specified drawing. |
| **Cancel** | Closes the View Open Dialog box. |
| **Filters>>** | Lets you narrow the list of components or designs. |
| **Cell Name** | use wildcard characters<br>* matches any text string.<br>? matches any character string. |
| **View Name** | use wildcard characters<br>* matches any text string.<br>? matches any character string. |
| **View Type** | Choose Schematic to list logic views, Symbol to list symbol views, or All for a complete list of components. |

# Rules Checker Setup - Run

***Procedures***

- Setting Up the Run Directory
- Specifying the Default .ini File

- Specifying the Type of Pin Direction Check
- Specifying Rule Dependencies
- Using Rule Dependency Information
- Specifying Maximum Message Count During a Design Check

The Run tab of Rules Checker Setup dialog box lets you specify setup details for Rules Checker.

**Default *.ini* File**

Enter the name of the initialization file, which contains information regarding the rule browser state. This file will be loaded automatically whenever Rules Checker is invoked with this project file.

**Run Directory**

Specify the path where you want Rules Checker to store the output files.

**Max Message Count**

Specify the maximum number of messages that go to the Message file (.msg) when checking your design.

**Pin Direction Check**

This is a checkbox. If it is checked, Rules Checker uses the following table to determine the direction of the pin.

| Bidirectional | Input-Load | Output-Load | Output-Type | Pin Direction |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | Error |
| 0 | 1 | 0 | 0 | Input |
| 0 | 0 | 1 | 0 | Output |
| 0 | 1 | 1 | 0 | Error |
| 0 | X | X | 1 | Output |
| 1 | X | X | X | InOut |

If it is not checked, Rules Checker uses the following table to determine the direction of the pin.

| Bidirectional | Input | Output | Pin Direction |
|---|---|---|---|
| 0 | 0 | 0 | Input |
| 0 | 1 | 0 | Input |
| 1 | X | X | InOut |

| | | | | |
|---|---|---|---|---|
| | 0 | 1 | 1 | InOut |
| | 0 | 0 | 1 | Output |

| | |
|---|---|
| **Rule Dependency Option** | Toggle on to use the rule dependency option. Dependency between rules decide whether or not a rule needs to be run and the order in which the rules are run. |
| **Dump Dependency Information** | If you toggled the rule dependency on, click Always or Never to determine whether dependency information should be printed. |
| **View** | |
| **Logical** | This is the expansion type on which Rules Checker runs when the logical environment is selected. The default is the value of Physical Layout view specified in the expansion tab of the project setup utility. The Physical Layout view is used by the packager when expanding the design. |
| **Physical** | The physical view specifies the location of the packager netlist files when the Rules Checker physical environment is selected. The default is the value of the Packaged view specified in the Views tab of the project setup utility. |

# Rules Checker Setup - Search Paths

See Specifying the Rule File/Include File Search Path for the procedure.

The Search Paths tab of Rules Checker Setup dialog box lets you specify the search paths used by Rules Checker to look for *rule files* and *include files* while checking your design.

| | |
|---|---|
| **Edit Search Path for** | Specify whether you want to edit the search path for Rule Files or Include Files. Select the appropriate option from the drop-down menu. |

| | |
|---|---|
| **Directories** | Select the appropriate path from the directory browser. Use the buttons at the top-right corner of this list to:<br>■ Add a directory to the search path.<br>■ Delete a directory from the search path.<br>■ Change the search path order by moving directories up and down.<br>You can also change the directory path by double-clicking on the path in directory browser. |

# Rules Checker Setup - Toolkit

See Setting Up Rules Checker Toolkit for the procedure.

The Toolkit tab of Rules Checker Setup dialog box lets you specify setup details for Rules Checker Toolkit. Rules Checker Toolkit lets you create your own rules and edit existing rules for checking designs.

| | |
|---|---|
| **Edit Rules** | Check this toggle if you want to create your own rules or edit existing rules. |
| **Compiled File Directory** | Specify the directory where the compiled file is to be stored. Click on Same Directory As Source File or Other. If you choose Other, enter the name of the directory in the field. |
| **Dump Debug Information** | Click on one of the radio buttons (Always, Never, If True, If Null), to determine when dependency information is to be dumped. The default is Never. |

# Select Directory Dialog

Use this dialog box to specify the directory you want to use.

# Parameters for Rule

This dialog box lets you customize parameters for rules. You can modify the existing values for a parameter and add values to the parameter. It also lets you customize the severity and content of messages associated with rules that you see in Rules Checker.

### Rule Parameters

| | |
|---|---|
| **Name** | This field displays all parameters. Select a parameter to display its values in the Value field. |
| **Value** | This field displays the current values for the selected parameter.<br>■ To modify values<br>Select the value that you want to change by clicking on it. It will be displayed in the field below.<br>Enter the new value. Click Change for the changes to apply.<br>■ To add values<br>Click Add and type new values in Value field.<br>■ To delete a value<br>Select the value that you want to delete. It will appear in the field below the Values. Click Delete to delete these values. |

### Messages

| | |
|---|---|
| Parameter | Select the parameter for which you want to change the message. |
| Value | Enter the type of message. Use the pull-down menu to select from Info, Oversight, Warning, Error or Fatal. |
| Short Message | Enter a short message. |
| Long Message | Enter a long message. |

# Global Parameters

See Edit > Global Parameters for the procedure.

This dialog box lets you customize parameters used by more than one rule.

■ To change a value for a parameter

Select the parameter in the Name field. Select a value from the list of values. It will appear in the field at the bottom. Type the new value. Click Change to apply changes to the value of the parameter.

- To add a value to a parameter

  Select the parameter in the Name field. Enter the value to be added in the empty field at the bottom of the dialog box. Click Add to add the value to the parameter.

- To delete a value from the parameter

  Select the parameter in the Name field. Select the value to be deleted from the list of values. Click Delete to delete the value from the parameter.

| | |
|---|---|
| **Name** | Select the required parameter from the drop-down menu. |
| **Value** | Displays the corresponding current values of the parameter. |

# Cadence Product Choices

This dialog box helps you specify the product suite you want to use in a session. A product suite allows you to access components that are not available in the current product suite.

# File > Open

Procedure

Opens an *.ini* file. This file contains information on the last selected environment and selected/deselected rules.

# File >Save

Procedure

Saves the active *.ini* file. If there is no active *.ini* file, you need to specify a file name. (Same as Save As)

# File > Save As

Procedure

Saves the *.ini* file by the name you specify.

# File > Change Suite

Displays the Product Choices dialog box which allows you to select the product suite whose license you want to use to run Rules Checker.

# File > Exit

Procedure

Quits Rules Checker.

# Edit > Select All

Selects all rule files and rules in your browser window. You can select an individual rule by clicking on the toggle button for that rule.

# Edit > Deselect All

Deselects all rule files in the rule browser window.

# Edit > Rules

Parameters for Rule Dialog Box

Lets you create your own rules or edit existing rules from the Rules Checker Toolkit dialog box. This is highlighted only if the Edit Rules toggle is checked in the Toolkit Setup dialog box.

***Procedures***

- Creating (Editing) a Rule
- Compiling a Rule File
- Debugging the Rule File
- Creating a Help File for the Rule

# Edit > Global Parameters

***Procedures***

- Adding a Value to Rule Parameters
- Deleting a Value from the Parameters
- Changing the Value of Parameters

Customizes common parameters used by many rules. Brings up the Global Parameters dialog box.

# Edit > Update

Updates the rule browser. Click this if you have added a new rule file or include file in your search path.

# Edit > Setup

Brings up the Rules Checker Setup dialog box. The Rules Checker Setup dialog box has three tabbed pages, that are as follows:

- Run; see Rules Checker Setup - Run for details
- Search Paths; see Rules Checker Setup - Search Paths for details
- Toolkit; see Rules Checker Setup - Toolkit for details

# Index

## Symbols



## A



## B



## C



## D



## G



## H



## I



## L

## M



## O



## R



## S



## U



## V