Leda
VeRSL Reference Guide

Version 2006.06
June 2006

Comments?
E-mail your comments about this manual to
leda-support@synopsys.com.

SYNOPSYS®

Copyright Notice and Proprietary Information

Destination Control Statement

All technical data contained in this publication is subject to the export control laws of the United States of America. Disclosure to nationals of other countries contrary to United States law is prohibited. It is the reader's responsibility to determine the applicable regulations and to comply with them.

Disclaimer

SYNOPSYS, INC., AND ITS LICENSORS MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Registered Trademarks (®)

Synopsys, AMPS, Arcadia, C Level Design, C2HDL, C2V, C2VHDL, Cadabra, Calaveras Algorithm, CATS, CSim, Design Compiler, DesignPower, DesignWare, EPIC, Formality, HSPICE, Hypermodel, iN-Phase, in-Sync, Leda, MAST, Meta, Meta-Software, ModelAccess, ModelTools, NanoSim, OpenVera, PathMill, Photolynx, Physical Compiler, PowerMill, PrimeTime, RailMill, Raphael, RapidScript, Saber, SiVL, SNUG, SolvNet, Stream Driven Simulator, Superlog, System Compiler, Testify, TetraMAX, TimeMill, TMA, VCS, Vera, and Virtual Stepper are registered trademarks of Synopsys, Inc.

Trademarks (™)

abraCAD, abraMAP, Active Parasitics, AFGen, Apollo, Apollo II, Apollo-DPII, Apollo-GA, ApolloGAII, Astro, Astro-Rail, Astro-Xtalk, Aurora, AvanTestchip, AvanWaves, BCView, Behavioral Compiler, BOA, BRT, Cedar, ChipPlanner, Circuit Analysis, Columbia, Columbia-CE, Comet 3D, Cosmos, CosmosEnterprise, CosmosLE, CosmosScope, CosmosSE, Cyclelink, Davinci, DC Expert, DC Expert Plus, DC Professional, DC Ultra, DC Ultra Plus, Design Advisor, Design Analyzer, Design Vision, DesignerHDL, DesignTime, DFM-Workbench, DFT Compiler, Direct RTL, Direct Silicon Access, Discovery, DW8051, DWPCI, Dynamic-Macromodeling, Dynamic Model Switcher, ECL Compiler, ECO Compiler, EDAnavigator, Encore, Encore PQ, Evaccess, ExpressModel, Floorplan Manager, Formal Model Checker, FoundryModel, FPGA Compiler II, FPGA Express, Frame Compiler, Galaxy, Gatran, HDL Advisor, HDL Compiler, Hercules, Hercules-Explorer, Hercules-II, Hierarchical Optimization Technology, High Performance Option, HotPlace, HSPICE-Link, iN-Tandem, Integrator, Interactive Waveform Viewer, i-Virtual Stepper, Jupiter, Jupiter-DP, JupiterXT, JupiterXT-ASIC, JVXtreme, Liberty, Libra-Passport, Library Compiler, Libra-Visa, Magellan, Mars, Mars-Rail, Mars-Xtalk, Medici, Metacapture, Metacircuit, Metamanager, Metamixsim, Milkyway, ModelSource, Module Compiler, MS-3200, MS-3400, Nova Product Family, Nova-ExploreRTL, Nova-Trans, Nova-VeriLint, Nova-VHDLlint, Optimum Silicon, Orion_ec, Parasitic View, Passport, Planet, Planet-PL, Planet-RTL, Polaris, Polaris-CBS, Polaris-MT, Power Compiler, PowerCODE, PowerGate, ProFPGA, ProGen, Prospector, Protocol Compiler, PSMGen, Raphael-NES, RoadRunner, RTL Analyzer, Saturn, ScanBand, Schematic Compiler, Scirocco, Scirocco-i, Shadow Debugger, Silicon Blueprint, Silicon Early Access, SinglePass-SoC, Smart Extraction, SmartLicense, SmartModel Library, Softwire, Source-Level Design, Star, Star-DC, Star-MS, Star-MTB, Star-Power, Star-Rail, Star-RC, Star-RCXT, Star-Sim, Star-SimXT, Star-Time, Star-XP, SWIFT, Taurus, Taurus-Device, Taurus-Layout, Taurus-Lithography, Taurus-Process, Taurus-Topography, Taurus-Visual, Taurus-Workbench, TimeSlice, TimeTracker, Timing Annotator, TopoPlace, TopoRoute, Trace-On-Demand, True-Hspice, TSUPREM-4, TymeWare, VCS Express, VCSi, Venus, Verification Portal, VFormal, VHDL Compiler, VHDL System Simulator, VirSim, and VMC are trademarks of Synopsys, Inc.

Service Marks (℠)

MAP-in, SVP Café, and TAP-in are service marks of Synopsys, Inc.

SystemC is a trademark of the Open SystemC Initiative and is used under license.
ARM and AMBA are registered trademarks of ARM Limited.
All other product or company names may be trademarks of their respective owners.

Contents

Tables

Preface

About This Manual

This manual is designed for engineers who want to develop rules in the Verilog Rule Specification Language (VeRSL). If you are unfamiliar with VeRSL, you should read and work the examples in the *Leda Rule Specifier Tutorial* before consulting this book. This manual is intended for use by design and quality assurance engineers who are already familiar with VeRSL and Verilog.

Related Documents

This manual is part of the Leda documentation set. To see a complete listing, refer to the *Leda Document Navigator*.

Manual Overview

This manual contains the following chapters:

Preface	Describes the manual and lists the typographical conventions and symbols used in it. Tells how to get technical assistance.
Chapter 1 Verilog Rule Specification Language	Overview and detailed description of the Verilog Rule Specification Language (VeRSL) that you use to write rules to constrain Verilog designs.
Chapter 2 Templates and Attributes	An API-like reference for all of the Leda templates and attributes.
Appendix A Template x Attribute SpecDex	Hyperlinked lookup tool for finding all attributes associated with each template.
Appendix B Attribute x Template SpecDex	Hyperlinked lookup tool for finding all templates associated with each attribute.

Typographical and Symbol Conventions

The following conventions are used throughout this document:

Table 1: Documentation Conventions

Convention	Description and Example
%	Represents the UNIX prompt.
Bold	User input (text entered by the user). `% cd $LMC_HOME/hdl`
`Monospace`	System-generated text (prompts, messages, files, reports). `No Mismatches: 66 Vectors processed: 66 Possible"`
Italic or *`Italic`*	Variables for which you supply a specific value. As a command line example: `% setenv LMC_HOME prod_dir` In body text: In the previous example, *prod_dir* is the directory where your product must be installed.
| (Vertical rule)	Choice among alternatives, as in the following syntax example: `-effort_level low | medium | high`
[ ] (Square brackets)	Enclose optional parameters: `pin1 [pin2 ... pinN]` In this example, you must enter at least one pin name (*pin1*), but others are optional ([*pin2* … *pinN*]).
TopMenu > SubMenu	Pulldown menu paths, such as: **File > Save As …**

Getting Leda Help

For help with Leda, send a detailed explanation of the problem, including contact information, to leda-support@synopsys.com.

The Synopsys Web Site

General information about Synopsys and its products is available at this URL:

http://www.synopsys.com

1 Verilog Rule Specification Language

Introduction

This chapter provides reference information for the Verilog Rule Specification Language (VeRSL). You use VeRSL to write custom coding rules when there isn't a prepackaged rule that meets your design team's needs. The VeRSL language supports standard Verilog, Verilog 2001, and SystemVerilog.

Note

For hardware rules, you use C or Tcl instead to write custom rules and integrate them into the Leda environment using VeRSL wrappers. For details on how to do this, see the *Leda Tcl Interface Guide* or the *Leda C Interface Guide*.

The information in this chapter is organized in the following major sections:

What is VeRSL?

VeRSL is a macro-based language that you use to write rules that check Verilog code for errors or anomalies. You write rules using prebuilt templates and attributes that match certain kinds of Verilog code, and a simple set of commands. There are six VeRSL commands: force, no, limit, set, max, and min. Each command has a precise syntax with allowed keywords.

Most of the terminology used for writing rules comes from the *Verilog Language Reference Manual* (LRM). The LRM is the basis upon which much of this manual was created. Most of the VeRSL keywords correspond to Verilog grammar rules and clauses, and can be found in the LRM. However, to avoid excessively long rule chains for some Verilog structure, VeRSL does not always follow the LRM grammar. These deviations are noted in this manual where they apply.

VeRSL's extra keywords allow you to easily define application-specific rules. For example, for synthesis it may be necessary to distinguish between a combinatorial and a sequential process or to limit the number of clocks in a process.

Note

To distinguish between Verilog rules defined in the LRM and rules for a given coding style, the former are referred to as Verilog_rules in this manual.

VeRSL Templates and Attributes

Templates and attributes are the building blocks that you combine with VeRSL commands to write rules.

A template defines a model of how the Verilog code should appear. Templates are basic elements of VeRSL code that you use to build rules or even other templates. Templates are all prepackaged (VeRSL primary template or VeRSL secondary template). You can assign any string to be (template *my_template* is *template*) where *template* is one of the prepackaged templates and define its focus using VeRSL commands, but you cannot create new templates or attributes yourself.

Each template has a set of attributes or characteristics of Verilog code that you can use with it. When you define a template to model the Verilog code you want to constrain, you select one or more attributes from this set and use VeRSL commands like force, no, or limit to precisely define that model or template. Then you write a rule that calls that template and constrains the code that the template matches.

VeRSL Templatesets and Rulesets

VeRSL contains two types of high-level semantic units: templatesets and rulesets. A templateset is like a Verilog module. It contains a set of template declarations. No commands are allowed in templatesets, but they can contain other templateset units. Rulesets can contain template declarations, commands, and other templateset units.

VeRSL Rules

A VeRSL rule can contain five parts, most often in this order:

```
command attribute context message severity
```

Table 2 describes each of these components.

Table 2: VeRSL Rule Components

Rule Component	Description	Type
command	Keyword that specifies how to constrain a Verilog item.	Required
attribute	Keyword that specifies the Verilog item to be constrained.	Required
context	Keyword that refines the focus of where the Verilog item is to be constrained. **Note:** You don't need to specify a context when you use a primary template for a rule, but you must provide a context when you use a secondary template for a rule.	Required when using secondary templates
message	Text to be displayed in the Checker's Error Viewer when the rule is violated.	Optional
severity	The severity level for violation of this rule (note, warning, error, or fatal)	Optional

The following sections provide more information about each part of a VeRSL rule:

- "VeRSL Commands" on page 22
- "VeRSL Attributes" on page 23
- "VeRSL Context" on page 24
- "VeRSL Message" on page 25
- "VeRSL Severity" on page 25

VeRSL Commands

Table 3 explains how each of the VeRSL commands works.

Table 3: VeRSL Commands

Command	Use
no	The specified Verilog_rule or clause must not be present. If it is present, the Checker flags a rule violation.
force	The specified Verilog_rule or clause must be present. If it is not present, the Checker flags a rule violation.
limit	The specified Verilog_rule or clause must match at least one of the specified set of templates. If none of the specified templates match, the Checker flags a rule violation. The limit command is the only type of VeRSL command that you can use with conditional logic (for example, if ..., then ...).
set	The specified Verilog_rule must match the specified fixed value. If it does not match, the Checker flags a rule violation.
max	The maximum number of times that the specified Verilog_rule may appear. If the maximum number is exceeded, the Checker flags a rule violation.
min	The minimum number of times that the specified Verilog_rule must appear. If the minimum number is not met, the Checker flags a rule violation.

VeRSL Attributes

An attribute represents the Verilog item that you want to constrain. There are several hundred attributes in VeRSL, and you can use them in a variety of ways, depending on their type. There are five different types of attributes. Each type is restricted for use with certain VeRSL commands, as explained in Table 4.

Table 4: Types of VeRSL Attributes

Type	Description	Use Only with These Commands
template_kind	These can be attributes of a given template and also function as templates themselves, with their own attributes. How you use this type of attribute depends on how tightly you want to focus your rule. For a wide view, use the attribute as part of another template. For a close-in view, use the attribute as its own template. For example, template_kind attributes include all statements and declarations.	• limit • no • force
local_attribute	These attributes can belong to one or more templates, but they cannot themselves function as templates. For example, you can use the downto attribute to control the use of the downto keyword.	• no • force
set_attribute	These attributes have precise meanings that are represented by enumerated types or strings. For example, you can use the evaluation_time attribute of expression templates to indicate if the expression should be locally or globally static. You specify this using a built-in enumerated type as follows: `set evaluation_time to locally_static_evaluation`	• set
max_min_attribute	These attributes are similar to set attributes, but you use them strictly for specifying bounds on certain attributes. For example, you can use the dimension attribute, (which is a max_min type) to specify the maximum or minimum number of characters in an array.	• max • min
aggregate_attribute	To impose constraints on different occurrences of the same attribute, you use aggregate_attributes. These attributes all have the "_s" suffix. For example, the condition_s attribute of the if_statement template is an aggregate_attribute.	• aggregate limit • no • force

Attribute Examples

For example, in the commands below, the attributes are shown in **bold**.

```
no      initial_construct
no      BINARY_OP1 in expression of case_statement
limit   expression in clock to CLOCK_SIMPLE_NAME
max     case_statement in always construct is 1
```

In the first example, Leda flags an error if an initial construct appears in the design unit. The second example shows a no-match command. Here, if the expression of a case_statement matches the template BINARY_OP1, Leda flags an error.

In the third example, the expression of a clock is compared against the template CLOCK_SIMPLE_NAME. If it does not match, Leda flags an error. Note that this is the opposite of the no-match command.

In the last example, Leda flags an error if the number of case statements in an always construct is greater than 1.

VeRSL Context

The context specifies where a VeRSL command applies. A context is usually a template or attribute, depending on how broad or narrow you want the application of your rule to be. You don't need to specify a context when you use a primary template for a rule, but you must provide a context when you use a secondary template for a rule.

Following are some example VeRSL rules that show the context in **bold** for easy identification:

```
no      wait_statement in always_construct
no      BINARY_UNEQUAL_OP1 in expression of case_statement
limit   condition in case_statement to BINARY_UNEQUAL_OP
max     clock in always_construct is 1
```

Note how the context is usually preceded by the "in" keyword. This is the general rule. The exception is shown in the second example, where you specify something you don't want to be present. In this case, you precede the context with the "of" keyword.

VeRSL Message

The message part of a VeRSL rule is defined as:

```
message ::= message_string [[html_doc]html_app]]
```

The message that you specify when writing a rule later appears in the Error Viewer when that rule is violated. Note that a message only appears if the severity level for that rule is high enough to get past the filtering options that you set.

You use the html_doc and html_app clauses to specify links to HTML-based help documentation for a rule. For more information, see “Linking to HTML Files” on page 30.

VeRSL Severity

Use the severity portion of a VeRSL rule to inform users about the relative seriousness of the infraction when this rule is violated. The possible values are note, warning, error, and fatal. A fatal error does not cause the Leda tool to crash, but is an indication that a violation of this kind in your Verilog code may cause serious problems for downstream tools in the design and verification flow.

If you do not specify a severity for a rule, Leda uses the default value, which is note.

VeRSL Examples

The following sections explain basic features of the VeRSL language, using examples to demonstrate how to use them:

Conditional Commands

In some cases, depending on the outcome of one rule, you may want to apply a different set of rules. For example, consider the following rules:

- Modules may only have names “RTL” or “STRUCT”
- Module instantiations are not allowed in RTL modules
- Always constructs are not allowed in STRUCT modules

To limit module identifiers, you can declare the following templates:

```
template RTL_MODULE is module_declaration
       limit identifier to "RTL"
end
template STRUCT_MODULE is module_declaration
       limit identifier to "STRUCTURAL"
end
```

and then use a limit command to restrict module names to the strings you identified in your template declarations:

```
limit module_declaration in all to RTL_MODULE, STRUCT_MODULE
       message "Illegal name for module declaration"
       severity error
```

Next, you can define rules for limiting the contents of the module declaration depending on which template was matched. If RTL_MODULE was matched you might want to forbid the use of module instantiation statements. If STRUCT_MODULE was matched, you might want to forbid the use of always constructs.

This introduces the concept of conditional commands, or commands that only get checked if a given condition is true. For the example above, you could write:

```
limit module_declaration in all to RTL_MODULE, STRUCT_MODULE
       message "Illegal name for module declaration"
       severity error

if RTL_MODULE then
       no module_instantiation
       message "Module instantiations are illegal"
       severity error
end if
if STRUCT_MODULE then
       no always_construct
       message "Always constructs are illegal"
       severity error
end if
```

Notice how these conditional rules use VeRSL limit commands. In fact, you must stick to using limit commands for conditional rules. Other VeRSL commands are not allowed with conditional commands. If you want a command to apply regardless of which template is matched, write the rule without conditionals.

ANDing Template Matching

If you want to match your Verilog code with all templates in a list, use ANDing template matching. Suppose you want to limit module instantiations so that:

- Only named port connections are used, and
- All port references are declared

You can write these rules as follows:

```
ruleset TEST is

  template NAMED_INSTANTIATION is module_instantiation
    force named_port_connection
  end

  template EXPLICIT_REFENRECE_INST is module_instantiation
    force port_reference_declaration
  end

limit    module_instantiation    to
NAMED_INSTANTIATION,EXPLICIT_REFENRECE_INST
    message "Invalid instantiation"
    severity ERROR

end ruleset
```

The disadvantage of this rule-writing method is that you have to specify a very general message with the limit rule, because it applies to more than one situation. If you want to be more specific with messages, you can use the allof keyword to indicate that you want the rule applied to every template in the list. Then, for each rule violation, you can print a specific message. For example, you can rewrite the above rules as follows:

```
ruleset TEST is

  template NAMED_INSTANTIATION is module_instantiation
    force named_port_connection
  end

  template EXPLICIT_REFENRECE_INST is module_instantiation
    force port_reference_declaration
  end
```

```
limit module_instantiation to allof
 NAMED_INSTANTIATION message "Use named association in port mapping",
 EXPLICIT_REFERENCE_INST message "All ports must be declared explicitly"
severity ERROR

end ruleset
```

Writing File Rules

Many coding standards impose rules specifying that there should be only one unit per file and that the name of the file should contain the name and type of the unit. For example, modules must be stored in files named <module_name>.v

To allow for the representation of strings containing unit names in rules, VeRSL defines a reserved string that you can use as part of other strings. This string is:

```
<module>
```

If Leda encounters the above sequence of characters in a string, it replaces that sequence of characters with the name of the corresponding unit. You can specify file names that include the name of a unit as follows:

```
limit file_name in module_declaration to "^<module>.v$"
message "Illegal file name for module"
severity error
```

If Leda is testing a module M that is not in file M.v, it flags an error.

Parameterizing Rules Using Regular Expressions

Naming conventions are not part of the Verilog language, but such rules frequently appear in various coding standards. To accommodate rules like this, Leda accepts regular expressions in strings used to constrain names.

Note

In general, results you get using regular expressions with Leda are the same as you get using grep. Both programs use regex (5) to parse regular expressions. Note that Leda currently supports simple regular expressions, but not extended regular expressions of the form \{m\}, \{m,\}, or \{m,n\} that are supported with egrep or grep -E. Also, Leda uses the GNU version of regex, which differs slightly from the UNIX version (for details see the man pages).

For example, suppose you want to constrain the instance names of module instantiation statements so that they have one of the following formats:

```
<instantiated module name>0
<instantiated module name>_<index>
<instantiated module name>_<more_info>_<index>
```

where <index> is an integer value and <more_info> is a user-defined string. You could use the following VeRSL command to test the model instantiation statements:

```
limit instance_identifier in module_instantiation to
       "^<module>0$",
       "^<module>_[0-9]*$",
       "^<module>_[.]*_[0-9]*$"
```

You can define variable parameters like this for certain attributes that are then used by the Leda Checker. This is because, in some cases, rules may be too strict. For example, when testing legacy code, some naming convention rules may be violated many times. You can deactivate these rules in the Checker or modify the rules online.

For example, the Reuse Methodology Manual (RMM) policy specifies that clock signals must have the prefix clk_. This rule is written in VeRSL as follows:

```
template CLOCK_ID is clock
       limit identifier to "^clk_"
end
```

This rule is often violated in code that was written before the RMM was implemented or that used another naming convention for clock signals. But, you can extend the rule definition in VeRSL to include a variable parameter as follows:

```
template CLOCK_ID is clock
       limit identifier to "^clk_","<macro1>"
end
```

You then must specify a value for the <macro> for Checker to read. You specify the value using a rule_set_parameter command in your config.tcl file. For more information, see the *Leda User Guide*.

Arithmetic Expressions

One area where VeRSL deviates from the LRM is in the specification of rules governing expressions. This is because the LRM grammar is too precise for writing rules. Rules that followed the LRM's description of an expression would be too long in VeRSL.

The following example shows how to model a logic 1 value (’b1) in VeRSL:

```
template LOGIC_1 is literal
       limit value       to " ^1$"
       set value_type to logic_string_literal_type
       set base to 2
end
```

You can use also use limit commands with template_kind attributes without first declaring a template. For example, the command:

```
limit object_definition to input_declaration
```

is equivalent to:

```
template PD is input_declaration
end
..

limit object_definition to PD
```

For comprehensive information about expression attributes, see “Templates and Attributes” on page 53.

Linking to HTML Files

Rule developers often implement policies, or sets of rules, based on a formal document defining the coding standard (for example, the RMM). The VeRSL language provides a way to specify a link to an HTML file with information derived from this formal document. This way, when a rule is violated, users can link directly from Leda Checker’s Error Viewer to an HTML file with more information about each rule. This makes it easy for users to look up the reason for a particular rule, and perhaps find valid and invalid code examples and circuit diagrams that help explain the issue.

In other cases, the formal document describing a coding standard may not always be useful in the context of in-house conventions that vary somewhat from the external standards. To document such variations, the VeRSL language provides a mechanism for linking to a second HTML document that is more like an application note, or current interpretation of a formal coding standard.

When you write rules using VeRSL, you can code in the locations of one or both of these HTML documents, along with the error message to be printed when the rule is violated. You use two different environment variables as common prefixes to these file locations. The two environment variables are:

- Leda_HTML_DOC_PATH (for the standard interpretations)
- Leda_HTML_USR_PATH (for application notes that vary from standards)

If you want your rules to point to files stored locally, you can set these variables as follows:

```
% setenv Leda_HTML_DOC_PATH 'file:/home/Standards/html/'
% setenv Leda_HTML_USR_PATH 'file:/home/user/notes/html/'
```

Note

You must use the file: or http: prefix when you set these variables.

Leda supports the Netscape browser for viewing these help files.

Parameterizing Error Messages

You can make error messages for Leda rules more informative by using parameters in the VeRSL/VRSL code for the message text. For example, suppose you have a rule that checks for redundant signals in sensitivity lists:

```
process (a1, a2, 14, 15, ..., a10)
[ERROR] Redundant signal in sensitivity list
```

If you write this rule using a parameter that identifies the specific signal causing the error, sifting through the error messages and correcting the problems in your HDL source code becomes a simpler task. For example:

```
process (a1, a2, 14, 15, ..., a10)
[ERROR] Signal a10 is not required in sensitivity list
```

The VeRSL rule specification language supports a set of parameters that you can use in error messages to improve their readability and usefulness (see Table 5).

Table 5: Leda Error Message Parameters

Parameter	Function	Use With ...
<%item>	Returns the name of an attribute that is a valid RTL object (flip-flop, latch, clock, or reset) or HDL named object (declarative item, object, simple name, indexed name, slice name, or selected name). For example, use with any template that has an "identifier" attribute. Replace the identifier with the parameter in the error message text.	Any VeRSL command (no, force, limit, set, max, min). For limitations on the use of "**<%item>,** see "Limitations on use of <%item>" on page 33**.**

	Example VeRSL code:
	`no latch in all`
	`message "<%item> is inferred as a latch"`
	`severity error;`
	For this rule, Leda reports the name of the latch output port in the error message.	

<%context>	Returns the name of the region/context in your HDL code where Leda flags an error. Note that if you specify the rule context as "all" in your VeRSL code, Leda cannot determine a more specific context.	Any VeRSL command (no, force, limit, set, max, min).

	Example VeRSL code:
	`no signal_decl in package_decl`
	`message "<%item> in <%context>: signals in packages not supported"`
	`severity error;`
	For this rule, Leda reports the name of the package declaration in the error message.	

Table 5: Leda Error Message Parameters (Continued)

Parameter	Function	Use With ...
<%value>	Returns the value calculated for an item.	Set, max, or min commands.

	Example VeRSL code:
	`max states in fsm is 40`
	`message"fsm with <%value> states: must be less than 40:"`
	`severity error;`
	For this rule, Leda reports the actual number of states calculated for the FSM in the error message.	

<%formal> **<%actual>**	These parameters return the formal and actual names for items in association lists (for example, port maps and subprogram calls). Use with instantiations (components, modules) or subprogram calls.	Any VeRSL command (no, force, limit, set, max, min).

	Example VeRSL Code:
	`limit port_connection in module_instantiation to T_G_521_11_1`
	`message "Use same or similar names for ports (<%formal>) and signals (<%actual>)"`
	`severity warning;`
	For this rule, Leda replaces **<%formal>** with the port name in the module definition and **<%actual>** with the associated signal.	

Limitations on use of <%item>

The **<%item>** parameter will only work if the parameterized attribute is part of the command and not embedded in a template.

For example:

```
limit <attribute> in <context> to
```

and not:

```
template T is <context>
    limit <attribute> to ...
end
limit <context> to T.
```

Only the following attributes can be parameterized:

- asynchronous_reset
- bit_select
- clock

- cmos_switch
- complete_sensitivity
- consistent_range
- disable_statement
- driver_declaration
- enable_gate
- event_declaration
- event_trigger
- flipflop
- function_declaration
- gated_clock
- global_signals_read
- inout_declaration
- input_declaration
- integer_declaration
- is_read
- latch
- missing_signals_in_sensitivity_list
- mixed_async_sync_resetline
- module_declaration
- mos_switch
- multiplexed_clock
- name
- n_input_gate
- n_output_gate
- net_declaration
- output_declaration
- parameter_declaration
- parameter_override

- part_select
- pass_en_switch
- pass_switch
- port_connection
- pull_gate
- real_declaration
- realtime_declaration
- redundancy_in_sensitivity_list
- reg_declaration
- registered_outputs
- seq_block
- side_effect
- synchronous_reset
- system_function
- system_task_enable
- task_declaration
- task_enable
- time_declaration
- udp_declaration
- unused_declaration

Formal Definition of VeRSL

This section provides an abbreviated formal definition of VeRSL syntax, using modified Backus Naur Form (BNF) notation to describe the VeRSL grammar.

A VeRSL description is a set of proton library units. There must be at least one ruleset unit, because that is where the commands are defined. The grammar for a VeRSL description is:

```
subset_specification ::= {proton_library_unit}
proton_library_unit ::= templateset | ruleset
```

Templateset

A templateset consists of a set of template declarations. It can also include other templatesets that contain commonly-used template declarations. The grammar for a templateset is:

```
templateset ::= templateset unit_identifier is
                    [template_include_part]
                    [declarative_part]
              end templateset

template_include_part ::= use [templateset] unit_identifier
```

The identifier you use in the template_include_part must be the name of a previously compiled templateset unit.

Ruleset

A ruleset consists of a set of template declarations and a set of commands. It can also include other templatesets. A ruleset is the only syntactical unit that you can use to define rules for a coding standard. The grammar for a ruleset is:

```
ruleset ::= ruleset unit_identifier is
       [template_include_part]
       [declarative_part]
       command_part
end ruleset
```

Declarative Part

Nodes are internally defined for most Verilog_rules (rules defined in the LRM). You can instantiate these nodes as templates in the declarative part of a templateset or a ruleset unit to add limitations to the Verilog_rule. The grammar for the declarative part of a ruleset is:

```
declarative_part ::= {template_declaration}

template_declaration ::=
       template template_identifier is template
             {template_command_list}
       end

template_command_list ::=
       template_decl_no_command
       | template_decl_force_command
       | template_decl_limit_command
       | template_decl_set_command
       | template_decl_max_or_min_command
```

Template identifiers are user-defined. You must specify unique template identifiers for each template.

Template No Command

The grammar for the template no command is:

```
template_decl_no_command ::= no full_attribute

full_attribute ::= template_kind
                   | local_attribute
```

The attribute must belong to the template identified in the template grammar rule in the immediately preceding template declaration.

Template Force Command

The grammar for the template force command is:

```
template_decl_force_command ::= force full_attribute
```

The attribute kind must belong to the template identified in the template grammar rule in the immediately preceding template declaration.

Template Limit Command

Template limit commands are divided into two types depending on whether the attribute to be limited is an aggregate attribute or not. The grammar for the template limit command is:

```
template_decl_limit_command ::=
             complete_template_decl_limit_command

complete_template_decl_limit_command ::= limit template_kind to
                                                  one_of_limit_list

one_of_limit_list ::=
             template_id_list
             | string_list

id_list ::= identifier |  identifier ',' id_list

string_list ::= “identifier” |  “identifier” ',' string_list
```

You must first declare a template before listing it in the template_id_list. In addition, the attribute you specify for a template must belong to all nodes referenced in the command_context.

The limit_list can be a set of previously-defined template identifiers, a set of template kinds, or a set of strings. For example:

```
template SN is name
end

template RE is binary_operation
       limit left_expression to SN
       limit right_expression to name
end
```

In this VeRSL template declaration example, the left_expression and right_expression attributes have the same limitations. They can only point to expressions of type name. For left_expression, we declared a dummy template in the example that matches all attributes of simple_name kind, whereas for right_expression, we used the template kind name directly. You cannot mix template identifiers and template kinds in the same list.

Template Set Command

The grammar for the template set command is:

```
template_decl_set_command ::= set set_attribute to set_value

set_value ::= STRING
              | number
              | enumerated_type_value

number ::= [-] DECIMAL_NUMBER

enumerated_type_value ::=
              charge_strength_type
              | edge_type
              | evaluation_periods
              | literal_type

charge_strength_type ::=
              small
              | medium
              | large

edge_type ::=
              rising
              | falling

evaluation_period ::=
              unresolved
              | locally_static_evaluation
              | globally_static_evaluation
              | dynamic_evaluation

literal_type ::=
              integer_literal_type
              | real_literal_type
              | string_literal_type
              | logic_string_literal_type
```

The set_attribute must belong to the template identified as template_name in the immediately preceding template declaration. The set_value type must correspond to the type accepted by the attribute specified.

Template Max or Min Command

The grammar for the template max or min command is:

```
template_decl_max_or_min_command ::=
             template_decl_max_command
             | template_decl_ min_command

template_decl_max_command ::= max max_min_attribute is number

template_decl_min_command ::= min max_min_attribute is number
```

The max_min_attribute must belong to the template identified as template_name in the immediately preceding template declaration. You use the max and min commands to control the number of elements in a given Verilog clause. For example, the following rule detects if the number of case items in a case statement exceeds 40:

```
template MAX_ALT_CASE is case_statement
       max_item_count is 40
end

limit case_statement to MAX_ALT_CASE
       message "More than 40 case items found in this case
       statement"
       severity WARNING
```

You can also use max and min commands to limit the number of template kinds in a rule. For example; if you want only one wait statement per always construct, you could write:

```
template SEQ_ALWAYS is always_construct
       max if_statement is 1
end
```

Command Part

You define subset rules in the command part of a rule_file. The grammar for the command part is:

```
command_part ::= {command}

command ::= no_command
             | force_command
             | limit_command
             | set_command
             | max_or_min_command
```

No Command

The grammar for the no command is:

```
no_command ::= [label :] no no_command_context no_command_options

no_command_context ::= complete_no_command
                     |no_match_command

complete_no_command ::= full_attribute [command_context]

no_match_command ::= template_name in template_kind [of template_kind]

no_command_options ::= [message] [severity]

command_context ::= in context

context ::= all
|template_kind

message ::=  message "message_string" [html_doc [html_app]]

severity ::= severity severity_level

severity_level ::= note | warning | error | fatal

html_doc ::= html_document “html_address_string”

html_app ::= html_note “html_address_string”
```

You can use the no command to prevent specified parts of the Verilog language from being used in a given context. For example:

```
no wait_statement in always_construct
```

You can also use the no command to indicate that a defined template is not allowed in a given context. For example:

```
template SUBP_WAIT is wait_statement
       no procedural_timing_control_statement
end
no SUBP_WAIT in wait_statement of always_construct
```

This example rule specifies that wait statements are allowed in always constructs unless they match the template SUBP_WAIT. The attribute specified by full_attribute must belong to all template kinds referenced in the context clause.

If you do not specify the command context, the rule applies to all relevant template kinds by default.

Force Command

The grammar for the force command is:

```
force_command ::= [label :] force full_attribute command_options

command_options ::= [command_context ] [message] [severity]
```

The force command specifies that the corresponding clause must appear in the Verilog input code. The attribute specified by full_attribute must belong to all template kinds referenced in the context clause.

Limit Command

The grammar for the limit command is:

```
limit_command ::= complete_limit_command

complete_limit_command ::= [label :] limit template_kind [command_context]
          to limit_list
                          [message] [severity]
                          {conditional_rule_block}

conditional_rule_block ::= if identifier then {command} end if

limit_list ::= one_of_limit_list
          | allof allof_id_list
          | allof allof_template_list

one_of_limit_list ::=
          template_id_list
          | string_list

all_of_id_list ::= identifier [message] | identifier [message] ','
          all_of_id_list

all_of_template_list ::= template_kind [message] | template_kind [message]
          ',' all_of_template_list
```

The attribute specified by template_kind must belong to all nodes referenced in the command context. You cannot define template identifiers and template kinds in the same list.

Set Command

The grammar for the set command is:

```
set_command ::= [label :] set set_attribute [command_context] to set_value
                [message] [severity]
```

When Leda finds the corresponding Verilog_rule in the input Verilog code, it matches the input code against the specified value. If they are not equal, Leda flags an error.

Max or Min Command

The grammar for the max or min command is:

```
max_or_min_command ::=
          max_command
          | min_command

max_command ::= [label :] max max_min_atribute is number [message] [severity]
min_command ::= [label :] min max_min_atribute is number [message] [severity]
```

You use the max and min commands to control the number of elements in a given Verilog clause. You can also use max and min commands to limit the number of template kinds in a rule. For example; if you wanted only one wait statement per always construct, you could write the following rule:

```
max wait_statement in always_construct is 1
```

VeRSL Primitives

The following primitive types are defined in VeRSL:

Charge_Strength_Type

Enumerates the different types that a charge strength value can take. You point to this enumerated type using the charge_value attribute that is part of the charge_strength template. The charge_strength_type values are:

- small
- medium
- large

Edge_Type

Enumerates the different types that a clock or asynchronous_reset edge can take. You point to this enumerated type using the edge attribute of the clock and asynchronous_reset templates. The edge_type values are:

- undefined_edge
- rising
- falling

Error_Status

Enumerates the different severity levels that a command can use. The error_status values are:

- Note
- Warning
- Error
- Fatal

Evaluation_Periods

Enumerates the different evaluation times for expressions. You point to this enumerated type using the evaluation_time attribute. The evaluation_periods values are:

- unresolved
- locally_static_evaluation
- globally_static_evaluation
- dynamic_evaluation

Literal_Type

Enumerates the different types that a literal value can take. You point to this enumerated type using the value_type attribute of the literal template. The literal_type values are:

- integer_literal_type
- real_literal_type
- string_literal_type
- logic_string_literal_type

Pragma_Type

Enumerates the different types that a case pragma can have. You point to this enumerated type using the pragma attribute of the case_statement template. The pragma_type values are:

- undefined_pragma
- synopsys_full_case
- synopsys_parallel_case
- synopsys_full_parallel_case

VeRSL Classes

Each VeRSL template belongs to one or more classes, or different types of Verilog code. Classes are also used to specify a group of template kinds that are valid for a given attribute.

There is one important exception. The class ID only includes one template, the identifier template. However, attributes of type ID can also accept string literals as values. For example, for the following template description:

```
template MODULE_ID is identifier
       limit limit_id to "mod_rtl"
end
template MODULE_WITH_ID is module_declaration
       limit identifier to MODULE_ID
end
```

the attribute identifier is of class ID, so you also could have written template MODULE_WITH_ID as:

```
template MODULE_WITH_ID is module_declaration
       limit identifier to "mod_rtl"
end
```

The following classes are defined in VeRSL:

- "ASSIGNMENT" on page 47
- "CONCURRENT_STATEMENT" on page 47
- "DECLARATIVE_ITEM" on page 48
- "DRIVEN_OBJECT" on page 48
- "DRIVER_OBJECT" on page 49
- "EXPRESSION" on page 49

ASSIGNMENT

The ASSIGNMENT class includes the following templates:

- blocking_assignment
- continuous_assign
- non_blocking_assignment
- procedural_continuous_assign
- procedural_continuous_force

CONCURRENT_STATEMENT

The CONCURRENT_STATEMENT class includes the following templates:

- always_construct
- cmos_switch
- continuous_assign
- enable_gate
- initial_construct
- module_instantiation
- mos_switch
- n_input_gate
- n_output_gate
- parameter_override

- pass_en_switch
- pass_switch
- pull_gate
- specify_block
- udp_instantiation

DECLARATIVE_ITEM

The DECLARATIVE_ITEM class includes the following templates:

- event_declaration
- function_declaration
- inout_declaration
- input_declaration
- output_declaration
- integer_declaration
- net_declaration
- parameter_declaration
- reg_declaration
- realtime_declaration
- real_declaration
- task_declaration
- time_declaration

DRIVEN_OBJECT

The DRIVEN_OBJECT class includes the following templates:

- inout_declaration
- input_declaration
- integer_declaration
- net_declaration
- reg_declaration

DRIVER_OBJECT

The DRIVER_OBJECT class includes the following templates:

- integer_declaration
- net_declaration
- realtime_declaration
- reg_declaration
- time_declaration

EXPRESSION

The EXPRESSION class includes the following templates:

- binary_operation
- bit_select
- conditional_expression
- concatenation
- function_call
- literal
- name
- negedge_event
- mintypmax_expression
- part_select
- posedge_event
- reg_assignment
- repeat_expression
- system_function

FIELD

The FIELD class includes the following template:

- range

ID

The ID class includes the following template:

- identifier

NAME

The NAME class includes the following templates:

- bit_select
- concatenation
- name
- part_select

OBJECT_ITEM

The OBJECT_ITEM class includes the following templates:

- module_declaration
- parameter_declaration
- input_declaration
- inout_declaration
- output_declaration
- net_declaration
- reg_declaration
- time_declaration
- integer_declaration
- real_declaration
- realtime_declaration
- event_declaration
- task_declaration
- function_declaration
- udp_declaration

PROCEDURAL_CONT_ASSIGNMENT

The PROCEDURAL_CONT_ASSIGNMENT class includes the following templates:

- procedural_continuous_assign
- procedural_continuous_deassign
- procedural_continuous_force
- procedural_continuous_release

REGION

The REGION class includes the following templates:

- function_declaration
- module_declaration
- par_block
- seq_block
- task_declaration
- udp_declaration

SEQUENTIAL_STATEMENT

The SEQUENTIAL_STATEMENT class includes the following templates:

- blocking_assignment
- case_statement
- casex_statement
- casez_statement
- conditional_statement
- disable_statement
- event_trigger
- for_statement
- forever_statement
- non_blocking_assignment
- par_block

- procedural_continuous_assign
- procedural_continuous_deassign
- procedural_continuous_force
- procedural_continuous_release
- procedural_timing_control_statement
- repeat_statement
- seq_block
- system_task_enable
- wait_statement
- while_statement

TARGET

The TARGET class includes the following templates:

- bit_select
- concatenation
- name
- part_select

2 Templates and Attributes

Introduction

This chapter presents an API-like reference for the VeRSL templates and attributes, organized in the following major sections:

Chip-Level Templates and Attributes

Leda applies chip-level rules to the entire design hierarchy, whereas it applies block-level rules to each unit or module individually. You can write chip-level rules using the templates and attributes described in this section. Note that there are some block-level templates, such as clock and synchronous_initialization, which contain attributes that you can also use for writing chip-level rules. Reference information for the chip-level templates and attributes is presented in the following subsections, one for each template:

Array Literal Template

You can use the array_literal template to check the use of array literals.

Connectivity Path Template

Some DFT rules concern what appears on the path of data or control signals. Also, some rules imply the analysis of data signal's path; thus, a connectivity_path attribute is added to each data signal. The connectivity_path template is a secondary template belonging to no classes. It contains the attributes shown in Table 6.

Table 6: connectivity_path Template

Attribute	Kind	Limit_Kind
buffer_count	max/min	N/A
inverter_count	max/min	N/A
is_combinatorial	local	N/A
starting_unit	template	REGION_PART
data	local	N/A
flipflop_as_source	local	N/A
latch_as_source	local	N/A
is_reset	local	N/A
control_src_count	max/min	N/A
within_same_clkdomain	local	N/A

- Use the starting_unit attribute to constrain if there is any control or data on.
- Use the data attribute to constrain if there is any data on the connectivity path.
- Use the flipflop_as_source attribute to constrain a signal to be driven by a flip-flop output signal. This attribute was used to specify the DFT rules TEST_980 and TEST_981.
- Use the latch_as_source attribute to constrain a signal to be driven by a latch output signal. This attribute was used to specify the DFT rules TEST_974, TEST_975, TEST_978, and TEST_979.
- Use the is_reset attribute to constrain the clock signal to be also a reset. This attribute was used to specify the DFT rule TEST_994.

- Use the control_src_count attribute to constrain the number of clock signals that control a register. This attribute was used to specify the DFT rules TEST_976 and TEST_977.
- Use the within_same_clkdomain attribute to constrain the path between two registers to be within the same clock domains. This attribute was combined with the flipflop_as_source and latch_as_source attributes to specify the DFT rules TEST_974, TEST_975, and TEST_978 through TEST_981.

Connectivity Path Example

Here is an example rule written in VeRSL that uses the connectivity path template:

```
TEST_974 : avoid latch enabled by clock clk which affects data input of
latch on the same clock.

template CN974 is connectivity_path
   force within_same_clkdomain
   no latch_as_source
end

template D974 is data_signal
   limit connectivity_path to CN974
end

template L974 is latch
   limit data_signal to D974
end

TEST_974:
limit latch in design to L974
   message "Latch as source and latch as destination of data path on the
   same clock is not allowed"
severity ERROR
```

Data Signal Template

Just as the clock, asynchronous_initialization, and synchronous_initialization templates constrain clocks and resets for registers, the data_signal template constrains the data path to registers. Each register (flip-flop or latch) therefore has a data_signal attribute that you can use to write Design For Test (DFT) rules. The data_signal template has an attribute named connectivity_path just like the clock, asynchronous_initialization, and synchronous_initialization templates. The data_signal template is a secondary template belonging to no classes. It contains the attributes shown in Table 7.

Table 7: data_signal Template

Attribute	Kind	Limit_Kind
connectivity_path	template	connectivity_path
fixed_value	local	N/A

Use the fixed_value attribute to detect data with inputs that have fixed values.

Design Template

The design template is the basic chip-level template. It contains attributes for all on-off rules. In addition, the design template contains attributes that you can use to limit clocks and resets, and build other, more complex rules. Insert the design template at the top level of a ruleset like all primary templates (template ruleset has attribute design, which points to templates of this type). The design template is a primary template belonging to no classes. It contains the attributes shown in Table 8.

Table 8: design Template

Attribute	Kind	Limit_Kind
asynchronous_initialization	template	asynchronous_initialization
clock	template	clock
synchronous_initialization	template	synchronous_initialization
initialization_count	max/min	N/A
load_count	max/min	N/A
set_count	max/min	N/A
top_unit	template	REGION_PART
asynchronous_feedback	local	N/A

Table 8: design Template (Continued)

Attribute	Kind	Limit_Kind
asynchronous_logic	local	N/A
comb_delay	max/min	N/A
drivers_per_signal	max/min	N/A
non_tristate_drivers_per_signal	max/min	N/A
pulse_generator	local	N/A
meta_stability	local	N/A
flipflop	template	flipflop
gated_clock	local	N/A
clock_count	max/min	N/A
latch	template	latch
mixed_clock	local	N/A
multiplexed_clock	local	N/A
reset_count	max/min	N/A
gated_initialization	max/min	N/A
mixed_async_sync_line	max/min	N/A
glue_logic_at_top	local	N/A
registered_outputs	local	N/A
registered_inputs	local	N/A
sync_ff_count	max/min	N/A
comb_cost	limit	logic_cost
logic_level	limit	logic_cost

- Use the top_unit attribute to constrain the top-level unit (for example, control its name or its containing file).
- Use the meta_stability attribute to make sure that there are at least two consecutive flip-flops on the data flow path when changing clock domains.

- Use the gated_clock attribute to constrain the presence of gated clocks in the whole design. If you want to allow gated clocks in specific units, use clock templates instead.
- Use the flip-flop and latch attributes to constrain the flip-flops and latches in the design. This can be a constraint of its connectivity_path attribute to specify a DFT rule.
- Use the mixed_async_sync_resetline attribute to detect reset lines that are used as both synchronous and asynchronous register resets.
- Use the sync_ff_count attribute to constrain the flip-flop synchronizer number for metastability rule checks. The default flip-flop synchronizer number is two, as shown in the following VeRSL rule:

  ```
  no meta_stability in design
  ```

  If you want a different number of synchronizers, write the rule as shown in the following example:

  ```
  template META is design
     max sync_ff_count is 4
     min sync_ff_count is 4
  end
  limit design to META
  message "needs 4 synchronizers"
  severity error
  ```
- Use the comb_cost and logic_level attributes to associate an identical cost to each operation in the logic cost template and make sure the max_cost does not exceed the specified threshold.

Flipflop Template

The flipflop template does not correspond to any part of the formal definition of the Verilog language. Instead, you use this template to constrain hardware flip-flops inferred from the Verilog code. The flipflop template is a primary template belonging to no classes. It contains the attributes shown in Table 9.

Table 9: flipflop Template

Attribute	Kind	Limit_Kind
asynchronous_initialization	template	asynchronous_initialization
identifier	template	ID
input	template	EXPRESSION
data_signal	template	data_signal
has_clock_as_data	local	N/A
synchronous_initialization	template	synchronous_initialization

Use the has_clock_as_data attribute to constrain a clock signal to be a clock and data input signal of the same flip-flop. For example, DFT rule TEST_972 differs from TEST_970, where a clock is a data input of another flip-flop. TEST_970 is specified using the data attribute of the connectivity_path attached to the clock template.

Latch Template

The latch template does not correspond to any part of the formal definition of the Verilog language. Instead, you use this template to constrain hardware latches inferred from the Verilog code. The latch template is a primary template belonging to no classes. It contains the attributes shown in Table 10.

Table 10: latch Template

Attribute	Kind	Limit_Kind
asynchronous_initialization	limit	asynchronous_initialization
identifier	template	ID
input	template	EXPRESSION
data_signal	template	data_signal
has_clock_as_data	local	N/A

Use the has_clock_as_data attribute to constrain a clock signal to be a clock and data input signal of the same latch. For example, DFT rule TEST_973 differs from TEST_971, where a clock is a data input of another latch. TEST_973 is specified using the data attribute of the connectivity_path attached to the clock template.

Logic Cost Template

The logic_cost template does not correspond to any part of the formal definition of the Verilog language. Instead, you use this template to constrain the logic in hardware logic blocks. The logic_cost template is a secondary template belonging to no classes. It contains the attributes shown in Table 11.

Table 11: logic_cost Template

Attribute	Kind	Limit_Kind
max_cost	set	<integer>
multiply_cost	set	<integer>
divide_cost	set	<integer>
modulus_cost	set	<integer>
abs_cost	set	<integer>
plus_cost	set	<integer>
minus_cost	set	<integer>
remainder_cost	set	<integer>
power_cost	set	<integer>
and_cost	set	<integer>
nand_cost	set	<integer>
or_cost	set	<integer>
nor_cost	set	<integer>
xor_cost	set	<integer>
xnor_cost	set	<integer>
comparator_cost	set	<integer>
mux_cost	set	<integer>
buffer_cost	set	<integer>
function_cost	set	<integer>
shift_cost	set	<integer>
decoder_cost	set	<integer>
reset_at_hierarchical_boundary	local	N/A

Use the cost attributes with the set command to constrain hardware logic blocks to specified levels for the various items. For example, you can use this template to build rules that constrain the number of arithmetic operators used in a logic block. Note that cost in the context of the logic_cost template is a generic term that can refer to code complexity, timing, area, or power, depending on the attribute used and your application of the rule.

For example, if you want to write a rule that prohibits the use of complex arithmetic if it includes multiplication expressions similar to the following:

```
a = b*c+d
```

you can set the multiply_cost operator to be extremely high without regard to the other operators as follows:

```
template MY_COSTS is logic_cost
    set max_cost to 100
    set multiply_cost to 100
    set adder_cost to 0
end
MYRULE_1:
limit comb_logic in design to MY_COSTS
...
```

This rule fires if it finds a multiplication operator (*) in the hardware logic block.

Use the max_cost attribute to set a total cost for the sum of all other cost attributes specified in a rule.

Test Signal Template

A lot of design for test (DFT) rules concern what appears on the path of test control signals, especially the rules detecting errors which prevent scan insertion (labelled TEST_953, TEST_954, and TEST_963 through TEST_969). The asynchronous_initialization, synchronous_initialization, and clock templates therefore all have a test_signal attribute which you can use to constrain the test clock and test reset signals. This allows you to specify rules that check the controllability of a register (flip-flop or latch) for DFT.

The test_signal template is a secondary template belonging to no classes. It contains the attributes shown in Table 12.

Table 12: test_signal Template

Attribute	Kind	Limit_Kind
control_at_start	local	N/A
disable_control	local	N/A
hold_latch_data	local	N/A
reach_memory	local	N/A

- Use the control_at_start attribute to constrain a test clock signal to control a register at the beginning of the cycle. This attribute was used to write the DFT rules TEST_963 and TEST_964.
- Use the disable_control attribute to constrain a test reset signal to be able to disable the register reset control. This attribute was used to write the DFT rules TEST_968 and TEST_969.
- Use the hold_latch_data attribute to ensure that the test clock signal holds data in latches at the beginning of the cycle. This attribute was used to write the DFT rule TEST_965.
- Use the reach_memory attribute to ensure that the test clock signal and the test reset signal are able to reach control signals of registers.

Test Signal Template Example

Here is an example rule written in VeRSL that uses the test_signal template:

```
TEST_953: Flipflop's clock is not reached by any test clock
template TS953 is test_signal
   force reach_memory
end

template CK953 is clock
   limit test_signal to TS953
end

template FF953 is flipflop
   limit clock to CK953
end

TEST_953:
limit flipflop in design to FF953
   message "Flipflop is not reached by any test clock"
   severity ERROR
```

Block-Level Templates and Attributes

You use the templates and attributes described in this section to develop coding rules that operate on the HDL block or module level, in contrast to the chip-level templates and attributes, which you use to develop rules that work on the entire design hierarchy. There are two main types of block-level rules that you can build using these templates and attributes:

- **Language-based rules**—use to constrain different Verilog constructs to ensure that they correspond to acceptable values, ranges, or conventions.
- **Hardware-based rules**—use to control the hardware semantics of Verilog. Certain Verilog code results in specific hardware features when you synthesize the descriptions (for example, latches, flip-flops, and finite state machines). You can build hardware-based rules to check that inferred hardware in your design is used correctly.

Reference information for the block-level templates and attributes is presented in the following subsections, one for each template:

Always Construct Template

The LRM (§9.9.2) defines the grammar for always construct as follows:

```
always_construct ::= always statement

statement ::= blocking_assignment ;
            | non_blocking_assignment ;
            | procedural_continuous_assignments ;
            | procedural_timing_control_statement
            | conditional_statement
            | case_statement
            | loop_statement
            | wait_statement
            | disable_statement
            | event_trigger
            | seq_block
            | par_block
            | task_enable
            | system_task_enable
```

The always_construct template is a primary template belonging to the CONCURRENT_STATEMENT class. It contains the attributes shown in **Table 13**.

Table 13: always_construct Template

Attribute	Kind	Limit_Kind
always_type	set	always_type
asynchronous_initialization	template	asynchronous_initialization
blocking_assignment	template	blocking_assignment
case_statement	template	case_statement
casex_statement	template	casex_statement
casez_statement	template	casez_statement
conditional_statement	template	conditional_statement
disable_statement	template	disable_statement
event_trigger	template	event_trigger
for_statement	template	for_statement
forever_statement	template	forever_statement
fsm	local	N/A
input_count	max/min	N/A

Table 13: always_construct Template (Continued)

Attribute	Kind	Limit_Kind
mixed_assignment	local	N/A
asynchronous_reset_signal	max/min	asynchronous_reset_signal
asynchronous_set_signal	max/min	asynchronous_set_signal
asynchronous_load_signal	max/min	asynchronous_load_signal
asynchronous_initialization_signal	max/min	asynchronous_initialization_signal
synchronous_reset_signal	max/min	synchronous_reset_signal
synchronous_set_signal	max/min	synchronous_set_signal
synchronous_load_signal	max/min	synchronous_load_signal
synchronous_initialization	template	synchronous_initialization
synchronous_initialization_signal	max/min	synchronous_initialization_signal
multiply_assigned_signals	template	N/A
modified_sensitivity_list_variable	local	N/A
mux	template	mux
non_blocking_assignment	template	non_blocking_assignment
par_block	template	par_block
procedural_continuous_assign	template	procedural_continuous_assign
procedural_continuous_deassign	template	procedural_continuous_deassign
procedural_continuous_force	template	procedural_continuous_force
procedural_continuous_release	template	procedural_continuous_release
procedural_timing_control_statement	template	procedural_timing_control_statement
process	template	process
process_statement	template	process_statement
repeat_statement	template	repeat_statement
sensitivity_element_is_fully_used	local	N/A
sensitivity_element_is_incomplete	local	N/A

Table 13: always_construct Template (Continued)

Attribute	Kind	Limit_Kind
sensitivity_list	limit	sensitivity_list
seq_block	template	seq_block
multiply_blocking_assigned_ signal	local	N/A
statement	template	SEQUENTIAL_STATEMENT
system_task_enable	template	system_task_enable
task_enable	template	task_enable
wait_statement	template	wait_statement
while_statement	template	while_statement
clock	template	clock
combinatorial	local	N/A
complete_sensitivity	local	N/A
flipflop	template	flipflop
fully_assign_signals	local	N/A
latch	template	latch
redundancy_in_sensitivity_list	local	N/A
missing_signals_in_sensitivity_list	local	N/A
signals_driven	max/min	N/A
arith_expression_count	max/min	N/A
if_statement_count	max/min	N/A
clock_in_condition	local	N/A

- Use the combinatorial attribute to control whether an always construct is combinatorial. An always construct is combinatorial if it has an event list that does not contain the posedge or negedge keywords.
- Use the statement attribute to control the statement of an always block.

- Use the fsm attribute to test if the always construct belongs to a finite state machine (FSM). For example, the following VeRSL flags always constructs that are not part of an FSM:

```
force fsm in always_construct
```

- Use the fully_assign_signals attribute with the force command to make sure every variable assigned in an always construct is assigned on every possible flow of control.
- Use the mutiply_assigned_signals attribute with force or no commands to force or forbid multiple assignments for signals.
- Use the complete_sensitivity attribute with the force command to make sure every object in the sensitivity is read in the process and every object read is in the sensitivity list.
- Use the redundancy_in_sensitivity_list attribute to detect unnecessary signals in the sensitivity list.
- Use the missing_signals_in_sensitivity_list to detect signals that are read inside the always block, but are not in the sensitivity list.
- Use the signals_driven attribute to control the number of signals driven inside an always construct.

Note that reg_declaration is not an attribute of the always_construct template, as per the Verilog LRM. Therefore, the following template declaration does not compile:

```
template A1 is always_construct
   no reg_declaration -- FAIL does not compile
end
```

but you can write:

```
no reg_declaration in always_contruct
```

because it respects the grammar of the Leda no command.

Also, notice that:

```
no reg_declaration in always_contruct
```

has the same effect as:

```
template A is always_construct
end
limit always_construct to A
  if A then
    no reg_declaration
      end if
```

- Example (mixed_assignment):

```
module test(a,b,c);
input a,b;
output [1:0] c;
reg [1:0] c;
reg d;

always @(a or b) begin
    c[1] = a;
    c[0] <= b; //FAIL
end

always @(a) begin
    d = a;
    d <= b; //FAIL
    c = 2'b00;
end

endmodule

no mixed_assignment in always_construct
message "Variable is assigned in both blocking and non-blocking
assignments"
severity WARNING
```

- Example (modified_sensitivity_list_variable):

```
always @(sig1 or sig2)
    sig1 = 1'b0; //FAIL
    ...

no modified_sensitivity_list_variable in always_construct
message "A variable in the sensitivity list is modified inside the
block"
severity WARNING
```

- Example (sensitivity_element_is_fully_used):

```
force sensitivity_element_is_fully_used in always_construct
message "Bus variable in the sensitivity list, but not all its bits
are used in the block"
severity WARNING

always @ ( portr[6:0] ) /* FAIL */
    begin
          if ( portr[0] ==1'b0 )
                out = &portr[3:1];
          else
                out = portr[3];
    end
```

- Example (sensitivity_element_is_incomplete):

```
input [7:0] a;
input en;
output o;
reg o;

always @(a[1] or en) begin
    if(en)
    o = &a;
...

no sensitivity_element_is_incomplete in always_construct
message "not all the bits of the variable are in the sensitivity
list"
severity WARNING
```

- Example (sensitivity_list):

```
template ALWAYS_WITH_VALID_SENS_LIST is always_construct
    force sensitivity_list
end

limit always_construct to ALWAYS_WITH_VALID_SENS_LIST
message "no event control statement in always block"

always // PASS
    begin
          @(a)
          y = a;
    end

always // FAIL
    begin
          y = a;
          @ (a) ;
    end
```

- Use the arith_expression_count attribute to restrict the total number of arithmetic expression under always construct.
- Use the if_statement_count attribute to count the number of if statements in an always block in the same level.
- Use the clock_in_condition attribute to check if the clock is used in a conditional expression.
- Use the multiply_blocking_assigned_signal attribute to check if a signal is assigned more than once using blocking assignment in an always block.

Asynchronous Initialization Template

The asynchronous_initialization template does not correspond to any part of the formal definition of the Verilog language. Instead, you use this template to constrain hardware asynchronous resets inferred from the Verilog code. The asynchronous_initialization template is a primary template belonging to no classes. It contains the attributes shown in **Table 14**.

Table 14: asynchronous_intialization Template

Attribute	Kind	Limit_Kind
edge	set	edge_type
expression	template	EXPRESSION
gated_initialization	local	N/A
identifier	template	ID
is_load	local	N/A
is_reset	local	N/A
is_set	local	N/A
object_definition	template	object_item
connectivity_path	template	connectivity_path
gated_in_unit	template	ID

Attribute Instance Template

```
attribute_instance ::= (* attr_spec { , attr_spec } *)
attr_spec ::=attr_name = constant_expression
        | attr_name
attr_name ::= identifier
```

The attribute_instance template contains the attributes shown in Table 15.

Table 15: attribute_instance Template

Attribute	Kind	Limit_Kind
identifier	template	ID
expression	template	EXPRESSION

Binary Operation Template

The VeRSL representation of expressions deviates from the LRM representation because the latter is too detailed and leads to long chains of design rules to express even simple expressions.

Use the binary_operation template to constrain the left- and right-hand expressions and operators of binary operations. You can also control unary operations using the binary_operation template by using the left_expression attribute along with the VeRSL no command.

The binary_operation template is a primary template belonging to the EXPRESSION class. It contains the attributes shown in Table 16.

Table 16: binary_operation Template

Attribute	Kind	Limit_Kind
left_expression	template	EXPRESSION
right_expression	template	EXPRESSION
operator_symbol	template	operator_symbol
evaluation_time	set	evaluation_periods
bit_length	max/min	N/A
operand_size_match	local	N/A
operand_size_match_no_carry	local	N/A
operand_sign_match	local	N/A
in_assertion	template	N/A
special_percentile_handle	local	N/A

- Use the operator_symbol attribute with character strings or templates of class ID representing the acceptable operators. For example:

```
limit operator_symbol in binary_operation to "+"
```

- You can set the evaluation_time attribute with any of the following enumerated literals:
  - unresolved
  - locally_static_evaluation
  - globally_static_evaluation

- dynamic_evaluation

For example:

```
set evaluation_time to locally_static_evaluation
```

- Use the bit_length attribute to control of the resulting bit length of the binary operation. If Leda cannot evaluate the bit length of the left- or right-hand side, the resulting bit length is –1.
- Use the operand_size_match attribute to control whether the left- and right-hand sides of the binary operation are the same size. If Leda cannot evaluate the size of any of the elements, the operation is considered to have an operand size match.
- Use the operand_size_match_no_carry attribute to control whether the left- and right-hand sides of a binary operation are the same size. This attribute differs from the operand_size_match attribute in that it does not take into account the carry bit from addition or subtraction. For example:

```
force operand_size_match_no_carry in continuous_asign

wire a, b, c;
assign a = b+c; // rule does not fire here
```

- Use the operand_sign_match attribute to check that operands have same sign as per LRM. If a mixture of signed and unsigned operands are used, the result is unsigned and may give unexpected results. The LRM indicates for all expression whether it's signed or unsiged, and this information is used by the attribute. It is used by prepackaged rule B_3212 in Leda policy and VER_2_10_6_3 in STARC_DSG policy. For example:

```
module B_3212 (s_a, s_b2, s_b3, us_b);
       input signed [9:0] s_a,s_b2;
       input signed [9:0] s_b3;
       input unsigned [9:0] us_b;
       reg signed [10:0] s_x;
       wire signed [10:0] s_x2;

//Tests for signed/unsigned arithmetic operations

       assign s_x2 = s_a - s_b3 * f(us_b); //pass
       assign s_x2 = s_a - s_b2 * s_a; //pass
       assign s_x2 = f(us_b) + $unsigned(b); //Fail
       assign s_x2 = g(us_b) + $unsigned(b); //pass
       assign s_x2 = $signed(a) + $unsigned(b); //Fail
       assign s_x2 = $unsigned(a) + $unsigned(b); //pass
       assign s_x2 = s_a + us_b; //Fail

       always @(s_a or us_b or s_b2) begin
             s_x <= s_a - us_b * s_b2; //Fail
```

```
            s_x <= s_a - s_b2 * us_b; //Fail
            s_x <= s_a - s_b3 * s_a; //pass
            s_x <= s_a - s_b3 * f(us_b); //pass
            s_x <= s_a - s_b3 * g(us_b); //fail
        end
endmodule

function signed f;
input unsigned [9:0] c;
  return 1;
endfunction

function unsigned g;
input unsigned [9:0] c;
  return $unsigned(1);
endfunction
```

- Use the in_assertion attribute to check if the expression is within a SystemVerilog assertion. For example:

```
template BINARY_NOT_WITHIN_ASSERTION is binary_operation
    no in_assertion
end

template B_2010_UNSYNTH_OPERATORS is binary_operation
   limit operator_symbol to "===","!=="
end

B_2010:M_0346:
limit binary_operation to BINARY_NOT_WITHIN_ASSERTION
if BINARY_NOT_WITHIN_ASSERTION then
B_2010:M_0346:
  no B_2010_UNSYNTH_OPERATORS in binary_operation
    message "Non synthesizable operator ===, !== encountered"
    severity ERROR
end if
```

Bit Declaration Template

The bit_declaration template contains the attributes shown in Table 17.

Table 17: bit_declaration Template

Attribute	Kind	Limit_Kind
identifier	template	ID
expression	template	EXPRESSION

Table 17: bit_declaration Template (Continued)

Attribute	Kind	Limit_Kind
signed	local	N/A
unsigned	local	N/A
static	local	N/A
automatic	local	N/A
packed_dimension	template	range
packed_dimension_count	max/min	N/A
unpacked_dimension	template	range
unpacked_dimension_count	max/min	N/A

Bit Select Template

An expression matches the bit_select template if it has the following syntax:

```
identifier [expression]
```

The bit_select template is a primary template belonging to the EXPRESSION_NAME class. It contains the attributes shown in **Table 18**.

Table 18: bit_select Template

Attribute	Kind	Limit_Kind
identifier	template	ID
index	template	EXPRESSION
flipflop	template	filpflop
full_range	local	N/A
latch	template	latch
object_definition	template	OBJECT_ITEM
upward_reference	local	N/A
out_of_range	local	N/A

- Use the index attribute to control the expression between brackets.
- Use the flipflop attribute to control whether the object infers a flip-flop.

- Use the latch attribute to control whether the object infers a latch.
- Use the object_definition attribute to control the type of definition associated with bit_select. For example, if you only want to consider ports in a bit_select expression, you can write the following VeRSL template:

```
template CLK_NAME is bit_select
    limit object_definition to input_declaration,
          inout_declaration,
          output_declaration
end
```

- Use the out_of_range attribute to control whether the index value is outside of the range specification. For example:

```
reg [3:0] a;
reg b,c;
always begin
  case(a[4])
   ….
```

  The following VeRSL rule causes Leda to flag an error on the previous Verilog code example:

```
template LEGAL_BIT_SELECT is bit_select
  no out_of_range
end
limit expression in case_statement to LEGAL_BIT_SELECT
  message "Index out of range"
  severity ERROR
```

- Use the full_range attribute to detect if the index value can cover the whole range of a vector. For example:

```
input clock;
input [1:2] i;
input DI;
output [0:8] mem;
reg [0:8] mem; //memory cells
wire [1:15] d;

always @(posedge clock)
    begin
          ...
          mem[i] <= aa(d) ; //FAIL
          ...

template ANY_BA is blocking_assignment
end

limit blocking_assignment to ANY_BA
if ANY then
force full_range in bit_select
message "index variable is too short"
severity WARNING
end if
```

- Upward references are a feature of SystemVerilog. You can use the upward_reference attribute to write rules that do not allow the use of upward references in your Verilog code. For example, the following VeRSL code prohibits the use of upward references in module instantiations:

```
template MY_TEMPLATE is name
no upward_reference
end

template MY_TEMPLATE is port_connection
force port_expression
limit port_expression to MY_TEMPLATE
end

Rule1:
limit port_connection in module_instantiation to MY_TEMPLATE
  message "Upward refs not allowed in module instantiation"
  severity WARNING
```

Blocking Assignment Template

The LRM (§9.2.1) defines the grammar for blocking assignment as follows:

```
blocking_assignment ::= reg_lvalue = [delay_or_event_control] expression

delay_or_event_control ::= delay_control
                         | event_control
                         | repeat ( expression ) event_control

reg_lvalue ::= reg_identifier
             | reg _identifier [ expression ]
             | reg _identifier [ msb_constant_expression :
               lsb_constant_expression ]
             | reg_concatenation

delay_control ::= # delay_value)
                | # ( mintypmax_expression )

event_control ::= @ event_identifier
                | @ ( event_expression )

event_expression ::= expression
                   | event_identifier
                   | posedge expression
                   | negedge expression
                   | event_expression or event_expression
```

The blocking_assignment template is a primary template belonging to the EXPRESSION class. It contains the attributes shown in Table 19.

Table 19: blocking_assignment Template

Attribute	Kind	Limit_Kind
delay_control	template	delay_control
event_control	template	event_control
expression	template	EXPRESSION
reg_lvalue	template	EXPRESSION
read_write	local	N/A
repeat_event	template	repeat_event
function_in_lhs	local	N/A
overflow	local	N/A
operand_size_match	local	N/A

Table 19: blocking_assignment Template (Continued)

Attribute	Kind	Limit_Kind
operand_size_match_no_carry	local	N/A
operator	template	Operator
one_assignment_per_line	local	N/A
lr_sign_match	local	N/A
special_percentile_handle	local	N/A

- Use the overflow attribute to catch cases where the left-hand side of an assignment is smaller than the right-hand side. Note that the operand_size_match attribute applies to any expression, whereas the overflow attribute is just for assignments.
- Use the operand_size_match attribute to control whether the operands (right- and left-hand side) of any expression are the same size. If the size of any of the elements cannot be evaluated, Leda considers the statement to have an operand size match.
- Use the operand_size_match_no_carry attribute to control whether the left- and right-hand sides of a binary operation are the same size. This attribute differs from the operand_size_match attribute in that it does not take into account the carry bit from addition or subtraction. For example:

```
force operand_size_match_no_carry in continuous_asign

wire a, b, c;
assign a = b+c; // rule does not fire here
```

- Use the read_write attribute to control whether the same signal is present on both sides of an assignment.
- Use the lr_sign_match attribute with force/no keywords to check if the left hand side and right hand side of the assignment statement are both signed or unsigned.

Byte Declaration Template

The byte_declaration template contains the attributes shown in Table 20.

Table 20: byte_declaration Template

Attribute	Kind	Limit_Kind
identifier	template	ID
expression	template	EXPRESSION

Table 20: byte_declaration Template (Continued)

Attribute	Kind	Limit_Kind
signed	local	N/A
unsigned	local	N/A
static	local	N/A
automatic	local	N/A
unpacked_dimension	template	range
unpacked_dimension_count	max/min	N/A

Case Item Template

The LRM (§9.5) defines the grammar for case statement as follows:

```
case_statement ::= case ( expression ) case_item { case_item } endcase
                   casex ( expression ) case_item { case_item }
                     endcase
                   casex ( expression ) case_item { case_item }
                     endcase

case_item ::=  expression {, expression } : statement_or_null
             | default [ : ] statement_or_null

statement_or_null ::= statement| ;
```

The case_item template is a primary template belonging to no classes. It contains the attributes shown in Table 21.

Table 21: case_item Template

Attribute	Kind	Limit_Kind
default	local	N/A
expression	template	EXPRESSION
expression_count	max/min	N/A
statement	template	SEQUENTIAL_STATEMENT

- Use the default attribute to control whether the case expression is the default expression.
- Use the expression attribute to control case item expressions.

- Use the expression_count attribute to control the number of case item expressions that are separated by commas.
- Use the statement attribute to control case item statements.

Case Statement Template

The case_statement template is a primary template belonging to the SEQUENTIAL_STATEMENT class. It contains the attributes shown in Table 22.

Table 22: case_statement Template

Attribute	Kind	Limit_Kind
process_statement	template	process_statement
unique	local	N/A
priority	local	N/A
expression	template	EXPRESSION
default	local	N/A
null	local	N/A
blocking_assignment	template	blocking_assignment
case_statement	template	case_statement
casex_statement	template	casex_statement
casez_statement	template	casez_statement
conditional_statement	template	conditional_statement
disable_statement	template	disable_statement
event_trigger	template	event_trigger
for_statement	template	for_statement
forever_statement	template	forever_statement
repeat_statement	template	repeat_statement
while_statement	template	while_statement
do_while_statement	template	do_while_statement
non_blocking_assignment	template	non_blocking_assignment
par_block	template	par_block

Table 22: case_statement Template (Continued)

Attribute	Kind	Limit_Kind
procedural_continuous_assign	template	procedural_continuous_assign
procedural_continuous_deassign	template	procedural_continuous_deassign
procedural_continuous_force	template	procedural_continuous_force
procedural_continuous_release	template	procedural_continuous_release
procedural_timing_control_statement	template	procedural_timing_control_statement
seq_block	template	seq_block
system_task_enable	template	system_task_enable
task_enable	template	task_enable
wait_statement	template	wait_statement
full_case	local	N/A
parallel_case	local	N/A
case_item	template	case_item
item_count	max/min	N/A
null_statement	template	null_statement
pragma	set	pragma_type
default_as_last	local	N/A
operand_size_match	local	N/A
operand_size_match_no_carry	local	N/A
duplicated_case_item	local	N/A
event_control	template	event_control
signals_driven	max/min	N/A
statement_format	template	statement_format
redundant_case_item	local	N/A
case_exp_size_exceeding	local	N/A

- Use the case_item attribute to control a case item expression and statement (including the default).
- Use the default attribute to control the presence of a default branch.
- Use the expression attribute to control case expressions.
- Use the item_count attribute to control the number of case items (excluding the default item).
- Use the full_case attribute to control whether all the possible branches of the case statement are covered (regardless of the default case item). For example, the following VeRSL command:

```
force_full_case in case_statement
```

causes Leda to flag an error for Example 1 but not for Example 2**:**

```
// Example 1
input [1:0] a;
always @ ( a or w or x or y or z)
    case (a)
       2'b00: b = w;
       2'b01: b = x;
       default : b = 1'bz;
    endcase

// Example 2
input [1:0] a;
always @ ( a or w or x or y or z)
    case (a)
        2'b00: b = w;
        2'b01: b = x;
        2'b10: b = y;
        2'b11: b = z;
        default : b = 1'bz;
    endcase
```

- You can set the pragma attribute to any of the following enumerated values:
  - undefined_pragma
  - synopsys_full_case
  - synopsys_parallel_case
  - synopsys_full_parallel_case
- Use the undefined_pragma attribute to make sure that no Synopsys case directive is present.
- Use the synopsys_full_case attribute to make sure that the //synopsys full_case directive is present.

- Use the synopsys_parallel_case attribute to make sure that the //synopsys parallel_case directive is present.
- Use the synopsys_full_parallel_case attribute to make sure that the //synopsys full_parallel_case directive is present.
- Use the signals_driven attribute to control the number of signals driven inside a case statement.
- Use the operand_size_match_no_carry attribute to control whether the left- and right-hand sides of a binary operation are the same size. This attribute differs from the operand_size_match attribute in that it does not take into account the carry bit from addition or subtraction. For example:

```
force operand_size_match_no_carry in continuous_asign

wire a, b, c;
assign a = b+c; // rule does not fire here
```

- Use the case_exp_size_exceeding attribute to issue a warning if the case expression size exceeds the value specified in the +max_case+<value> switch in the command line.

Casex Statement Template

The casex_statement template is a primary template belonging to the SEQUENTIAL_STATEMENT class. It contains the attributes shown in Table 23.

Table 23: casex_statement Template

Attribute	Kind	Limit_Kind
expression	template	EXPRESSION
default	local	N/A
null	local	N/A
blocking_assignment	template	blocking_assignment
case_statement	template	case_statement
casex_statement	template	casex_statement
casez_statement	template	casez_statement
conditional_statement	template	conditional_statement
disable_statement	template	disable_statement
event_trigger	template	event_trigger

Table 23: casex_statement Template (Continued)

Attribute	Kind	Limit_Kind
for_statement	template	for_statement
forever_statement	template	forever_statement
repeat_statement	template	repeat_statement
while_statement	template	while_statement
do_while_statement	template	do_while_statement
non_blocking_assignment	template	non_blocking_assignment
par_block	template	par_block
procedural_continuous_assign	template	procedural_continuous_assign
procedural_continuous_deassign	template	procedural_continuous_deassign
procedural_continuous_force	template	procedural_continuous_force
procedural_continuous_release	template	procedural_continuous_release
procedural_timing_control_statement	template	procedural_timing_control_statement
seq_block	template	seq_block
system_task_enable	template	system_task_enable
wait_statement	template	wait_statement
full_case	local	N/A
parallel_case	local	N/A
case_item	template	case_item
item_count	max/min	N/A
null_statement	template	null_statement
pragma	set	pragma_type
default_as_last	local	N/A
operand_size_match	local	N/A
operand_size_match_no_carry	local	N/A
duplicated_case_item	local	N/A

Table 23: casex_statement Template (Continued)

Attribute	Kind	Limit_Kind
event_control	template	event_control
signals_driven	max/min	N/A
statement_format	template	statement_format
redundant_case_item	local	N/A
process_statement	template	process_statement
unique	local	N/A
priority	local	N/A
expression	template	EXPRESSION
casexz_exp_size_exceeding	local	N/A

- Use the case_item attribute to control a case item expression and statement (including the default).
- Use the default attribute to control whether a default branch is present.
- Use the expression attribute to control casex expressions.
- Use the item_count attribute to control the number of case items (excluding the default item).
- Use the signals_driven attribute to control the number of signals driven inside a casex statement.
- Use the operand_size_match_no_carry attribute to control whether the left- and right-hand sides of a binary operation are the same size. This attribute differs from the operand_size_match attribute in that it does not take into account the carry bit from addition or subtraction. For example:

```
force operand_size_match_no_carry in continuous_asign

wire a, b, c;
assign a = b+c; // rule does not fire here
```

- Use the casexz_exp_size_exceeding attribute to issue a warning if the case expression size exceeds the value specified in the +max_casexz+<value> switch in the command line.

Casez Statement Template

The casez_statement template is a primary template belonging to the SEQUENTIAL_STATEMENT class. It contains the attributes shown in Table 24.

Table 24: casez_statement Template

Attribute	Kind	Limit_Kind
expression	template	EXPRESSION
default	local	N/A
null	local	N/A
blocking_assignment	template	blocking_assignment
case_statement	template	case_statement
casex_statement	template	casex_statement
casez_statement	template	casez_statement
conditional_statement	template	conditional_statement
disable_statement	template	disable_statement
event_trigger	template	event_trigger
for_statement	template	for_statement
forever_statement	template	forever_statement
repeat_statement	template	repeat_statement
while_statement	template	while_statement
do_while_statement	template	do_while_statement
non_blocking_assignment	template	non_blocking_assignment
par_block	template	par_block
procedural_continuous_assign	template	procedural_continuous_assign
procedural_continuous_deassign	template	procedural_continuous_deassign
procedural_continuous_force	template	procedural_continuous_force
procedural_continuous_release	template	procedural_continuous_release
procedural_timing_control_statement	template	procedural_timing_control_statement
seq_block	template	seq_block

Table 24: casez_statement Template (Continued)

Attribute	Kind	Limit_Kind
system_task_enable	template	system_task_enable
task_enable	template	task_enable
wait_statement	template	wait_statement
full_case	local	N/A
parallel_case	local	N/A
case_item	template	case_item
item_count	max/min	N/A
null_statement	template	null_statement
pragma	set	pragma_type
default_as_last	local	N/A
operand_size_match	local	N/A
operand_size_match_no_carry	local	N/A
duplicated_case_item	local	N/A
event_control	template	event_control
signals_driven	max/min	N/A
statement_format	template	statement_format
redundant_case_item	local	N/A
process_statement	template	process_statement
unique	local	N/A
priority	local	N/A
casexz_exp_size_exceeding	local	N/A

- Use the case_item attribute to control case item expressions and statements (including the default).
- Use the default attribute to control whether the default branch is present.
- Use the expression attribute to control casez expressions.

- Use the item_count attribute to control the number of case items (excluding the default item).
- Use the signals_driven attribute to control the number of signals driven inside a casez statement.
- Use the operand_size_match_no_carry attribute to control whether the left- and right-hand sides of a binary operation are the same size. This attribute differs from the operand_size_match attribute in that it does not take into account the carry bit from addition or subtraction. For example:

```
force operand_size_match_no_carry in continuous_asign

wire a, b, c;
assign a = b+c; // rule does not fire here
```

- Use the casexz_exp_size_exceeding attribute to issue a warning if the case expression size exceeds the value specified in the +max_casexz+<value> switch in the command line.

Char Declaration Template

The char_declaration template contains the attributes shown in Table 25.

Table 25: char_declaration Template

Attribute	Kind	Limit_Kind
identifier	template	ID
expression	template	EXPRESSION
signed	local	N/A
unsigned	local	N/A
static	local	N/A
automatic	local	N/A
unpacked_dimension	template	range
unpacked_dimension_count	max/min	N/A

Charge Strength Template

The LRM defines the grammar for charge strength as follows:

```
charge_strength ::= (small) | (medium) | (large)
```

The charge_strength template is a primary template belonging to no classes. It contains the attribute shown in **Table 26.**

Table 26: charge_strength Template

Attribute	Kind	Limit_Kind
charge_value	set	Charge_Strength_Type

Clock Template

The clock template does not correspond to any part of the formal definition in the Verilog language. Instead, you use the clock template to constrain hardware clocks inferred from the Verilog code.

The clock template is a primary template belonging to no classes. It contains the attributes shown in **Table 27**.

Table 27: clock Template

Attribute	Kind	Limit_Kind
edge	set	edge_type
expression	template	EXPRESSION
identifier	template	ID
object_definition	template	object_item
connectivity_path	template	connectivity_path
starting_unit	template	REGION_PART
gated_in_unit	template	ID
data	local	N/A
fixed_value	local	N/A

- Use the gated_in_unit attribute to control the placement of gated clocks with greater precision. This is basically a post-elaboration attribute.
- Use the data attribute to constrain the use of this signal as data.

- Use the starting_unit attribute to make the clock start from a given unit.
- Use the fixed_value attribute to detect clocks with fixed value inputs.

Cmos Switch Template

The LRM (§7.1) defines the grammar for cmos switch as follows:

```
cmos_switchtype [delay3] cmos_switch_instance {, cmos_switch_instance} ;

cmos_switch_instance ::= [name_of_gate_instance] (output_terminal,
                          input_terminal,
                          ncontrol_terminal,
                          pcontrol_terminal)

name_of_gate_instance ::= gate_instance_identifier [range]

input_terminal ::= scalar_expression

ncontrol_terminal ::= scalar_expression

pcontrol_terminal ::= scalar_expression

output_terminal ::= terminal_identifier | terminal_identifier
                      [constant_expression]

cmos_switchtype ::= cmos | rcmos
```

The cmos_switch template is a primary template belonging to the CONCURRENT_STATEMENT class. It contains the attributes shown in **Table 28**.

Table 28: cmos_switch Template

Attribute	Kind	Limit_Kind
cmos	local	N/A
delay2	template	delay2
delay3	template	delay3
delay_control	template	delay_control
identifier	template	ID
range	template	range
rcmos	local	N/A

Comment Template

The comment template is a primary template belonging to no classes. It contains the attributes shown in **Table 29**.

Table 29: comment Template

Attribute	Kind	Limit_Kind
block_comment	local	N/A
text	template	ID

- Use the block_comment attribute to control whether comments are denoted as in the C programming language. For example, the following rule:

```
no block_comment in comment
```

  causes Leda to flag an error for all comments of type:

```
/* something ...*/
```

- Use the text attribute to control the contents of comments. For example, the following rule:

```
template is comment
    limit text to "dc_shell"
end
no DC_SHELL_COMMENT in comment
```

  causes Leda to flag an error for the following Verilog code:

```
module test;
    //dc_shell ...
```

Concatenation Template

The LRM defines the grammar for concatenation as follows:

```
concatenation ::= {expression {, expression}}
```

The concatenation template is a primary template belonging to the EXPRESSION_NAME class. It contains the attributes shown in **Table 30.**

Table 30: concatenation Template

Attribute	Kind	Limit_Kind
expression	template	EXPRESSION
bit_length	max/min	N/A
evaluation_time	set	Evaluation_Periods
operand_size_match	local	N/A
operand_size_match_no_carry	local	N/A
unsized_element	local	N/A

- You can set the evaluation_time attribute to any of these enumerated literals:
  - unresolved
  - locally_static_evaluation
  - globally_static_evaluation
  - dynamic_evaluation

  For example:

  ```
  set evaluation_time to locally_static_evaluation
  ```
- Use the bit_length attribute to control the bit length of the concatenated expression. If Leda cannot evaluate the bit length of any element of the concatenated expression, the result is –1.
- Use the operand_size_match_no_carry attribute to control whether the left- and right-hand sides of a binary operation are the same size. This attribute differs from the operand_size_match attribute in that it does not take into account the carry bit from addition or subtraction. For example:

  ```
  force operand_size_match_no_carry in continuous_asign

  wire a, b, c;
  assign a = b+c; // rule does not fire here
  ```

- Use the unsized_element attribute to check if all elements in a concatenation have explicit sizes. It flags for unsized parameters and integers. This attribute makes an exception for the first element (MSB). For example:

```
module TEST ();
    parameter P_A1 = 7;
    integer i;
    wire [31:0] y1,y2;
    assign y1 = {32'b0,1};      // Fail, 1 is unsized
    assign y1 = {32'b0,P_A1};      // Fail, P_A1 is unsized
    assign y1 = {32'b0,P_A1,1'b0}; // Fail, P_A1 is unsized
    assign y1 = {P_A1,32'b0}; // OK, P_A1 is unsized but it is MSB
    assign y1 = {32'b0,i};         // Fail, i is unsized
    assign y1 = {32'b0,i,1'b0};    // Fail, i is unsized
    assign {y1,i} = 32'd0;         // Fail, i is unsized
    assign {y1,i,y2} = 32'd0;      // Fail, i is unsized
    assign {i,y1} = 32'd0;         // OK
endmodule
```

Conditional Compilation Directive Template

The LRM (§16.4) defines the grammar for text macro definition as follows:

```
undefine_compiler_directive ::= undef text_macro_name
```

The conditional_compilation_directive template is a primary template belonging to no classes. It contains the attribute shown in **Table 31**.

Table 31: conditional_compilation_directive Template

Attribute	Kind	Limit_Kind
text_macro_name	template	ID
enclosing_filename	template	ID

Use the text_macro_name attribute to control the name of the macro.

Conditional Expression Template

The LRM (§4.1.13) defines the grammar for conditional expression as follows:

```
conditional_expression ::= left_expression ? middle_expression :
                           right_expression
```

The conditional_expression template is a primary template belonging to the EXPRESSION class. It contains the attributes shown in Table 32.

Table 32: conditional_expression Template

Attribute	Kind	Limit_Kind
left_expression	template	EXPRESSION
middle_expression	template	EXPRESSION
right_expression	template	EXPRESSION
evaluation_time	set	evaluation_periods
bit_length	max/min	N/A
operand_size_match	local	N/A
operand_size_match_no_carry	local	N/A

- You can set the evaluation_time attribute to any of the following enumerated literals:
  - unresolved
  - locally_static_evaluation
  - globally_static_evaluation
  - dynamic_evaluation

  For example:

  ```
  set evaluation_time to locally_static_evaluation
  ```

- You can use the bit_length attribute to control the resulting bit length of the conditional expression. If Leda can evaluate the bit length of the middle and right expression elements, the resulting bit length is defined as:

  ```
  Max(Length(middle_expression),Length(right_expression))
  ```

  Otherwise, the result is –1.

- You can use the operand_size_match attribute to control whether the right and middle expressions are the same size. If Leda cannot evaluate the size of one of these elements, the conditional expression is considered to have an operand size match.

- You can use the operand_size_match_no_carry attribute to control whether the left- and right-hand sides of a binary operation are the same size. This attribute differs from the operand_size_match attribute in that it does not take into account the carry bit from addition or subtraction. For example:

```
force operand_size_match_no_carry in continuous_asign

wire a, b, c;
assign a = b+c; // rule does not fire here
```

Conditional Statement Template

The LRM (§9.4) defines the grammar for conditional statement as follows:

```
conditional_statement ::=[ unique_priority ] if ( expression )
statement_or_null [ else statement_or_null ]
       | if_else_if_statement
if_else_if_statement ::=[ unique_priority ] if ( expression )
statement_or_null
       { else [ unique_priority ] if ( expression ) statement_or_null }
[ else statement_or_null ]

event_expression ::=expression [ iff expression ]
       | hierarchical_identifier [ iff expression ]
       | [ edge ] expression [ iff expression ]
       | event_expression or event_expression
       | event_expression , event_expression
edge ::= posedge | negedge | changed

unique_priority ::= unique | priority
```

The conditional_statement template is a primary template belonging to the SEQUENTIAL_STATEMENT class. It contains the attributes shown in Table 33.

Table 33: conditional_statement Template

Attribute	Kind	Limit_Kind
blocking_assignment	template	blocking_assignment
case_statement	template	case_statement
casex_statement	template	casex_statement
casez_statement	template	casez_statement
conditional_statement	template	conditional_statement
disable_statement	template	disable_statement
else	local	N/A

Table 33: conditional_statement Template (Continued)

Attribute	Kind	Limit_Kind
event_trigger	template	event_trigger
expression	template	EXPRESSION
for_statement	template	for_statement
forever_statement	template	forever_statement
repeat_statement	template	repeat_statement
non_blocking_assignment	template	non_blocking_assignment
par_block	template	par_block
priority	local	N/A
procedural_continuous_assign	template	procedural_continuous_assign
procedural_continuous_deassign	template	procedural_continuous_deassign
procedural_continuous_force	template	procedural_continuous_force
procedural_continuous_release	template	procedural_continuous_release
procedural_timing_control_statement	template	procedural_timing_control_statement
process_statement	template	process_statement
seq_block	template	seq_block
system_task_enable	template	system_task_enable
task_enable	template	task_enable
unique	local	N/A
wait_statement	template	wait_statement
while_statement	template	while_statement
false_alt	template	SEQUENTIAL_STATEMENT
true_alt	template	SEQUENTIAL_STATEMENT

- Use the else attribute to control the presence of the else clause.
- Use the false_alt attribute to control the statement in the else branch.
- Use the true_alt attribute to control the statement in the true branch.

For example, if you only want to allow the use of the conditional statement that has the following syntax:

```
if (expression)
        begin
        …
        end
[else
        begin
        …
        end]
```

You can write this rule using the following templates and commands:

```
template IF_BLOCK is conditional_statement
        no else
        limit true_alt to seq_block
end
template IF_ELSE_BLOCK is conditional_statement
        force else
        limit true_alt to seq_block
        limit false_alt to seq_block
end
limit conditional_statement to IF_BLOCK, IF_ELSE_BLOCK
        message "Use begin-end blocks inside if statements"
        severity ERROR
```

Constant Declaration Template

The grammar for constant declaration is:

```
constant_declaration ::= const data_type const_assignment ;
```

The constant_declaration template is a primary template. It contains the attributes shown in Table 34.

Table 34: constant_declaration Template

Attribute	Kind	Limit_Kind
automatic	local	N/A
expression	template	EXPRESSION
integer_type	set	integer_type
non_integer_type	set	non_integer_type
type_declaration_identifier	template	ID
struct_union_member	template	struct_union_member

Table 34: constant_declaration Template (Continued)

Attribute	Kind	Limit_Kind
struct	local	N/A
union	local	N/A
enum	local	N/A
enum_identifier	template	ID
signed	local	N/A
static	local	N/A
unsigned	local	N/A
void	local	N/A
identifier	template	ID
packed_dimension	template	range
byte	local	N/A
char	local	N/A
shortint	local	N/A
int	local	N/A
longint	local	N/A
integer	local	N/A
bit	local	N/A
logic	local	N/A
reg	local	N/A
time	local	N/A
shortreal	local	N/A
real	local	N/A
realtime	local	N/A
data_type	template	DECLARATIVE_ITEM

Continuous Assign Template

The LRM (§6.1) defines the grammar for continuous assignment as follows:

```
continuous_assign ::= assign [drive_strength] [delay3]
                      list_of_net_assignments ;

list_of_net_assignments ::= net_assignment {, net_assignment}

net_assignment ::= net_1value = expression
```

The continuous_assign template is a primary template belonging to the CONCURRENT_STATEMENT class. It contains the attributes shown in Table 35.

Table 35: continuous_assign Template

Attribute	Kind	Limit_Kind
delay2	template	delay2
delay3	template	delay3
delay_control	template	delay_control
drive_strength	template	drive_strength
expression	template	EXPRESSION
implicit_net	template	OBJECT_ITEM
net_lvalue	template	EXPRESSION
overflow	local	N/A
operand_size_match	local	N/A
operand_size_match_no_carry	local	N/A
read_write	local	N/A
one_assignment_per_line	local	N/A
special_percentile_handle	local	N/A

- Use the overflow attribute to catch cases where the left-hand side of an assignment is smaller than the right-hand side. Note that the operand_size_match attribute applies to any expression, whereas the overflow attribute is just for assignments.
- Use the operand_size_match attribute to control whether the operands (right- and left-hand side) of any expression are the same size.

- If Leda cannot evaluate the size of any of the elements, the statement is considered to have an operand size match.
- Use the read_write attribute to control whether a same signal is present on both sides of the assignment.
- Use the operand_size_match_no_carry attribute to control whether the left- and right-hand sides of a binary operation are the same size. This attribute differs from the operand_size_match attribute in that it does not take into account the carry bit from addition or subtraction. For example:

```
force operand_size_match_no_carry in continuous_asign

wire a, b, c;
assign a = b+c; // rule does not fire here
```

Default Nettype Compiler Directive Template

The default_nettype_compiler_directive template is a primary template belonging to no classes. It contains the attribute shown in Table 36.

Table 36: default_nettype_compiler_directive Template

Attribute	Kind	Limit_Kind
enclosing_filename	template	ID

Delay Control Template

The LRM defines the grammar for delay control as follows:

```
delay_control ::= # delay_value
                  | # ( mintypmax_expression )

delay_value ::= unsigned_number |parameter_identifier
```

The delay_control template is a primary template belonging to no classes. It contains just the one attribute shown in **Table 37**.

Table 37: delay_control Template

Attribute	Kind	Limit_Kind
delay_value	template	EXPRESSION

Delay2 Template

The LRM defines the grammar for delay2 control as follows:

```
delay2 ::= | #(rise_delay_value, fall_delay_value)
```

The delay2 template is a primary template belonging to no classes. It contains the attributes shown in **Table 38**.

Table 38: delay2 Template

Attribute	Kind	Limit_Kind
rise_delay	template	EXPRESSION
fall_delay	template	EXPRESSION

Delay3 Template

The LRM defines the grammar for delay3 control as follows:

```
delay3 ::= |#(rise_delay_value, fall_delay_value, turn_off_delay_value)
```

The delay3 template is a primary template belonging to no classes. It contains the attributes shown in **Table 39.**

Table 39: delay3 Template

Attribute	Kind	Limit_Kind
rise_delay	template	EXPRESSION
fall_delay	template	EXPRESSION
turn_off_delay	template	EXPRESSION

Disable Statement Template

The LRM (§11) defines the grammar for disable statement as follows:

```
disable_statement ::= | disable task_identifier;
                      | disable block_identifier;
```

The disable_statement template is a primary template belonging to the SEQUENTIAL_STATEMENT class. It contains the attribute shown in **Table 40**.

Table 40: disable_statement Template

Attribute	Kind	Limit_Kind
identifier	template	ID

Drive Strength Template

The LRM defines the grammar for drive strength as follows:

```
drive_strength ::= (strength0 , strength1)
                  |(strength1 , strength0)
                  |(strength0 , highz1)
                  |(strength1 , highz0)
                  |(highz1, strength0)
                  |(highz0, strength1)

strength0 ::= supply0 | strong0 | pull0 | weak0

strength1 ::= supply1 | strong1 | pull1 | weak1
```

The drive_strength template is a primary template belonging to no classes.

Do While Statement Template

The do_while template contains the attributes shown in Table 41.

Table 41: do_while_statement Template

Attribute	Kind	Limit_Kind
blocking_assignment	template	blocking_assignment
case_statement	template	case_statement
casex_statement	template	casex_statement
casez_statement	template	casez_statement
conditional_statement	template	conditional_statement
disable_statement	template	disable_statement
event_control	template	event_control
event_trigger	template	event_trigger
expression	template	EXPRESSION
for_statement	template	for_statement
forever_statement	template	forever_statement
non_blocking_assignment	template	non_blocking_assignment
par_block	template	par_block
procedural_continuous_assign	template	procedural_continuous_assign

Table 41: do_while_statement Template (Continued)

Attribute	Kind	Limit_Kind
procedural_continuous_deassign	template	procedural_continuous_deassign
procedural_continuous_force	template	procedural_continuous_force
procedural_continuous_release	template	procedural_continuous_release
procedural_timing_control_statement	template	procedural_timing_control_statement
repeat_statement	template	repeat_statement
seq_block	template	seq_block
statement	template	SEQUENTIAL_STATEMENT
system_task_enable	template	system_task_enable
task_enable	template	task_enable
do_while_statement	template	do_while_statement
wait_statement	template	wait_statement
while_statement	template	while_statement

Enable Gate Template

The LRM (§7.1) defines the grammar for enable gate as follows:

```
enable_gatetype [drive_strength] [delay3] enable_gate_instance
{, enable_gate_instance} ;

enable_gate_instance ::= [name_of_gate_instance] (output_terminal,
                     input_terminal,
                     enable_terminal)

name_of_gate_instance ::= gate_instance_identifier [range]

input_terminal ::= scalar_expression

enable_terminal ::= scalar_expression

output_terminal ::= terminal_identifier | terminal_identifier
                    [constant_expression]

enable_gatetype ::= bufif0 | bufif1 | notif0 | notif1
```

The enable_gate template is a primary template belonging to the CONCURRENT_STATEMENT class. It contains the attributes shown in **Table 42**.

Table 42: enable_gate Template

Attribute	Kind	Limit_Kind
bufif0	local	N/A
bufif1	local	N/A
delay2	template	delay2
delay3	template	delay3
delay_control	template	delay_control
identifier	template	ID
notif0	local	N/A
notif1	local	N/A
range	template	range

End Comment Template

The end_comment template is a secondary template belonging to no classes. The end_comment template is a primary template belonging to the CONCURRENT_STATEMENT class. It contains the attribute shown in **Table 43**.

Table 43: end_comment Template

Attribute	Kind	Limit_Kind
text	template	ID

Enum Declaration Template

The enum_declaration template contains the attributes shown in Table 44.

Table 44: enum_declaration Template

Attribute	Kind	Limit_Kind
static	local	N/A
automatic	local	N/A
signed	local	N/A

Table 44: enum_declaration Template (Continued)

Attribute	Kind	Limit_Kind
unsigned	local	N/A
enum_member	template	enum_member
member_count	Max/Min	N/A
byte	local	N/A
char	local	N/A
shortint	local	N/A
int	local	N/A
longint	local	N/A
integer	local	N/A
bit	local	N/A
logic	local	N/A
packed	local	N/A
packed_dimension	template	range
reg	local	N/A

Enum Member Template

The enum_member template contains the attributes shown in Table 45.

Table 45: enum_member Template

Attribute	Kind	Limit_Kind
identifier	template	ID
expression	template	EXPRESSION

Event Control Template

The LRM (§9.6) defines the grammar for event control as follows:

```
event_control ::=@ event_identifier
        | @ ( event_expression )
        | @*
        | @ (*)
event_expression ::= expression
                   | event_identifier
                   | posedge expression
                   | negedge expression
                   | event_expression or event_expression
```

The event_control template is a primary template belonging to no classes. It contains the attributes shown in Table 46.

Table 46: event_control Template

Attribute	Kind	Limit_Kind
comma	local	N/A
expression	template	EXPRESSION
expression_count	max/min	N/A
iff_expression	template	EXPRESSION
identifier	template	ID
posedge	local	N/A
negedge	local	N/A
or	local	N/A
star	local	N/A

- Use the expression_count attribute to control the total number of expressions.
- Use the or attribute to control whether the event_expression has multiple event expressions.
- Use the posedge attribute to control whether at least one event expression is preceded by the keyword posedge.
- Use the negedge attribute to control whether at least one event expression is preceded by the keyword negedge.

Event Declaration Template

The LRM (§9.7.3) defines the grammar for event declaration as follows:

```
event_declaration ::= event event_identifier {, event_identifier}
```

The event_declaration template is a primary template belonging to the OBJECT_ITEM class. It contains the attributes shown in **Table 47**.

Table 47: event_declaration Template

Attribute	Kind	Limit_Kind
comment	template	comment
declarative_region	template	REGION
identifier	template	ID
is_initialization	local	N/A
is_load	local	N/A
is_set	local	N/A
one_declaration_per_line	local	N/A

- Use the comment attribute to control the comment associated with a variable declaration.
- Use the declarative_region attribute to control the region where the variable is declared.
- Use the one_declaration_per_line attribute to control whether multiple variables are declared on the same line.

Event Trigger Template

The LRM (§9.7.3) defines the grammar for event trigger as follows:

```
event_trigger ::= -> event_identifier ;
```

The event_trigger template is a primary template belonging to the SEQUENTIAL_STATEMENT class. It contains the attribute shown in Table 48.

Table 48: event_trigger Template

Attribute	Kind	Limit_Kind
identifier	template	ID

File Layout Template

The file_layout template is a primary template belonging to no classes. It contains the attributes shown in **Table 49**.

Table 49: file_layout Template

Attribute	Kind	Limit_Kind
unit_count	max/min	N/A
header_comment	template	header_comment
characters_per_line	max/min	N/A

Use the unit_count attribute to control the number of units allowed in a file.

Use the header_comment attribute to control the appearance of header comments at the start of the file.

Use the characters_per_line attribute to control the number of characters allowed per line of code.

Flipflop Template

The flipflop template does not correspond to any part of the formal definition of the Verilog language. Instead, you use this template to constrain hardware flip-flops inferred from the Verilog code.

The flipflop template is a primary template belonging to no classes. It contains the attributes shown in **Table 50**.

Table 50: flipflop Template

Attribute	Kind	Limit_Kind
asynchronous_initialization	template	asynchronous_initialization
synchronous_initialization	template	synchronous_initialization
identifier	template	ID
input	template	EXPRESSION
data_signal	template	data_signal
has_clock_as_data	local	N/A
synchronous_reset_signal	local	N/A

Table 50: flipflop Template (Continued)

Attribute	Kind	Limit_Kind
synchronous_set_signal	local	N/A
synchronous_load_signal	local	N/A
asynchronous_reset_signal	local	N/A
asynchronous_set_signal	local	N/A
asynchronous_load_signal	local	N/A

- Use the identifier attribute to control the names of flip-flops inferred in the code.
- Use the synchronus_reset_signal, synchronus_set_signal, synchronus_load_signal attributes with min/max/force/no keywords to explicitly control the presence and the number of reset/set/load in a flipflop.

For Statement Template

The for_statement template is a primary template belonging to the SEQUENTIAL_STATEMENT class. It contains the attributes shown in Table 51.

Table 51: for_statement Template

Attribute	Kind	Limit_Kind
blocking_assignment	template	blocking_assignment
case_statement	template	case_statement
casex_statement	template	casex_statement
casez_statement	template	casez_statement
conditional_statement	template	conditional_statement
disable_statement	template	disable_statement
event_trigger	template	event_trigger
expression	template	EXPRESSION
for_statement	template	for_statement
forever_statement	template	forever_statement
initial_reg_assignment	template	reg_assignment

Table 51: for_statement Template (Continued)

Attribute	Kind	Limit_Kind
initial_signal_in_condition	local	N/A
initial_signal_in_step	local	N/A
initialize_iterator	local	N/A
iterator_modification	local	N/A
non_blocking_assignment	template	non_blocking_assignment
null_statement	template	null_statement
par_block	template	par_block
procedural_continuous_assign	template	procedural_continuous_assign
procedural_continuous_deassign	template	procedural_continuous_deassign
procedural_continuous_force	template	procedural_continuous_force
procedural_continuous_release	template	procedural_continuous_release
procedural_timing_control_statement	template	procedural_timing_control_statement
process_statement	template	process_statement
repeat_statement	template	repeat_statement
seq_block	template	seq_block
statement	template	SEQUENTIAL_STATEMENT
system_task_enable	template	system_task_enable
step_reg_assignment	template	reg_assignment
step_signal_in_condition	local	N/A
task_enable	template	task_enable
wait_statement	template	wait_statement
while_statement	template	while_statement

- Use the initial_reg_assignment attribute to control the left-hand reg_assignment.
- Use the inialize_iterator attribute to detect whether the loop iterator is initialized in the initial assignment of the for statement. For example:

```
for (i=1; j<10; j=j+1)//j is not initialized
A[j]=B[j-1]
```

- Use the iterator_modification attribute to detect if the loop variable is modified inside the for_statement. For example:

```
for (i=1; i<10; i=i+1) begin
A[i]=B[i-1]
i=i+1 ;// i is modified
end
```

- Use the step_reg_assignment attribute to control the right-hand reg_assignment.
- Use the initial_signal_in_condition attribute to detect if the initial expression is present in the conditional expression of a for-loop statement.
- Use the initial_signal_in_step attribute to detect if the initial expression must be present in the step expression of a for-loop statement.
- Use the step_signal_in_condition attribute to detect if the step expression must be present in the conditional expression of a for-loop statement.

Forever Statement Template

The LRM (§9.6) defines the grammar for looping statement as follows:

```
looping_statement ::= forever statement
                    | repeat (expression ) statement
                    | while (expression ) statement
                    | for ( reg_assignment ; expression ; reg_assignment )
                    statement
```

In VeRSL, each type of looping statement has a corresponding template. The forever_statement template is a primary template belonging to the SEQUENTIIAL_STATEMENT class. It contains the attributes shown in Table 52.

Table 52: forever_statement Template

Attribute	Kind	Limit_Kind
blocking_assignment	template	blocking_assignment
case_statement	template	case_statement
casex_statement	template	casex_statement
casez_statement	template	casez_statement

Table 52: forever_statement Template (Continued)

Attribute	Kind	Limit_Kind
conditional_statement	template	conditional_statement
disable_statement	template	disable_statement
event_trigger	template	event_trigger
for_statement	template	for_statement
forever_statement	template	forever_statement
non_blocking_assignment	template	non_blocking_assignment
par_block	template	par_block
procedural_continuous_assign	template	procedural_continuous_assign
procedural_continuous_deassign	template	procedural_continuous_deassign
procedural_continuous_force	template	procedural_continuous_force
procedural_continuous_release	template	procedural_continuous_release
procedural_timing_control_statement	template	procedural_timing_control_statement
process_statement	template	process_statement
repeat_statement	template	repeat_statement
seq_block	template	seq_block
statement	template	SEQUENTIAL_STATEMENT
system_task_enable	template	system_task_enable
task_enable	template	task_enable
wait_statement	template	wait_statement
while_statement	template	while_statement

FSM Template

Leda can infer finite state machines (FSMs) from a design and apply rules to constrain the type of FSM accepted. Leda can identify FSMs that are written using case statements to define the states and transitions. Leda cannot recognize FSMs defined using if statements. Leda recognizes 1-process, 2-process and 3-process FSMs if all processes are in the same block (architecture or module).

The fsm template is a primary template belonging to no classes. It contains the attributes shown in **Table 53**.

Table 53: fsm Template

Attribute	Kind	Limit_Kind
block_count	max/min	N/A
mealy	local	N/A
moore	local	N/A
state_count	max/min	N/A
state_variable	template	OBJECT_ITEM
transition_in_default	local	N/A

- Use the state_count attribute to constrain the number of states in an FSM. Leda only counts reachable states. The number of states is therefore not the number of different values that the state variable can take. You can also use this attribute to enforce that the number of states is a power of 2 (for example).
- Use the state_variable attribute to constrain the properties of the state variable (for example, its name and type).
- Use the block_count attribute to constrain the number of process or always blocks used to infer an FSM. The typical templates are for 1-, 2- or 3-block FSMs.
- Use the moore attribute to constrain FSMs to Moore-style. You can constrain FSMs to Moore style using the following VeRSL rule:

```
force moore in fsm
```

- Use the mealy attribute to constrain FSMs to Mealy-style. You can constrain FSMs to Mealy style using the following VeRSL rule:

```
force mealy in fsm
```

- Use the transition_in_default attribute to enforce the use of a default clause in the case statement (even if one isn’t necessary).

  You can find a set of prepackaged FSM rules in the STATE_MACHINES ruleset of the Leda policy. For more information, see the *Leda General Coding Rules Guide*.

Function Call Template

The LRM (§10.3.3) defines the grammar for function call as follows:

```
function_call ::= function_identifier (expression {, expression})
```

The function_call template is a primary template belonging to the EXPRESSION class. It contains the attributes shown in **Table 54**.

Table 54: function_call Template

Attribute	Kind	Limit_Kind
identifier	template	ID
expression	template	EXPRESSION
expression_count	max/min	N/A
operand_size_match	local	N/A
operand_size_match_no_carry	local	N/A
bit_length	max/min	N/A
port_connection	template	port_connection
special_percentile_handle	local	N/A

- Use the expression_count attribute to control the number of expressions in a function call.
- Use the port_connection attribute to control the port association in a function call. See also the “Port Connection Template” on page 182.
- Use the operand_size_match_no_carry attribute to control whether the left- and right-hand sides of a binary operation are the same size. This attribute differs from the operand_size_match attribute in that it does not take into account the carry bit from addition or subtraction. For example:

```
force operand_size_match_no_carry in continuous_asign

wire a, b, c;
assign a = b+c; // rule does not fire here
```

- Use the bit_length attribute to control the bit length of the function call. If Leda cannot evaluate the length, the result is -1.

Function Declaration Template

```
function_declaration ::=function [ automatic ] [ signing ]
[range_or_type ] function_identifier ;
       { function_item_declaration }
       { function_statement }
       endfunction
       | function [ automatic ] [ signing ] [ range_or_type ]
       function_identifier ( function_port_list ) ;
       { block_item_declaration }
       { function_statement }
       endfunction
function_item_declaration ::=block_item_declaration
       | { attribute_instance } input_declaration ;
       | { attribute_instance } output_declaration ;
       | { attribute_instance } inout_declaration ;
function_port_item ::={ attribute_instance } input_declaration
       | { attribute_instance } output_declaration
       | { attribute_instance } inout_declaration
function_port_list ::= function_port_item { , function_port_item }
```

The function_declaration template is a primary template belonging to the OBJECT_ITEM class. It contains the attributes shown in **Table 55**.

Table 55: function_declaration Template

Attribute	Kind	Limit_Kind
automatic	local	N/A
void	local	N/A
integer	local	N/A
range	template	range
real	local	N/A
realtime	local	N/A
time	local	N/A
end_comment	limit	end_comment
identifier	template	ID
input_count	max/min	N/A
input_declaration	template	input_declaration
is_partly_assigned	local	N/A

Table 55: function_declaration Template (Continued)

Attribute	Kind	Limit_Kind
parameter_declaration	template	parameter_declaration
reg_declaration	template	reg_declaration
integer_declaration	template	integer_declaration
real_declaration	template	real_declaration
time_declaration	template	time_declaration
realtime_declaration	template	realtime_declaration
event_declaration	template	event_declaration
blocking_assignment	template	blocking_assignment
case_statement	template	case_statement
casex_statement	template	casex_statement
casez_statement	template	casez_statement
conditional_statement	template	conditional_statement
disable_statement	template	disable_statement
event_trigger	template	event_trigger
for_statement	template	for_statement
forever_statement	template	forever_statement
global_signals_read	local	N/A
non_blocking_assignment	template	non_blocking_assignment
par_block	template	par_block
procedural_continuous_assign	template	procedural_continuous_assign
procedural_continuous_deassign	template	procedural_continuous_deassign
procedural_continuous_force	template	procedural_continuous_force
procedural_continuous_release	template	procedural_continuous_release
procedural_timing_control_statement	template	procedural_timing_control_statement
process_statement	template	process_statement

Table 55: function_declaration Template (Continued)

Attribute	Kind	Limit_Kind
repeat_statement	template	repeat_statement
return_last	local	N/A
signed	local	N/A
unsigned	local	N/A
seq_block	template	seq_block
system_task_enable	template	system_task_enable
task_enable	template	task_enable
wait_statement	template	wait_statement
while_statement	template	while_statement
side_effect	local	N/A
unused_declaration	local	DECLARATIVE_ITEM
global_signals_read	local	N/A
header_comment	template	header_comment
statement_format	template	statement_format
return_fully_assigned	local	N/A
recursion_type	local	N/A

- Use the integer, real, realtime, and time attributes to correspond to their respective types of functions. For example, the command:

  ```
  no real in net_declaration severity error
  ```

  means that functions of type real are not allowed.
- Use the global_signals_read attribute to prohibit global variables from being read inside a function.
- Use the return_last attribute to ensure that the last assignment in a function body is for the return variable.
- Use the side_effect attribute to control whether writing to variables declared outside a function is allowed.

- Use the unused_declaration attribute to control if a function should contain unused variables (for example, inputs and regs). You can also use this attribute with a VeRSL limit command to control the type of the unused variable. For example:

```
no unused_declaration in function_declaration
limit unused_declaration to integer_declaration
```

- Use the is_partly_assigned attribute to detect if the function name is partly assigned in the function body. For example:

```
function [15:0] f3; //FAIL
    input [15:0] inp;
    begin
          f3[7:0] = inp[7:0];
    end
endfunction
```

```
template FULLY_ASSIGNED_FUNC is function_declaration
    no is_partly_assigned
end
limit function_declaration to FULLY_ASSIGNED_FUNC
    message "Last function statement does not assign to all the bits
of the function"
    severity WARNING
```

Example:

```
module m_no_end_comment (a, y); // FAIL
input a;
output y;
reg y;
endmodule
```

```
force end_comment in module_declaration
message "No comment after 'end...' keyword "
severity WARNING
```

- Use the return_fully_assigned in function declaration to ensure that there is a return statement on every possible flow of control through the function. For example:

```
force return_fully_assigned in function_declaration

--Fail case
function test; //FAIL
    input D_IN, RST;
    if ( RST )
          test = D_IN;
endfunction
```

```
force return_fully_assigned in function_declaration
message “No value returned for a particular flow through the
function”
```

```
severity WARNING
```

- Use the recursion_type attribute to checks the type of recursion of function call in function declaration. There are 3 recursion types: no_recursion, static_recursion, all_recursion.

Header Comment Template

You can use the header_comment template to control the comments that come before a Verilog module, unit, or task. You can also use this template to make sure each field is present.

The header_comment template is a secondary template belonging to no classes. It contains the attribute shown in Table 118.

Table 56: header_comment Template

Attribute	Kind	Limit_Kind
comment_line	template	ID

Use the comment_line attribute to specify fields that must be present in a header comment. For example, if you want comments to look like the following:

```
// unit : SimDecl
// author : Me
// date : November 2001
```

you could write the following VeRSL rules to enforce that discipline:

```
template UNIT is header_comment
    limit comment_line to "unit :"
end
template AUTHOR is header_comment
    limit comment_line to "author :"
end
template DATE is header_comment
limit comment_line to "date :"
end

RULE1:
limit header_comment in module_declaration to allof
    UNIT message "Unit field is missing from header comment",
    AUTHOR message "Author field is missing from header comment",
    DATE message "Date field is missing from header comment

RULE2:
force header_comment in module_declaration
    message "Header comments are missing for module declaration"
    severity ERROR
```

Identifier Template

You can constrain all identifiers using the identifier template. This includes names in expressions, module names, file names, and any other character string you define.

The identifier template is a secondary template belonging to the ID class. It contains the attributes shown in **Table 57.**

Table 57: identifier Template

Attribute	Kind	Limit_Kind
error_id	template	ID
limit_id	template	ID
character_count	max/min	N/A

- Use the limit_id attribute with character strings, as shown in the following example:

```
limit limit_id to "^module_rtl$"
```

- Use the error_id attribute to define character strings that you don't want used in the identifier. For example, the following VeRSL rule causes Leda to flag an error if it finds a reg declaration named BAD_REG_NAME.

```
template BAD_REG_ID is identifier
    limit error_id to "BAD_REG_NAME"
end

limit identifier in reg_declaration to BAD_REG_ID
severity error
```

- Use the character_count attribute to define the maximum or minimum number of characters allowed in an identifier.

Include Compiler Directive Template

The LRM (§16.3) defines the grammar for text macro_definition as follows:

```
include_compiler_directive ::= include "filename"
```

The include_compiler_directive template is a primary template belonging to no classes. It contains the attribute shown in **Table 58**.

Table 58: include_compiler_directive Template

Attribute	Kind	Limit_Kind
file_name	template	ID
enclosing_filename	template	ID

- Use the file_name attribute to control the name of the include file. For example, the following VeRSL rule detects whether files with full paths are included:

```
template INC_FILE_NAME is identifier
    limit error_id to "^/"
end
limit file_name in include_compiler_directive to INC_FILE_NAME
    message "Do not use / at the beginning of an included file name"
    severity ERROR
```

Initial Construct Template

The LRM (§9.9.1) defines the grammar for initial construct as follows:

```
initial_construct ::= initial statement
```

The initial_construct template is a primary template belonging to the CONCURRENT_STATEMENT class. It contains the attributes shown in Table 59.

Table 59: initial_construct Template

Attribute	Kind	Limit_Kind
blocking_assignment	template	blocking_assignment
case_statement	template	case_statement
casex_statement	template	casex_statement
casez_statement	template	casez_statement
conditional_statement	template	conditional_statement
disable_statement	template	disable_statement
event_trigger	template	event_trigger
flipflop	template	flipflop
for_statement	template	for_statement
forever_statement	template	forever_statement
input_count	max/min	N/A
non_blocking_assignment	template	non_blocking_assignment
par_block	template	par_block
procedural_continuous_assign	template	procedural_continuous_assign
procedural_continuous_deassign	template	procedural_continuous_deassign
procedural_continuous_force	template	procedural_continuous_force
procedural_continuous_release	template	procedural_continuous_release
procedural_timing_control_statement	template	procedural_timing_control_statement
process_statement	template	process_statement
repeat_statement	template	repeat_statement
seq_block	template	seq_block

Table 59: initial_construct Template (Continued)

Attribute	Kind	Limit_Kind
statement	template	SEQUENTIAL_STATEMENT
system_task_enable	template	system_task_enable
task_enable	template	task_enable
wait_statement	template	wait_statement
while_statement	template	while_statement

Inout Declaration Template

The LRM (§12.3.2) defines the grammar for inout declaration as follows:

```
inout_declaration ::= inout [range] list_of_port_identifiers ;
```

The inout_declaration template is a primary template belonging to the OBJECT_ITEM_DRIVEN_OBJECT class. It contains the attributes shown in **Table 60**.

Table 60: inout_declaration Template

Attribute	Kind	Limit_Kind
ansic_style	local	N/A
combined	local	N/A
comment	template	comment
consistent_range	local	N/A
declarative_region	template	REGION
driver_declaration	template	DRIVER_OBJECT
driving_expression	template	EXPRESSION
expression	template	EXPRESSION
is_initialization	local	N/A
is_load	local	N/A
is_reset	local	N/A
is_read	local	N/A
is_set	local	N/A

Table 60: inout_declaration Template (Continued)

Attribute	Kind	Limit_Kind
identifier	template	ID
one_declaration_per_line	local	N/A
range	template	range
signals_driven	template	DRIVEN_OBJECT
tristate	local	N/A
outputs_driven	max/min	N/A
unpacked_dimension_count	max/min	N/A

- Use the comment attribute to control the comments associated with variable declarations.
- Use the consistent_range attribute to control whether the range specification of the inout remains unchanged if the inout port is redeclared as another variable. For example:

```
inout [3:0] clk_en;
reg [2:0] clk_en;
```

  In this example, the following VeRSL command causes Leda to flag an error:

```
force consistent_range in inout_declaration
```

- Use the declarative_region attribute to control the regions where variables are declared.
- Use the driver_declaration attribute to control the second declaration of a port. For example:

```
inout [3:0] clk_en;
wire [3:0] clk_en;
```

  In this example, the following VeRSL command causes Leda to flag an error:

```
limit driver_declaration in inout_declaration to reg_declaration
```

- Use the driving_expression attribute to control the expressions accepted as drivers for the current variable.
- Use the one_declaration_per_line attribute to control whether multiple variables are declared on the same line.
- Use the signals_driven attribute to control the signals driven by the current variable.
- Use the tristate attribute to control if a port is tristated.

- Use the is_read attribute to make sure all bits of the signal, variable, or port specified are read at least once somewhere in the Verilog code. When you force this attribute, Leda verifies that the signal is read. For example:

```
force is_read in inout_declaration
```

Input Declaration Template

The LRM (§12.3.2) defines the grammar for input declaration as follows:

```
input_declaration ::= input [range] list_of_port_identifiers ;
```

The input_declaration template is a primary template belonging to the OBJECT_ITEM_DRIVEN_OBJECT class. It contains the attributes shown in **Table 61**.

Table 61: input_declaration Template

Attribute	Kind	Limit_Kind
comment	template	comment
consistent_range	local	N/A
declarative_region	template	REGION
driver_declaration	template	DRIVER_OBJECT
driving_expression	template	EXPRESSION
identifier	template	ID
is_default	local	N/A
is_initialization	local	N/A
is_load	local	N/A
is_read	local	N/A
is_reset	local	N/A
is_set	local	N/A
one_declaration_per_line	local	N/A
range	template	range
signals_driven	template	DRIVEN_OBJECT
tristate	local	N/A

Table 61: input_declaration Template (Continued)

Attribute	Kind	Limit_Kind
outputs_driven	max/min	N/A
partly_used	local	N/A

- Use the comment attribute to control the comments associated with variable declarations.
- Use the consistent_range attribute to control whether the range specification for the input remains unchanged if the input port is redeclared as another variable. For example:

```
input [3:0] clk_en;
reg [2:0] clk_en;
```

  In this example, the following VeRSL command causes Leda to flag an error:

```
force consistent_range in input_declaration
```

- Use the declarative_region attribute to control the region where the variable is declared.
- Use the driver_declaration attribute to control the second declaration of the port. For example:

```
input [3:0] clk_en;
wire [3:0] clk_en;
```

  In this example, the following VeRSL command causes Leda to flag an error:

```
limit driver_declaration in input_declaration to reg_declaration
```

- Use the driving_expression attribute to control the expressions accepted as drivers for the current variable.
- Use the one_declaration_per_line attribute to control whether multiple variables are declared on the same line.
- Use the signals_driven attribute to control the signals driven by the current variable.
- Use the tristate attribute to control whether the port should be tristated.
- Use the is_read attribute to make sure all bits of the signal, variable, or port specified are read at least once somewhere in the Verilog code. When you force this attribute with a VeRSL rule:

```
force is_read in input_declaration
```

  Leda verifies that the signal is read, and flags an error if it is not.

- Use the outputs_driven attribute to control the number of directly driven outputs. For example, in the following code the number of outputs driven (feedthrough) by d1 is 1, whereas the number driven by d2 is 0.

```
output q2;
reg q2;
input d1, d2;
...
q1<=d1; //d1 drives port q1 directly
if (sel) //infers mux
    q2<=d2; //d2 is input to mux driving output q2
else
    q2<=1;
```

- Use the partly_used attribute to detect if the vector is partially used (only some bits used). For example:

```
no partly_used in input_declaration
message "Not all the bits of the vector are used"
severity WARNING

module w498 (in,out);
input [3:0] in; //FAIL
output [2:0] out;
wire [3:0] in;

assign out[1:0] = in[1:0];

endmdule
```

Int Declaration Template

The int_declaration template contains the attributes shown in Table 62.

Table 62: int_declaration Template

Attribute	Kind	Limit_Kind
identifier	template	ID
expression	template	EXPRESSION
signed	local	N/A
unsigned	local	N/A
static	local	N/A
automatic	local	N/A
unpacked_dimension	template	range
unpacked_dimension_count	max/min	N/A

Integer Declaration Template

The LRM (§3.9) defines the grammar for integer declaration as follows:

```
integer_declaration ::= integer list_of_register_identifiers ;

list_of_register_identifiers ::= register_ name {, register _name}

register_name ::= register_identifier
                | memory_identifier [ upper_limit_constant_expression :
                lower_limit_constant_expression ]
```

The integer_declaration template is a primary template belonging to the OBJECT_ITEM class. It contains the attributes shown in **Table 63.**

Table 63: integer_declaration Template

Attribute	Kind	Limit_Kind
dimension_count	max/min	N/A
comment	template	comment
declarative_region	template	REGION
identifier	template	ID
is_initialization	local	N/A
is_load	local	N/A
is_read	local	N/A
is_reset	local	N/A
is_set	local	N/A
memory_range	template	range
driving_expression	template	EXPRESSION
one_declaration_per_line	local	N/A
signals_driven	template	DRIVEN_OBJECT

- Use the comment attribute to control the comments associated with variable declarations.
- Use the declarative_region attribute to control the region where the variable is declared.

- Use the driving_expression attribute to control the expressions accepted as drivers for the current variable.
- Use the one_declaration_per_line attribute to control whether multiple variables are declared on the same line.
- Use the is_read attribute to make sure all bits of the signal, variable, or port specified are read at least once somewhere in the Verilog code. When you force this attribute with a VeRSL rule:

  ```
  force is_read in input_declaration
  ```

  Leda verifies that the signal is read, and flags an error if it is not.
- Use the signals_driven attribute to control the signals driven by the current variable.

Interface Declaration Template

The interface_declaration template contains the attributes shown in Table 64.

Table 64: interface_declaration Template

Attribute	Kind	Limit_Kind
automatic	local	N/A
identifier	template	ID
interface_declaration	template	interface_declaration
parameter_port_count	max/min	N/A
time_units_declaration	template	time_units_declaration
modport_declaration	template	modport_declaration
specparam_declaration	template	specparam_declaration
local_parameter_declaration	template	local_parameter_declaration
extern_task_declaration	template	extern_task_declaration
interface_instantiation	template	interface_instantiation
variable_declaration	template	variable_declaration
constant_declaration	template	constant_declaration
type_declaration	template	type_declaration
byte_declaration	template	byte_declaration

Table 64: interface_declaration Template (Continued)

Attribute	Kind	Limit_Kind
char_declaration	template	char_declaration
shortint_declaration	template	shortint_declaration
int_declaration	template	int_declaration
longint_declaration	template	longint_declaration
integer_declaration	template	integer_declaration
bit_declaration	template	bit_declaration
logic_declaration	template	logic_declaration
reg_declaration	template	reg_declaration
time_declaration	template	time_declaration
shortreal_declaration	template	shortreal_declaration
real_declaration	template	real_declaration
realtime_declaration	template	realtime_declaration
void_declaration	template	void_declaration
net_declaration	template	net_declaration
initial_construct	template	initial_construct
always_construct	template	always_construct
continuous_assign	template	continuous_assign
module_instantiation	template	module_instantiation
n_input_gate	template	n_input_gate
n_output_gate	template	n_output_gate
enable_gate	template	enable_gate
mos_switch	template	mos_switch
pass_switch	template	pass_switch
pass_en_switch	template	pass_en_switch
cmos_switch	template	cmos_switch

Table 64: interface_declaration Template (Continued)

Attribute	Kind	Limit_Kind
pull_gate	template	pull_gate
parameter_override	template	parameter_override
udp_instantiation	template	udp_instantiation
task_declaration	template	task_declaration
function_declaration	template	function_declaration
input_count	max/min	N/A
input_declaration	template	input_declaration
output_declaration	template	output_declaration
inout_declaration	template	inout_declaration
genvar_declaration	template	genvar_declaration
parameter_declaration	template	parameter_declaration

- Use the automatic attribute to check the use of automatic interfaces. For example:

```
no automatic in interface_declaration
```

Interface Port Declaration Template

The grammar for interface_port_declaration is:

```
interface_port_declaration ::=interface list_of_interface_identifiers
        | interface . modport_identifier list_of_interface_identifiers
        | identifier list_of_interface_identifiers
        | identifier . modport_identifier list_of_interface_identifiers
```

.The interface_port_declaration template contains the attributes shown in Table 65.

Table 65: interface_port_declaration Template

Attribute	Kind	Limit_Kind
asynchronous_initialization	limit	asynchronous_initialization
generic	local	N/A
is_initialization	local	N/A

Table 65: interface_port_declaration Template (Continued)

Attribute	Kind	Limit_Kind
is_load	local	N/A
is_reset	local	N/A
is_set	local	N/A

Jump Statement Template

The grammar for jump_statement is:

```
jump_statement ::= return [ expression ] ;
        | break ;
        | continue ;
```

The jump_statement template contains the attributes shown in Table 66.

Table 66: jump_statement Template

Attribute	Kind	Limit_Kind
expression	template	EXPRESSION
return	local	N/A
break	local	N/A
continue	local	N/A

Latch Template

The latch template does not correspond to any part of the formal definition of the Verilog language. Instead, you use this template to constrain hardware latches inferred from the Verilog code.

The latch template is a primary template belonging to no classes. It contains the attributes shown in **Table 67**.

Table 67: latch Template

Attribute	Kind	Limit_Kind
asynchronous_initialization	template	asynchronous_initialization
synchronous_initialization	local	synchronous_initialization
identifier	template	ID

Table 67: latch Template

Attribute	Kind	Limit_Kind
input	template	EXPRESSION
data_signal	template	data_signal
has_clock_as_data	local	N/A

Use the identifier attribute to control the names of latches inferred in the code. For example, if you want all latches to have an identifier with the suffix _latch, you can write the following VeRSL rule:

```
template LATCH_ID is latch
       limit identifier to "_latch$"
end
```

Use the input attribute to control the expressions driving a latch. You can point this attribute at any template in the EXPRESSION class.

Literal Template

You can represent all locally static values with literal templates.

The literal template is a primary template belonging to the EXPRESSION class. It contains the attributes shown in **Table 68.**

Table 68: literal Template

Attribute	Kind	Limit_Kind
base	set	<integer>
bit_length	max/min	N/A
extention_bits	local	N/A
ignore_in_default	local	N/A
parent_scope_case	local	N/A
size	max/min	N/A
value	template	literal
value_type	set	literal_type
truncating_leading_bits	local	N/A
ignore_in_default	local	N/A

Table 68: literal Template

Attribute	Kind	Limit_Kind
integer_literal_overflow	local	N/A
special_percentile_handle	local	N/A
truncating_extra_bits	local	N/A

- Use the value attribute to specify integer values or character strings that represent literal values. For example:

```
limit value in literal to 0
limit value in literal to "0"
```

  In the first case, the integer value zero is represented. In the second case, the bit_string “0” is represented. The actual interpretation depends on the value_type attribute.

- You can set the value_type attribute to any of the following enumerated literals:
  - integer_literal_type
  - real_literal_type
  - string_literal_type
  - logic_string_literal_type

  You can use logic_string_literal_type to make sure the literal is a binary, octal, decimal, or hexadecimal number with a base specification. For example:

```
1'b0
1'o1
1'd1
1'h1
```

  You can use integer_literal_type to make sure the literal is an integer without a base specification. For example:

```
1
0
100
```

  You can use real_literal_type to make sure the literal is a real number. (Real numbers do not have base specifications). For example:

```
1.1
1.0
1e3
-1.0
```

You can use string_literal_type to make sure the literal is a string. For example:

```
"Hello, World!"
```

- You can set the evaluation_time attribute to any of the following enumerated literals:
  - unresolved
  - locally_static_evaluation
  - globally_static_evaluation
  - dynamic_evaluation

  For example:

```
set evaluation_time to locally_static_evaluation
```

- Use the bit_length attribute to control the bit length of the literal expression. If Leda cannot evaluate the bit length of the literal expression, the result is –1.
- Use the size attribute to control the size specification of a based number. Leda ignores this attribute if the literal is not a based number. For example:

```
1’b0 -> size will match 1
3’o4 -> size will match 3
16’hFFFF -> size will match 16
‘b0 -> size will match -1 (unspecified)
```

- Use the ignore_in_default attribute to block the check of the default attribute in case statements.
- Use the extention_bits attribute to detect if the number of digits is less than the specified width of a literal. For example:

```
template EXTENDED_LITERAL is literal
force extention_bits
limit value to "^X", "^x"
end

no EXTENDED_LITERAL in literal
Message "Extension of X bits in constant"
severity WARNING

assign b = 4'bxx ; // extension of two 'x'
assign b = 4'bxxx ; // extension of one 'x'
```

- Use the truncating_extra_bits to check if the size of a literal is correct by value. For example:

```
no truncating_extra_bits in literal
```

  3’hff will flag, as hex ff requires 8 bits, whereas the size of the literal is 3 bits. It is used by prepackaged rule W19.

- Use the parent_scope_case attribute to determine if the literal is a part of the case item expression. For example:

```
template LITERAL is literal
  no paremt_scope_case
end
```

- The above template means, all the literals that do not belong to the case item expression. Use the integer_literal_overflow attribute to check that every unsized integer literal (12345689, ‘d34657383777) are within the range of -(2**31) to +(2**31)-1. For example:

```
      template T is literal
           no integer_literal_overflow
      end

      MY_RULE:
      limit literal to T
      message "integer value overflow"
      severity WARNING

module test( );

integer g = 429_496_72_0_0;
integer a = 17179869184;           // 2 ** 34
integer b = 'd1073741824;          // 2 ** 30
integer c = 2147483648;            // 2 ** 31
integer d = 'd4294967206;          // (2 ** 32) - 1
integer e = 33'd4294967296;        // (2 ** 32)
integer f = -2147483648;
integer h = 'b1001_1001_1001_1001_0111_1111_1111_1111_11111_1111_1111;

endmodule

example OUTPUT:-

     2: integer g = 429_496_72_0_0;
                    ^^^^^^^^^^^^^^

9000101202.v:2: STAR90> [WARNING] MY_RULE: integer value overflow

     3: integer a = 17179869184; // 2 ** 34
                    ^^^^^^^^^^^

9000101202.v:3: STAR90> [WARNING] MY_RULE: integer value overflow

     5: integer c = 2147483648;// 2 ** 31
                    ^^^^^^^^^^


9000101202.v:5: STAR90> [WARNING] MY_RULE: integer value overflow

     6: integer d = 'd4294967206;// (2 ** 32) - 1
                     ^^^^


9000101202.v:6: STAR90> [WARNING] MY_RULE: integer value overflow
```

Local Parameter Declaration Template

The grammar for local_parameter_declaration is:

```
local_parameter_declaration ::=
       localparam [ signing ] [ range ] list_of_param_assignments ;
       | localparam integer list_of_param_assignments ;
       | localparam real list_of_param_assignments ;
       | localparam realtime list_of_param_assignments ;
       | localparam time list_of_param_assignments ;
```

The local_parameter_declaration template contains the attributes shown in Table 69.

Table 69: local_parameter_declaration Template

Attribute	Kind	Limit_Kind
identifier	template	ID
expression	template	EXPRESSION
range	template	range
signed	local	N/A
unsigned	local	N/A
packed_dimension	local	N/A
packed_dimenson_count	max/min	N/A
byte	local	N/A
char	local	N/A
shortint	local	N/A
int	local	N/A
longint	local	N/A
integer	local	N/A
bit	local	N/A
logic	local	N/A
reg	local	N/A
time	local	N/A
shortreal	local	N/A
real	local	N/A

Table 69: local_parameter_declaration Template (Continued)

Attribute	Kind	Limit_Kind
realtime	local	N/A
void	local	N/A
enum	local	N/A
struct	local	N/A
union	local	N/A
data_type	template	data_type
overflow	local	N/A

Logic Declaration Template

The logic_declaration template contains the attributes shown in Table 70.

Table 70: logic_declaration Template

Attribute	Kind	Limit_Kind
identifier	template	ID
expression	template	EXPRESSION
signed	local	N/A
unsigned	local	N/A
static	local	N/A
automatic	local	N/A
packed_dimension	template	range
packed_dimension_count	max/min	N/A
unpacked_dimension	template	range
unpacked_dimension_count	max/min	N/A

Longint Declaration Template

The longint_declaration template contains the attributes shown in Table 71.

Table 71: longint_declaration Template

Attribute	Kind	Limit_Kind
identifier	template	ID
expression	template	EXPRESSION
signed	local	N/A
unsigned	local	N/A
static	local	N/A
automatic	local	N/A
unpacked_dimension	template	range
unpacked_dimension_count	max/min	N/A

Memory Addressing Template

The memory_addressing template contains the attributes shown in Table 72.

Table 72: memory_addressing Template

Attribute	Kind	Limit_Kind
identifier	template	ID
bit_length	max/min	N/A
flipflop	template	filpflop
latch	template	latch
object_definition	template	OBJECT_ITEM
upward_reference	local	N/A
range	template	range
rsp_bit_length	local	N/A
attribute_instance	local	N/A
attribute_instance_count	local	N/A

- Upward references are a feature of SystemVerilog. You can use the upward_reference attribute to write rules that do not allow the use of upward references in your Verilog code. For example, the following VeRSL code prohibits the use of upward references in module instantiations:

```
template MY_TEMPLATE is name
no upward_reference
end

template MY_TEMPLATE is port_connection
force port_expression
limit port_expression to MY_TEMPLATE
end

Rule1:
limit port_connection in module_instantiation to MY_TEMPLATE
  message "Upward refs not allowed in module instantiation"
  severity WARNING
```

Min, Typ, and Max Expression Template

The LRM (§4.3) defines the grammar for mintypmax expression as follows:

```
mintypmax_expression ::= expression
                       | min_expression : typ_expression : max_expression :
```

VeRSL represents the “expression : expression : expression” delay expression using the mintypmax_expression template.

The mintypmax_expression template is a primary template belonging to the EXPRESSION class. It contains the attributes shown in **Table 73.**

Table 73: mintypmax_expression Template

Attribute	Kind	Limit_Kind
max_expression	template	EXPRESSION
min_expression	template	EXPRESSION
typ_expression	template	EXPRESSION
bit_length	max/min	N/A
evaluation_time	set	evaluation_periods
operand_size_match	local	N/A
operand_size_match_no_carry	local	N/A

- You can set the evaluation_time attribute to any of the following enumerated literals:
  - unresolved
  - locally_static_evaluation
  - globally_static_evaluation
  - dynamic_evaluation

  For example:

  ```
  set evaluation_time to locally_static_evaluation
  ```

- Use the bit_length attribute to control the resulting bit length of the mintypmax expression. If Leda can evaluate the bit length of all the elements of the mintypmax expression, the resulting bit length is defined as:

  ```
  Max(Length(max_expression),
  Length(typ_expression),Length(min_expression))
  ```

  Otherwise, the result is –1.

- Use the operand_size_match attribute to control whether all the elements of the expression have the same size. If Leda cannot evaluate the size of one of the elements, the expression is considered to have an operand size match.

- Use the operand_size_match_no_carry attribute to control whether the left- and right-hand sides of a binary operation are the same size. This attribute differs from the operand_size_match attribute in that it does not take into account the carry bit from addition or subtraction. For example:

  ```
  force operand_size_match_no_carry in continuous_asign

  wire a, b, c;
  assign a = b+c; // rule does not fire here
  ```

Modport Declaration Template

The grammar for modport_declaration is:

```
modport_declaration ::= modport list_of_modport_identifiers ;
list_of_modport_identifiers ::= modport_item { , modport_item }
modport_item ::= modport_identifier ( modport_port { , modport_port } )
modport_port ::=input [port_type] port_identifier
       | output [port_type] port_identifier
       | inout [port_type] port_identifier
       | interface_identifier . port_identifier
       | import_export task named_task_proto
       | import_export function named_function_proto
       | import_export task_or_function_identifier { ,
      task_or_function_identifier }
import_export ::= import | export
```

The modport_declaration template contains the attributes shown in Table 74.

Table 74: modport_declaration Template

Attribute	Kind	Limit_Kind
modport_task_declaration	template	modport_task_declaration
modport_function_declaration	template	modport_function_declaration
identifier	template	ID
inout_declaration	template	inout_declaration
input_declaration	template	input_declaration
output_declaration	template	output_declaration

Module Declaration Template

The LRM (§11) defines the grammar for module declaration as follows:

```
module_declaration ::= module_keyword module_identifier [list_of_ports];
                    {module_item}
endmodule

module_keyword ::= module | macromodule

list_of_ports ::= (port {, port})

port ::= [port_expression]
       |. port_identifier ([port_expression])

port_expression ::= port_reference ({port_reference {, port_reference})

port_reference ::= port_identifier
                | port_identifier [constant_expression]
                | port_identifier [msb_constant_expression :
                lsb_constant_expression]

module_item ::= module_item_declaration
       | parameter_override
       | continuous_assign
       | gate_instantiation
       | udp_instantiation
       | module_instantiation
       | specify_block
       | initial_construct
       | always_construct

module_item_declaration ::= parameter_declaration
       | input_declaration
       | output_declaration
       | inout_declaration
       | net_declaration
       | reg_declaration
       | integer_declaration
       | real_declaration
       | time_declaration
       | realtime_declaration
       | event_declaration
       | task_declaration
       | function_declaration

parameter_override ::= defparam list_of_param_assignements
```

The module_declaration template is a primary template belonging to the OBJECT_ITEM class. It contains the attributes shown in **Table 75**.

Table 75: module_declaration Template

Attribute	Kind	Limit_Kind
automatic	local	N/A
asynchronous_initialization	limit	asynchronous_initialization
asynchronous_initialization_signal	max/min	N/A
asynchronous_load_signal	max/min	N/A
asynchronous_reset_signal	max/min	N/A
asynchronous_set_signal	max/min	N/A
synchronous_initialization_signal	max/min	N/A
synchronous_load_signal	max/min	N/A
synchronous_initialization	template	synchronous_initialization
synchronous_reset_signal	max/min	N/A
synchronous_set_signal	max/min	N/A
identifier	template	ID
always_construct	template	always_construct
end_comment	limit	end_comment
initial_construct	template	initial_construct
macromodule	local	N/A
module_instantiation	template	module_instantiation
root_module	local	N/A
module_declaration	template	module_declaration
parameter_declaration	template	parameter_declaration
file_layout	limit	file_layout
input_declaration	template	input_declaration
inout_declaration	template	inout_declaration
output_declaration	template	output_declaration

Table 75: module_declaration Template (Continued)

Attribute	Kind	Limit_Kind
net_declaration	template	net_declaration
reg_declaration	template	reg_declaration
time_declaration	template	time_declaration
integer_declaration	template	integer_declaration
real_declaration	template	real_declaration
realtime_declaration	template	realtime_declaration
duplicated_port	local	N/A
event_declaration	template	event_declaration
task_declaration	template	task_declaration
use_db_name	local	N/A
function_declaration	template	function_declaration
continuous_assign	template	continuous_assign
parameter_override	template	parameter_override
udp_instantiation	template	udp_instantiation
specify_block	template	specify_block
n_input_gate	template	n_input_gate
n_output_gate	template	n_output_gate
enable_gate	template	enable_gate
mos_switch	template	mos_switch
pass_switch	template	pass_switch
pass_en_switch	template	pass_en_switch
cmos_switch	template	cmos_switch
pull_gate	template	pull_gate
port	template	port
clock	template	clock

Table 75: module_declaration Template (Continued)

Attribute	Kind	Limit_Kind
top_module	template	module_declaration
unused_declaration	local	DECLARATIVE_ITEM
redundant_statement	local	ASSIGNMENT
port_order	local	N/A
file_length	max/min	N/A
fsm	template	fsm
header_comment	template	header_comment
port_count	max/min	N/A
statement_format	template	statement_format
file_name	local	N/A

- Use the macromodule attribute to control whether the module keyword must be module or macromodule. For example:

```
no macromodule in module_declaration
```

- Use the port_order attribute to predefine the order in which ports must appear. Specify the order according to the mode of the ports. When constraining this attribute, you can only use strings from the set “input”, “output”, and “inout”. Otherwise, Leda flags an error. Use blanks or commas to separate these words. For example:

```
limit port_order in module_declaration to “output inout input”
         message “ERROR: Ports are not in correct order”
         severity ERROR
```

  With this rule activated, Leda checks for the specified port order and flags an error if it is not respected.

- Use the file_name attribute to control the name of containing file. For example:

```
limit file_name in module_declaration to “<module>. v”
```

- Use the file_length attribute to specify the minimum or maximum length of the containing file.
- Use the clock specifies attribute to specify the minimum or maximum number of clock signals allowed in a module.

- Use the asynchronous_reset attribute to specify the minimum or maximum number of asynchronous resets allowed in a module.
- Use the synchronous_reset attribute to specify the minimum or maximum number of synchronous resets allowed in a module.
- Use the unused_declaration attribute to control whether a given module can contain unused variables (for example, regs, ports, or nets). You can also use this attribute with a VeRSL limit command to control the type of the unused variable. For example:

```
no unused_declaration in module_declaration
limit unused_declaration to integer_declaration
```

- Use the use_db_name attribute to ensure that the name of the module does not collide with the name of the cell in the $link_library.
- Use the redundant_statement attribute to control whether a given module can contain redundant statements. A redundant statement has identical left- and right-hand sides. For example:

```
a <= a;.
```

  The above example causes Leda to flag an error on the following rule:

```
no redundant_statement in module_declaration
```

- Use the duplicated_port attribute to control whether a module has duplicated ports. For example:

```
module (in1, in2, in3,
    in1 // Duplicated port name
```

  The above example causes Leda to flag an error on the following rule:

```
no duplicated_port in module_declaration
```

- Use the automatic attribute to check the use of automatic module. For example:

```
no automatic in module_declaration
```

- Use the top_module attribute to constrain the name of the top-level module in your design. Note that for this to work, you must specify the name of the top-level module using the -top option when using the command-line Checker. For example:

```
template TOP-MOD is module_declaration
    force top-module
end
template ID_WITHOUT_UNDERSCORE is identifier
    limit error_id to "_"
end

limit module_declaration to TOP_MOD
    if TOP_MOD then

    RULE1:
    limit identifier in input_declaration ID_WITHOUT_UNDERSCORE
    message "Top level port names should not have underscores"
    severity ERROR
    end if
```

  With this rule activated, Leda flags an error if the top-level module specified using the -top option has input names containing underscores.

  This next example makes sure that all register names in top_module start with "t".

```
template TOP-REG is reg_declaration
    limit_identifier to "^t"
end
template TOP_CHECK is module_declaration
    limit reg_declaration to TOP_REG
end
    RULE2:
    limit top_module in module_declaration to TOP_CHECK
    message "reg name in top module should begin with "t"
    severity ERROR
```

- Use the port_count attribute to control the number of external ports for a module. For example:

```
module m_no_end_comment (a, y); // FAIL
input a;
output y;
reg y;
endmodule

force end_comment in module_declaration
message "No comment after 'end...' keyword "
severity WARNING
```

Module Instantiation Template

The LRM (§12.1.2) defines the grammar for module instantiation as follows:

```
module_instantiation ::=module_identifier [ parameter_value_assignment ]
       module_instance { , module_instance } ;
parameter_value_assignment ::= # ( list_of_parameter_assignments )
list_of_parameter_assignments ::=ordered_parameter_assignment { ,
ordered_parameter_assignment }
       | named_parameter_assignment { , named_parameter_assignment }
ordered_parameter_assignment ::= expression | data_type
named_parameter_assignment ::=. parameter_identifier ( [ expression ] )
       | . parameter_identifier ( [ data_type ] )
module_instance ::= name_of_instance ( [ list_of_port_connections ] )
name_of_instance ::= module_instance_identifier { range }
list_of_port_connections ::=ordered_port_connection { ,
ordered_port_connection }
       | dot_named_port_connection { , dot_named_port_connection }
       | { named_port_connection , } dot_star_port_connection { ,
       named_port_connection }
ordered_port_connection ::= { attribute_instance } [ expression ]
named_port_connection ::= { attribute_instance } .port_identifier ( [
expression ] )
dot_named_port_connection ::={ attribute_instance } .port_identifier
       | named_port_connection
dot_star_port_connection ::={ attribute_instance } .*
```

The module_instantiation template is a primary template belonging to the CONCURRENT_STATEMENT class. It contains the attributes shown in **Table 76**.

Table 76: module_instantiation Template

Attribute	Kind	Limit_Kind
instance_identifier	template	ID
module_identifier	template	ID
multiply_connected_port	local	N/A
parameter_assignment	template	parameter_assignment
parameter_value_assignment	limit	EXPRESSION
port_expression	local	EXPRESSION
range	template	range
port_connection	template	port_connection
port_reference_declaration	local	N/A

Table 76: module_instantiation Template (Continued)

Attribute	Kind	Limit_Kind
unresolved	local	N/A
use_db_name	local	N/A
named_port_connection	local	N/A
complete_port_connection	local	N/A
operand_size_match	local	N/A
operand_size_match_no_carry	local	N/A
db_instantiation	local	N/A
keep_bits_order	local	N/A
special_percentile_handle	local	N/A

- Use the complete_port_connection attribute to control whether all ports must be connected.
- Use the module_identifier attribute to control the identifier of the instantiated module. Note that if a rule written for module_identifier fires, Leda points to the instance name:

```
<Module decl ID> <Instance ID> (....,....,....)
```

- Use the named_port_connection attribute to control if the port association is specified by name.
- Use the port_connection attribute to control the port association in a module instantiation. See also the “Port Connection Template” on page 182.
- Use the port_reference_declaration attribute to control whether all ports must be declared.
- Use the unresolved attribute to control whether there must be a reference (declaration) to the instantiated module.
- Use the parameter_value_assignment attribute to constrain the actual parameter expressions in a parameter association list. Note that you cannot use this parameter to test whether parentheses are missing in the list.

- Use the operand_size_match_no_carry attribute to control whether the left- and right-hand sides a binary operation are the same size. This attribute differs from the operand_size_match attribute in that it does not take into account the carry bit from from addition or subtraction. For example:

```
force operand_size_match_no_carry in continuous_asign
wire a, b, c;
assign a = b+c; // rule does not fire here
```

- Use the use_db_name attribute to ensure that the label of the module instantiation does not collide with the name of the cell in the $link_library.
- Use db_instantiation attribute to constrain the instantiation of db cells. This attribute can be used to ensure that the name of the module does not collide with the name of the cell in the $link_library. For example:

```
no db_instantiation in module_instantiation

module top();

db_cell I0( );        // Fail -- no db_instantiation rule
my_cell db_cell( );   // Fail -- no use_db_name rule

endmodule
```

- Use the multiply_connected_port attribute to detect if a port is multiply connected in a same instance. For example:

```
no multiply_connected_port in module_instantiation

module w289_test (a, b, c, d, y);
input a, b, c, d;
output y;
m test1(.aa(a), .bb(b), .aa(c) /* Error Here on .aa */, .dd(d),
yy(y));
endmodule
```

- Use the keep_bits_order attribute to checks if the port connection is of the same bit direction with that of definition in module instantiation.

Mos Switch Template

The LRM (§7.1) defines the grammar for mos switch as follows:

```
mos_switchtype [delay3] mos_switch_instance {, mos_switch_instance} ;

mos_switch_instance ::= [name_of_gate_instance] (output_terminal,
                         input_terminal,enable_terminal)

input_terminal ::= scalar_expression

enable_terminal ::= scalar_expression

output_terminal ::= terminal_identifier | terminal_identifier
                    [constant_expression]

mos_switchtype ::= nmos | pmos | rnmos | rpmos
```

The mos_switch template is a primary template belonging to the CONCURRENT_STATEMENT class. It contains the attributes shown in **Table 77.**

Table 77: mos_switch Template

Attribute	Kind	Limit_Kind
delay3	template	delay3
delay2	template	delay2
delay_control	template	delay_control
identifier	template	ID
nmos	local	N/A
pmos	local	N/A
range	template	range
rnmos	local	N/A
rpmos	local	N/A

Mux Template

The mux template contains the attribute shown in **Table 78.**

Table 78: mux Template

Attribute	Kind	Limit_Kind
identifier	template	ID

Use the mux template to detect the inference of a mux. For example:

```
no mux in always_construct
    message "Mux is inferred"
    severity WARNING

module test(gate, q);
    input gate;
    output q;
    wire gate;
    reg q;

always @ (gate ) begin
    case (gate) // q infers a mux.
    1'b0: q= 1'b0;
    1'b1: q= 1'b1;
    endcase
end

endmodule
```

N Input Gate Template

The LRM (§7.1) defines the grammar for n input gate as follows:

```
n_input_gatetype [drive_strength] [delay2] n_input_gate_instance {,
n_input_gate_instance} ;

n_input_gate_instance ::= [name_of_gate_instance] (output_terminal,
                      input_terminal
                      ,input_terminal})

name_of_gate_instance ::= gate_instance_identifier [range]

pullup_strength ::= (strength0, strength1)
                  | (strength1, strength0)
                  | (strength1

pulldown_strength ::= (strength0, strength1)
                   | (strength1, strength0)
                   | (strength0

input_terminal ::= scalar_expression

output_terminal ::= terminal_identifier | terminal_identifier
                      [constant_expression]

n_input_gatetype ::= and | nand | or | nor | xor | xnor
```

The n_input_gate template is a primary template belonging to the CONCURRENT_STATEMENT class. It contains the attributes shown in **Table 79.**

Table 79: n_input_gate Template

Attribute	Kind	Limit_Kind
and	local	N/A
delay2	template	delay2
delay_control	template	delay_control
drive_strength	template	drive_strength
identifier	template	ID
nand	local	N/A
nor	local	N/A
or	local	N/A
range	template	range
xnor	local	N/A
xor	local	N/A

N Output Gate Template

The LRM (§7.1) defines the grammar for n output gate as follows:

```
n_output_gatetype [drive_strength] [delay2] n_ output_instance
{, n_output_gate_instance} ;

n_output_gate_instance ::= [name_of_gate_instance] (output_terminal,
                       {output_terminal},
                       input_terminal)

name_of_gate_instance ::= gate_instance_identifier [range]

input_terminal ::= scalar_expression

output_terminal ::= terminal_identifier | terminal_identifier
                    [constant_expression]

n_output_gatetype ::= but | not
```

The n_output_gate template is a primary template belonging to the CONCURRENT_STATEMENT class. It contains the attributes shown in **Table 80**.

Table 80: n_output_gate Template

Attribute	Kind	Limit_Kind
buf	local	N/A
delay2	template	delay2
delay_control	template	delay_control
identifier	template	ID
not	local	N/A
range	template	range

Name Template

An expression is modeled by a name template if it has the following syntax:

```
identifier
```

The name template is a primary template belonging to the EXPRESSION_NAME class. It contains the attributes shown in **Table 81.**

Table 81: name Template

Attribute	Kind	Limit_Kind
identifier	template	ID
bit_length	max/min	N/A
flipflop	template	filpflop
latch	template	latch
upward_reference	local	N/A
object_definition	template	OBJECT_ITEM

- Use the object_definition attribute to control of the type of definition associated with a name. For example, if you only want to consider ports in a given expression, you can write the following rule:

```
template CLK_NAME is name
    limit object_definition to input_declaration,
inout_declaration,
output_declaration
end
```

- There are some application-specific attributes associated with the name template that make it easier to write commonly-used rules. For example:
  - Use the flipflop attribute to control whether the object infers a flip-flop.
  - Use the latch attribute to control whether the object infers a latch.
  - Use the bit_length attribute to control the bit length of the name. Note that if Leda cannot evaluate the bit length, the result is –1.
- Upward references are a feature of SystemVerilog. You can use the upward_reference attribute to write rules that do not allow the use of upward references in your Verilog code. For example, the following VeRSL code prohibits the use of upward references in module instantiations:

```
template MY_TEMPLATE is name
no upward_reference
end

template MY_TEMPLATE is port_connection
force port_expression
limit port_expression to MY_TEMPLATE
end

Rule1:
limit port_connection in module_instantiation to MY_TEMPLATE
  message "Upward refs not allowed in module instantiation"
  severity WARNING
```

Negedge Event Template

A negedge event is an expression preceded by the keyword negedge:

```
negedge expression
```

The negedge_event template is a primary template belonging to the EXPRESSION class. It contains the attribute shown in Table 82.

Table 82: negedge_event Template

Attribute	Kind	Limit_Kind
expression	template	EXPRESSION

Net Declaration Template

The LRM (§3.2.1) defines the grammar for net declaration as follows:

```
net_declaration ::= net_type [vectored| scalared] [range] [delay3]
   list_of_net_identifiers ;
   | trireg [vectored | scalared] [charge_strength] [range] [delay3]
   list_of_net_identifiers ;
    | net_type [vectored | scalared] [drive_strength] [range] [delay3]
   list_of_net_decl_assignments ;

net_type ::= wire | tri | tri1 | supply0 | wand | triand | supply1| wor|
          trior

range ::= [msb_constant_expression : lsb_constant_expression]

drive_strength ::= (strength0 , strength1)
        | (strength1 , strength0)
        | (strength0 , highz1)
        | (strength1 , highz0)
        | (highz1, strength0)
        | (highz0, strength1)

strength0 ::= supply0 | strong0 | pull0 | weak0

strength1 ::= supply1 | strong1 | pull1 | weak1

charge_strength ::= (small) | (medium) | (large)

delay3 ::= #delay_value |#(delay_value [, delay_value[,delay_value]])

delay2 ::= #delay_value |#(delay_value [, delay_value])

delay_value ::= unsigned_number |parameter_identifier
             | constant_mintypmax_expression
```

```
list_of_net_decl_assignments ::= net_decl_assignment
                            {, net_decl_assignment}.

net_decl_assignment ::= net_identifier = expression
```

The net_declaration template is a primary template belonging to the OBJECT_ITEM class. It contains the attributes described in **Table 83**.

Table 83: net_declaration Template

Attribute	Kind	Limit_Kind
is_partly_used	limit	N/A
implicit_type	local	N/A
wire	local	N/A
tri	local	N/A
tri1	local	N/A
signed	local	N/A
unsigned	local	N/A
supply0	local	N/A
wand	local	N/A
triand	local	N/A
tri0	local	N/A
supply1	local	N/A
wor	local	N/A
trior	local	N/A
trireg	local	N/A
scalared	local	N/A
vectored	local	N/a
charge_strength	template	charge_strength
comment	template	comment
identifier	template	ID
is_initialization	local	N/A

Table 83: net_declaration Template (Continued)

Attribute	Kind	Limit_Kind
is_load	local	N/A
is_reset	local	N/A
is_read	local	N/A
is_set	local	N/A
one_declaration_per_line	local	N/A
range	template	range
declarative_region	template	REGION
delay_control	template	delay_control
delay2	template	delay2
delay3	template	delay3
expression	template	EXPRESSION
drive_strength	template	drive_strength
driving_expression	template	EXPRESSION
signals_driven	template	DRIVEN_OBJECT
tristate	local	N/A
port	template	DECLARATIVE_ITEM
overflow	local	N/A
operand_size_match	local	N/A
operand_size_match_no_carry	local	N/A
outputs_driven	max/min	N/A
packed_dimension	template	dimension
partly_used	local	N/A
unpacked_dimension_count	max/min	N/A

- The wires through trireg attributes correspond to the different net types that a net can have. For example, the command:

```
no trireg in net_declaration severity error
```

means that nets of type trireg are not allowed.

- Use the comment attribute to control the comments associated with variable declarations.
- Use the declarative_region attribute to control the region where the variable is declared.
- Use the tristate attribute to control whether the net should be tristated.
- Use the driving_expression attribute to control the expressions accepted as drivers for the current net.
- Use the signals_driven attribute to control the signals driven by the current net.
- Use the is_read attribute to make sure all bits of the signal, variable, or port specified are read at least once somewhere in the Verilog code. When you force this attribute, Leda verifies that the signal is read. For example:

```
force is_read in inout_declaration
```

- Use the vectored attribute to control whether the keyword “vectored” should be present.
- Use the scalared attribute to control whether the keyword “scalared” should be present.
- Use the one_declaration_per_line attribute to control whether multiple variables are declared on the same line.
- Use the overflow attribute to catch cases where the left-hand side of an assignment is smaller than the right-hand side. Note that the operand_size_match attribute applies to any expression, whereas the overflow attribute is just for assignments.
- Use the operand_size_match attribute to control whether the operands (right- and left-hand side) of any expression are the same size.
- Use the operand_size_match_no_carry attribute to control whether the left- and right-hand sides of a binary operation are the same size. This attribute differs from the operand_size_match attribute in that it does not take into account the carry bit from addition or subtraction. For example:

```
force operand_size_match_no_carry in continuous_asign

wire a, b, c;
assign a = b+c; // rule does not fire here
```

- Use the port attribute to control the type of port associated with the net. For example:

```
template NET_WITH_PORT is net_declaration
   no port
end
limit net_declaration to NET_WITH_PORT
message “This net is a redeclaration of an input/output/inout”
severity WARNING
```

  The above rule above fires for the declaration of A as a wire:

```
output A;
wire A;
```

  Whereas, this next rule:

```
limit port in net_declaration to input_declaration,
output_declaration
```

  causes Leda to flag an error on all nets declared as follows:

```
inout A;
wire A;
```

- Use the partly_used attribute to detect if the vector is partially used (only some bits used). For example:

```
no partly_used in input_declaration
message "Not all the bits of the vector are used"
severity WARNING
```

```
module w498 (in,out);
input [3:0] in; //FAIL
output [2:0] out;
wire [3:0] in;
```

```
assign out[1:0] = in[1:0];
```

```
endmodule
```

Non Blocking Assignment Template

The LRM (§9.2.2) defines the grammar for non-blocking assignment as follows:

```
non_blocking_assignment ::= reg_lvalue <= [delay_or_event_control]
                        expression
```

The non_blocking_assignment template is a primary template belonging to the EXPRESSION class. It contains the attributes shown in Table 84.

Table 84: non_blocking_assignment Template

Attribute	Kind	Limit_Kind
delay_control	template	delay_control
event_control	template	event_control
expression	template	EXPRESSION
read_write	local	N/A
reg_lvalue	template	EXPRESSION
repeat_event	template	repeat_event
function_in_lhs	local	N/A
overflow	local	N/A
operand_size_match	local	N/A
operand_size_match_no_carry	local	N/A
one_ assignment_per_line	local	N/A
lr_sign_match	local	N/A
special_percentile_handle	local	N/A

- Use the overflow attribute to catch cases where the left-hand side of an assignment is smaller than the right-hand side. Note that the operand_size_match attribute applies to any expression, whereas the overflow attribute is just for assignments.
- Use the operand_size_match attribute to control whether the operands (right- and left-hand side) of any expression are the same size. If Leda cannot evaluate the size of any of the elements, the statement is considered to have an operand size match.

- Use the operand_size_match_no_carry attribute to control whether the left- and right-hand sides of a binary operation are the same size. This attribute differs from the operand_size_match attribute in that it does not take into account the carry bit from addition or subtraction. For example:

```
force operand_size_match_no_carry in continuous_asign

wire a, b, c;
assign a = b+c; // rule does not fire here
```

- Use the read_write attribute to control whether the same signal is present on both sides of the assignment.
- Use the lr_sign_match attribute with force/no keywords to check if the left hand side and right hand side of the assignment statement are both signed or unsigned.

Nounconnected Drive Template

The nounconnected_drive template is a primary template belonging to no classes. It contains the attributes shown in Table 85.

Table 85: nounconnected_drive Template

Attribute	Kind	Limit_Kind
enclosing_filename	template	ID

Output Declaration Template

The LRM (§12.3.2) defines the grammar for output declaration as follows:

```
output_declaration ::= output [range] list_of_port_identifiers ;
```

The output_declaration template is a primary template belonging to the OBJECT_ITEM_DRIVEN_OBJECT class. It contains the attributes shown in **Table 86**.

Table 86: output_declaration Template

Attribute	Kind	Limit_Kind
ansic_style	local	N/A
combined	local	N/A
comment	template	comment
declarative_region	template	REGION
identifier	template	ID
is_initialization	local	N/A

Table 86: output_declaration Template (Continued)

Attribute	Kind	Limit_Kind
is_load	local	N/A
is_read	local	N/A
is_reset	local	N/A
is_set	local	N/A
range	template	range
driver_declaration	template	DRIVER_OBJECT
driving_expression	template	EXPRESSION
expression	template	EXPRESSION
signals_driven	template	DRIVEN_OBJECT
consistent_range	local	N/A
one_declaration_per_line	local	N/A
tristate	local	N/A
outputs_driven	max/min	N/A
unpacked_dimension_count	max/min	N/A

- Use the comment attribute to control the comments associated with variable declarations.
- Use the consistent_range attribute to control whether the range specification of the output remains unchanged if the input port is redeclared as another variable. For example:

```
ouput [3:0] clk_en;
reg [2:0] clk_en;
```

  The following rule:

```
force consistent_range in output_declaration
```

  causes Leda to flag an error on the example above.
- Use the declarative_region attribute to control the region where the variable is declared.

- Use the driver_declaration attribute to control the second declaration of the port. For example:

```
output [3:0] clk_en;
wire [3:0] clk_en;
```

  The following rule:

```
limit driver_declaration in output_declaration to reg_declaration
```

  causes Leda to flag an error on the example above.

- Use the driving_expression attribute to control the expressions accepted as drivers for the current variable.
- Use the one_declaration_per_line attribute to control whether multiple variables are declared on the same line.
- Use the signals_driven attribute to control the signals driven by the current variable.
- Use the tristate attribute to control whether the port should be tristated.
- Use the is_read attribute to make sure all bits of the signal, variable, or port specified are read at least once somewhere in the Verilog code. When you force this attribute, Leda verifies that the signal is read. For example:

```
force is_read in output_declaration
```

Parameter Assignment Template

The grammar for parameter assignment is:

```
parameter_value_assignment ::= # ( list_of_parameter_assignments )
list_of_parameter_assignments ::=ordered_parameter_assignment { ,
ordered_parameter_assignment }
| named_parameter_assignment { , named_parameter_assignment }
ordered_parameter_assignment ::= expression
named_parameter_assignment ::=. parameter_identifier ( [ expression ] )
```

The parameter_assignment template contains the attributes shown in Table 87.

Table 87: parameter_assignment Template

Attribute	Kind	Limit_Kind
actual_identifier	template	ID
formal_declaration	template	OBJECT_ITEM
expression	template	EXPRESSION

Par Block Template

The LRM (§9.8.2) defines the grammar for par block as follows:

```
par_block ::= fork [ : block_identifier { block_item_declaration } ]
           {statement} join

block_item_declaration ::= parameter_declaration
       | reg_declaration
       | integer_declaration
       | real_declaration
       | time_declaration
       | realtime_declaration
       | event_declaration
```

The par_block template is a primary template belonging to the SEQUENTIAL_STATEMENT class. It contains the attributes shown in Table 88.

Table 88: par_block Template

Attribute	Kind	Limit_Kind
blocking_assignment	template	blocking_assignment
case_statement	template	case_statement
casex_statement	template	casex_statement
casez_statement	template	casez_statement
conditional_statement	template	conditional_statement
disable_statement	template	disable_statement
end_comment	limit	end_comment
event_trigger	template	event_trigger
identifier	template	ID
for_statement	template	for_statement
forever_statement	template	forever_statement
non_blocking_assignment	template	non_blocking_assignment
par_block	template	par_block
procedural_continuous_assign	template	procedural_continuous_assign
procedural_continuous_deassign	template	procedural_continuous_deassign
procedural_continuous_force	template	procedural_continuous_force

Table 88: par_block Template (Continued)

Attribute	Kind	Limit_Kind
procedural_continuous_release	template	procedural_continuous_release
procedural_timing_control_statement	template	procedural_timing_control_statement
process_statement	template	process_statement
repeat_statement	template	repeat_statement
seq_block	template	seq_block
system_task_enable	template	system_task_enable
task_enable	template	task_enable
wait_statement	template	wait_statement
while_statement	template	while_statement
parameter_declaration	template	parameter_declaration
reg_declaration	template	reg_declaration
integer_declaration	template	integer_declaration
real_declaration	template	real_declaration
time_declaration	template	time_declaration
realtime_declaration	template	realtime_declaration
event_declaration	template	event_declaration
unused_declaration	local	DECLARATIVE_ITEM
statement_format	template	statement_format

Use the unused_declaration attribute to control whether a given named begin/end block should contain unused variables (for example, regs or nets). You can also use this attribute with a VeRSL limit command to control the type of the unused variable. For example:

```
no unused_declaration in seq_block
limit unused_declaration to integer_declaration
```

Example:

```
module m_no_end_comment (a, y); // FAIL
input a;
output y;
reg y;
endmodule
force end_comment in module_declaration
message "No comment after 'end...' keyword "
severity WARNING
```

Parameter Declaration Template

The LRM (§3.9) defines the grammar for parameter declaration as follows:

```
parameter_declaration ::=
parameter [ signing ][ range ] list_of_param_assignments
        | parameter integer list_of_param_assignments
        | parameter real list_of_param_assignments
        | parameter realtime list_of_param_assignments
        | parameter time list_of_param_assignments
specparam_declaration ::= specparam [ range ]
list_of_specparam_assignments ;

list_of_param_assignments ::= param_assignment {, param_assignment}

param_assignment ::= parameter_identifier = expression
```

The parameter_declaration template is a primary template belonging to the OBJECT_ITEM class. It contains the attributes shown in **Table 89**.

Table 89: parameter_declaration Template

Attribute	Kind	Limit_Kind
ansic_style	local	N/A
comment	template	comment
expression	template	EXPRESSION
declarative_region	template	REGION
identifier	template	ID
one_declaration_per_line	local	N/A
range	template	range
is_port	local	N/A

Table 89: parameter_declaration Template (Continued)

Attribute	Kind	Limit_Kind
is_type	local	N/A
signed	local	N/A
unsigned	local	N/A
reg	local	N/A
time	local	N/A
shortreal	local	N/A
real	local	N/A
realtime	local	N/A
void	local	N/A
enum	local	N/A
struct	local	N/A
union	local	N/A
data_type	local	N/A
int	local	N/A
longint	local	N/A
integer	local	N/A
bit	local	N/A
logic	local	N/A
byte	local	N/A
char	local	N/A
shortint	local	N/A
unpacked_dimension_count	max/min	N/A

- Use the comment attribute to control the comments associated with variable declarations.
- Use the declarative_region attribute to control the region where the variable is declared.

- Use the one_declaration_per_line attribute to control whether multiple variables are declared on the same line.

Parameter Override Template

The LRM (§12.1) defines the grammar for parameter override statement as follows:

```
parameter_override ::= defparam list_of_param_assignements
```

The parameter_override template is a primary template belonging to the CONCURRENT_STATEMENT class. It contains the attributes shown in **Table 90**.

Table 90: parameter_override Template

Attribute	Kind	Limit_Kind
constant_expression	template	EXPRESSION
identifier	template	ID

Part Select Template

An expression is modeled by the part_select template if it has the following syntax:

```
identifier [msb_expression : lsb_expression]
```

The part_select template is a primary template belonging to the EXPRESSION_NAME class. It contains the attributes shown in **Table 91**.

Table 91: part_select Template

Attribute	Kind	Limit_Kind
identifier	template	ID
range	template	range
bit_length	max/min	N/A
flipflop	template	filpflop
latch	template	latch
object_definition	template	OBJECT_ITEM
upward_reference	local	N/A
out_of_range	local	N/A

- Use the range attribute to control the [msb_expression : lsb_expression] clause.

- There are several application-specific attributes associated with the part_select template that make it easier to write commonly used rules:
  - Use the object_definition attribute to control the type of definition associated with part_select. For example, if you only want to consider ports in a given expression, you can write:

```
template CLK_NAME is part_select
limit object_definition to input_declaration,
                        inout_declaration,
                        output_declaration
end
```

  - Use the flipflop attribute to control whether the object infers a flip-flop.
  - Use the latch attribute to control whether the object infers a latch.
  - Use the out_of_range attribute to control whether the part select bounds are outside the range specification. For example:

```
reg [3:0] a;
always begin
  case(a[4:3])
   ….
```

    With the following VeRSL rule activated, Leda signals an error on the above example:

```
template LEGAL_PART_SELECT is part_select
  no out_of_range
end
limit expression in case_statement to LEGAL_PART_SELECT
  message "Part select out of range"
  severity ERROR
```

- Use the bit_length attribute to control the bit length of the part_select expression. If Leda cannot evaluate the bit length of the part_select expression, the result is –1.

- Upward references are a feature of SystemVerilog. You can use the upward_reference attribute to write rules that do not allow the use of upward references in your Verilog code. For example, the following VeRSL code prohibits the use of upward references in module instantiations:

```
template MY_TEMPLATE is name
no upward_reference
end

template MY_TEMPLATE is port_connection
force port_expression
limit port_expression to MY_TEMPLATE
end

Rule1:
limit port_connection in module_instantiation to MY_TEMPLATE
  message "Upward refs not allowed in module instantiation"
  severity WARNING
```

Pass En Switch Template

The LRM (§7.1) defines the grammar for pass enable switch as follows:

```
pass_en_switchtype [delay3] pass_en_switch_instance
{, pass_en_switch_instance} ;

pass_enable_switch_instance ::= [name_of_gate_instance] (inout_terminal,
                            inout_terminal,
                            enable_terminal)

name_of_gate_instance ::= gate_instance_identifier [range]

enable_terminal ::= scalar_expression

inout_terminal ::= terminal_identifier | terminal_identifier
                   [constant_expression]

pass_en_switchtype ::= tranif0 | tranif1 | rtranif1 | rtranif0
```

The pass_en_switch template is a primary template belonging to the CONCURRENT STATEMENT class. It contains the attributes shown in **Table 92**.

Table 92: pass_en_switch Template

Attribute	Kind	Limit_Kind
identifier	template	ID
delay2	template	delay2

Table 92: pass_en_switch Template (Continued)

Attribute	Kind	Limit_Kind
delay3	template	delay3
delay_control	template	delay_control
range	template	range
rtranif0	local	N/A
rtranif1	local	N/A
tranif0	local	N/A
tranif1	local	N/A

Pass Switch Template

The LRM (§7.1) defines the grammar for pass switch as follows:

```
pass_switchtype pass_switch_instance {, pass_switch_instance} ;

pass_switch_instance ::= [name_of_gate_instance] (inout_terminal,
                         inout_terminal)

name_of_gate_instance ::= gate_instance_identifier [range]

inout_terminal ::= terminal_identifier | terminal_identifier
                       [constant_expression]

pass_switchtype ::= tran | rtran
```

The pass_switch template is a primary template belonging to the CONCURRENT_STATEMENT class. It contains the attributes shown in **Table 93.**

Table 93: pass_switch Template

Attribute	Kind	Limit_Kind
identifier	template	ID
range	template	range
rtran	local	N/A
tran	local	N/A

Port Connection Template

The port_connection template is a secondary template belonging to no classes. It contains the attributes shown in Table 94.

Table 94: port_connection Template

Attribute	Kind	Limit_Kind
actual_identifier	template	ID
dot_name_port_connection	local	N/A
formal_declaration	template	OBJECT_ITEM
port_expression	template	EXPRESSION

- Use the actual_identifier attribute to control the identifier of the actual port. For example, the following VeRSL rule:

```
template ID1 is identifier
    limit limit_id to "^<formal>$"
end

template PC1 is port_connection
    limit actual_identifier to ID1
end

limit port_connection in module_instantiation to PC1
message "Formal and actual port names should be the same"
severity error
```

causes Leda to flag and error for port A in inst1 of the SUB module in the following Verilog code:

```
module SUB(a, b);
    input a;
    output b;
 endmodule

 module TOP;
   reg A, b;
   SUB inst1(A, b); // Rule fires on A
endmodule
```

- Use the formal_declaration attribute to control the type of the variable used in a port expression. For example, the following VeRSL rule:

```
template PC2 is port_connection
    limit formal_declaration to input_declaration, output_declaration
end

limit port_connection in task_enable to PC2
message "Formal declaration of port should be either an input or an
       output"
severity WARNING
```

  causes Leda to flag an error on B in the following example Verilog code:

```
module test2;
task my_task;
input A;
inout B;
begin
     B = A;
end
endtask

reg A;
reg B;
always begin
    my_task(A, B); // Rule fires on B
end
```

- Use the port_expression attribute to control the expression of a port_connection. For example, the following VeRSL rule:

```
template BOP_PC3 is port_connection
  limit port_expression to binary_operation
end
no BOP_PC3 in port_connection of task_enable
  message "Do not use complex operations in port mapping"
  severity ERROR
```

causes Leda to flag an error on “A&B” in the following example Verilog code:

```
module TEST3;
   reg A,B,C,D;
task task3;
input A;
output B;
begin
  B = A;
end
endtask

always begin
      task3(A&B, C); // Rule fires on A&B
   end
endmodule
```

Port Template

The LRM (§12.3.1) defines the grammar for port declaration as follows:

```
port ::= [port_expression]
        |.port_identifier([port_expression])

port_expression ::= port_reference
                  ({port_reference{, port_reference})

port_reference ::= port_identifier
                 | port_identifier [constant_expression]
       | port_identifier [msb_constant_expression :
         lsb_constant_expression]
```

The port template is a primary template belonging to no classes. It contains the attributes shown in Table 95.

Table 95: port Template

Attribute	Kind	Limit_Kind
actual_identifier	template	ID
expression	template	EXPRESSION
formal_declaration	template	formal_declaration
identifier	template	ID
port_reference	template	NAME
comment	template	comment
one_declaration_per_line	local	N/A

Posedge Event Template

A posedge event is an expression preceded by the keyword posedge:

```
posedge expression
```

The posedge_event template is a secondary template belonging to the EXPRESSION class. It contains the attribute shown in Table 82.

Table 96: posedge_event Template

Attribute	Kind	Limit_Kind
expression	template	EXPRESSION

Procedural Continuous Assign Template

The LRM (§9.3) defines the grammar for procedural continuous assignments as follows:

```
procedural_continuous_assignments ::= assign reg_assignment
                                    | deassign reg_lvalue
                                    | force reg_assignment
                                    | force net_assignment
                                    | release reg_lvalue
                                    | release net_lvalue

reg_assignment ::= reg_lvalue = expression

net_assignment ::= net_lvalue = expression
```

The procedural_continuous_assign template is a primary template belonging to the SEQUENTIAL_STATEMENT class. It contains the attributes shown in Table 97.

Table 97: procedural_continuous_assign Template

Attribute	Kind	Limit_Kind
expression	template	EXPRESSION
read_write	local	N/A
reg_lvalue	template	reg_lvalue
function_in_lhs	local	N/A
overflow	local	N/A
operand_size_match	local	N/A
operand_size_match_no_carry	local	N/A
one_assignment_per_line	local	N/A
lr_sign_match	local	N/A
special_percentile_handle	local	N/A

- Use the overflow attribute to catch cases where the left-hand side of an assignment is smaller than the right-hand side. Note that the operand_size_match attribute applies to any expression, whereas the overflow attribute is just for assignments.
- Use the operand_size_match attribute to control whether the operands (right- and left-hand side) of any expression are the same size. If Leda cannot evaluate the size of any of the elements, the statement is considered to have an operand size match.

- Use the operand_size_match_no_carry attribute to control whether the left- and right-hand sides of a binary operation are the same size. This attribute differs from the operand_size_match attribute in that it does not take into account the carry bit from addition or subtraction. For example:

```
force operand_size_match_no_carry in continuous_asign

wire a, b, c;
assign a = b+c; // rule does not fire here
```

- Use the read_write attribute to detect whether the same signal is present on both sides of the assignment.
- Use the lr_sign_match attribute with force/no keywords to check if the left hand side and right hand side of the assignment statement are both signed or unsigned.

Procedural Continuous Deassign Template

The procedural_continuous_deassign template is a primary template belonging to the SEQUENTIAL_STATEMENT class. It contains the attribute shown in Table 98.

Table 98: procedural_continuous_deassign Template

Attribute	Kind	Limit_Kind
reg_lvalue	template	reg_lvalue
function_in_lhs	local	N/A

Procedural Continuous Force Template

The procedural_continuous_force template is a primary template belonging to the SEQUENTIAL_STATEMENT class. It contains the attributes shown in Table 99.

Table 99: procedural_continuous_force Template

Attribute	Kind	Limit_Kind
expression	template	EXPRESSION
read_write	local	N/A
reg_lvalue	template	reg_lvalue
function_in_lhs	local	N/A
overflow	local	N/A
operand_size_match	local	N/A
operand_size_match_no_carry	local	N/A

Table 99: procedural_continuous_force Template (Continued)

Attribute	Kind	Limit_Kind
one_assignment_per_line	local	N/A
lr_sign_match	local	N/A
special_percentile_handle	local	N/A

- Use the overflow attribute to catch cases where the left-hand side of an assignment is smaller than the right-hand side. Note that the operand_size_match attribute applies to any expression, whereas the overflow attribute is just for assignments.
- Use the operand_size_match attribute to control whether the operands (right- and left-hand side) of any expression are the same size. If Leda cannot evaluate the size of any of the elements, the statement is considered to have an operand size match.
- Use the operand_size_match_no_carry attribute to control whether the left- and right-hand sides of a binary operation are the same size. This attribute differs from the operand_size_match attribute in that it does not take into account the carry bit from addition or subtraction. For example:

```
force operand_size_match_no_carry in continuous_asign

wire a, b, c;
assign a = b+c; // rule does not fire here
```

- Use the read_ write attribute to detect if the same signal is present on both sides of the assignment.
- Use the lr_sign_match attribute with force/no keywords to check if the left hand side and right hand side of the assignment statement are both signed or unsigned.

Procedural Continuous Release Template

The procedural_continuous_release template is a primary template belonging to the SEQUENTIAL_STATEMENT class. It contains the attribute shown in **Table 100**.

Table 100: procedural_continuous_release Template

Attribute	Kind	Limit_Kind
reg_lvalue	template	reg_lvalue
function_in_lhs	local	N/A

Procedural Timing Control Statement Template

The LRM (§9.6) defines the grammar for looping statement as follows:

```
procedural_timing_control_statement ::= delay_or_event_control
                                        statement_or_null

delay_or_event_control ::= delay_control
                         |  event_control
                         |  repeat (expression ) event_control

delay_control ::= # delay_value
                  | # ( mintypmax_expression )

event_control ::=@ event_identifier
        | @ ( event_expression )
        | @*
        | @ (*)

event_expression ::= expression
                   | event_identifier
                   | posedge expression
                   | negedge expression
                   | event_expression or event_expression
```

The procedural_timing_control_statement template is a primary template belonging to the SEQUENTIAL_STATEMENT class. It contains the attributes shown in Table 101.

Table 101: procedural_timing_control_statement Template

Attribute	Kind	Limit_Kind
blocking_assignment	template	blocking_assignment
case_statement	template	case_statement
casex_statement	template	casex_statement
casez_statement	template	casez_statement
conditional_statement	template	conditional_statement
delay_control	template	delay_control
disable_statement	template	disable_statement
event_control	template	event_control
event_trigger	template	event_trigger
for_statement	template	for_statement

Table 101: procedural_timing_control_statement Template (Continued)

Attribute	Kind	Limit_Kind
forever_statement	template	forever_statement
non_blocking_assignment	template	non_blocking_assignment
null_statement	template	null_statement
par_block	template	par_block
procedural_continuous_assign	template	procedural_continuous_assign
procedural_continuous_deassign	template	procedural_continuous_deassign
procedural_continuous_force	template	procedural_continuous_force
procedural_continuous_release	template	procedural_continuous_release
procedural_timing_control_statement	template	procedural_timing_control_statement
process_statement	template	process_statement
repeat_event	template	repeat_event
repeat_statement	template	repeat_statement
seq_block	template	seq_block
system_task_enable	template	system_task_enable
task_enable	template	task_enable
wait_statement	template	wait_statement
while_statement	template	while_statement
statement	template	SEQUENTIAL_STATEMENT

Use the statement attribute to control the statement in the timing control statement.

Process Statement Template

The process_statement template contains the attributes shown in Table 102.

Table 102: process_statement Template

Attribute	Kind	Limit_Kind
asynchronous_initialization	limit	asynchronous_initialization
statement	template	process_statement

Pull Gate Template

The LRM (§7.1) defines the grammar for pull gate as follows:

```
pullup [pullup_strength] pull_gate _instance {, pull_gate_instance} ;

pulldown [pulldown_strength] pull_gate _instance {, pull_gate_instance};


pull_gate_instance ::= [name_of_gate_instance] (output_terminal)

name_of_gate_instance ::= gate_instance_identifier [range]

pullup_strength ::= (strength0, strength1)
                  | (strength1, strength0)
                  | (strength1

pulldown_strength ::= (strength0, strength1)
                    | (strength1, strength0)
                    | (strength0

output_terminal ::= terminal_identifier
                  | terminal_identifier
                  [constant_expression]
```

The pull_gate template is a primary template belonging to the CONCURRENT_STATEMENT class. It contains the attributes shown in **Table 103.**

Table 103: pull_gate Template

Attribute	Kind	Limit_Kind
identifier	template	ID
pulldown	local	N/A

Table 103: pull_gate Template

Attribute	Kind	Limit_Kind
pullup	local	N/A
range	template	range

Range Template

An expression is modeled by a range template if it has the following syntax:

```
[msb_expression : lsb_expression]
```

The range template is a secondary template belonging to the FIELD class. It contains the attributes shown in **Table 104**.

Table 104: range Template

Attribute	Kind	Limit_Kind
msb_constant_expression	template	EXPRESSION
lsb_constant_expression	template	EXPRESSION
ascending	local	N/A

If you use a VeRSL force command with the ascending attribute and if the msb_expression and lsb_expression are locally static, Leda checks to make sure the msb_expression is less than or equal to the lsb_expression.

Real Declaration Template

The LRM (§3.9) defines the grammar for real declaration as follows:

```
real_declaration ::= real list_of_real_identifiers ;
list_of_real_identifiers ::= real_identifier {, real _identifier}
```

The real_declaration template is a primary template belonging to the OBJECT_ITEM class. It contains the attributes shown in **Table 105.**

Table 105: real_declaration Template

Attribute	Kind	Limit_Kind
dimension_count	max/min	N/A
comment	template	comment
identifier	template	ID

Table 105: real_declaration Template (Continued)

Attribute	Kind	Limit_Kind
is_initialization	local	N/A
is_load	local	N/A
is_read	local	N/A
is_set	local	N/A
declarative_region	template	REGION
driving_expression	template	EXPRESSION
one_declaration_per_line	local	N/A
range	template	range
signals_driven	template	DRIVEN_OBJECT

- Use the comment attribute to control comments associated with variable declarations.
- Use the declarative_region attribute to control the region where the variable is declared.
- Use the driving_expression attribute to control the expressions accepted as drivers for the current variable.
- Use the one_declaration_per_line attribute to control whether multiple variables are declared on the same line.
- Use the signals_driven attribute to control the signals driven by the current variable.
- Use the is_read attribute to make sure all bits of the signal, variable, or port specified are read at least once somewhere in the Verilog code. When you force this attribute, Leda verifies that the signal is read. For example:

```
force is_read in real_declaration
```

Realtime Declaration Template

The LRM (§3.9) defines the grammar for realtime declaration as follows:

```
realtime_declaration ::= realtime list_of_real_identifiers ;
list_of_real_identifiers ::= real_identifier {, real _identifier}
```

The realtime_declaration template is a primary template belonging to the OBJECT_ITEM class. It contains the attributes shown in **Table 106.**

Table 106: realtime_declaration Template

Attribute	Kind	Limit_Kind
dimension_count	max/min	N/A
comment	template	comment
identifier	template	ID
is_initialization	local	N/A
is_load	local	N/A
is_read	local	N/A
is_reset	local	N/A
is_set	local	N/A
declarative_region	template	REGION
driving_expression	template	EXPRESSION
one_declaration_per_line	local	N/A
range	template	range
signals_driven	template	DRIVEN_OBJECT

- Use the comment attribute to control the comments associated with variable declarations.
- Use the declarative_region attribute to control the region where the variable is declared.
- Use the driving_expression attribute to control the expressions accepted as drivers for the current variable.
- Use the one_declaration_per_line attribute to control whether multiple variables are declared on the same line.
- Use the signals_driven attribute to control the signals driven by the current variable.

- Use the is_read attribute to make sure all bits of the signal, variable, or port specified are read at least once somewhere in the Verilog code. When you force this attribute, Leda verifies that the signal is read. For example:

```
force is_read in realtime_declaration
```

Reg Assignment Template

The reg_assignment template is a secondary template belonging to no classes. It contains the attributes shown in Table 107.

Table 107: reg_assignment Template

Attribute	Kind	Limit_Kind
expression	template	EXPRESSION
reg_lvalue	template	EXPRESSION
overflow	local	N/A

Use the reg_assignment template to control the initial and step reg assignment statements in a for loop statement. For example, the following rules cause Leda to flag an error if the initial loop bound of a for statement is not statically computable:

```
template LOC_STAT_BINOP is binary_operation
       set evaluation_time to locally_static_evaluation
end

template GLOB_STAT_BINOP is binary_operation
       set evaluation_time to globally_static_evaluation
end

template PARAM_SIMPLE_NAME is name
       limit object_definition to parameter_declaration
end

template LITERAL_VALUE is literal
end

template STAT_REG_ASS is reg_assignment
             limit expression to    LOC_STAT_BINOP,
                                    GLOB_STAT_BINOP,
                                    LITERAL_VALUE,
                                    PARAM_SIMPLE_NAME
end

limit initial_reg_assignment in for_statement to STAT_REG_ASS
```

```
message "Initial reg assignment bound in for loop statements
should be statically computable "
severity ERROR
```

- Use the overflow attribute to catch cases where the left-hand side of an assignment is smaller than the right-hand side.

Reg Declaration Template

The LRM (§3.2.2) defines the grammar for reg declaration as follows:

```
reg_declaration ::= reg [range] list_of_register_identifiers ;

list_of_register_identifiers ::= register_ name {, register _name}

register_name ::= register_identifier
| memory_identifier [ upper_limit_constant_expression :
lower_limit_constant_expression ]
```

The reg_declaration template is a primary template belonging to the OBJECT_ITEM class. It contains the attributes shown in **Table 108.**

Table 108: reg_declaration Template

Attribute	Kind	Limit_Kind
comment	template	comment
declarative_region	template	REGION
expression	template	EXPRESSION
identifier	template	ID
initialized_by_constant	local	N/A
is_assigned	local	N/A
is_initialization	local	N/A
is_load	local	N/A
is_read	local	N/A
is_reset	local	N/A
is_set	local	N/A
memory_range	template	range
one_declaration_per_line	local	N/A
partly_used	local	N/A

Table 108: reg_declaration Template (Continued)

Attribute	Kind	Limit_Kind
range	template	range
driving_expression	template	EXPRESSION
signals_driven	template	DRIVEN_OBJECT
tristate	local	N/A
port	template	DECLARATIVE_ITEM
outputs_driven	max/min	N/A
unpacked_dimension_count	max/min	N/A

- Use the comment attribute to control the comments associated with variable declarations.
- Use the declarative_region attribute to control the region where the variable is declared.
- Use the one_declaration_per_line attribute to control whether multiple variables are declared on the same line.
- Use the tristate attribute to control whether the net should be tristated.
- Use the driving_expression attribute to control the expressions accepted as drivers for the current register.
- Use the signals_driven attribute to control the signals driven by the current register.
- Use the is_read attribute to make sure all bits of the signal, variable, or port specified are read at least once somewhere in the Verilog code. When you force this attribute, Leda verifies that the signal is read. For example:

```
force is_read in reg_declaration
```

- Use the port attribute to control the type of port associated with the register. For example, the following VeRSL rule:

```
template NET_WITH_PORT is reg_declaration
   no port
end
limit reg_declaration to NET_WITH_PORT
message "This reg is a redeclaration of an input/output/inout"
severity WARNING
```

causes Leda to flag an error for the following declaration of A as a register:

```
output A;
reg A;
```

Whereas, this next VeRSL rule:

```
limit port in reg_declaration to input_declaration,
   output_declaration
```

causes Leda to flag an error for all registers declared as follows:

```
inout A;
reg A;
```

- Use the partly_used attribute to detect if the vector is partially used (only some bits used). For example:

```
no partly_used in input_declaration
message "Not all the bits of the vector are used"
severity WARNING

module w498 (in,out);
input [3:0] in; //FAIL
output [2:0] out;
wire [3:0] in;

assign out[1:0] = in[1:0];

endmodule
```

- Use the is_assigned attribute to check if a reg is assigned in an always block.
- Use initialized_by_constant attribute to check if a reg is initialized by a constant on declaration.

Repeat Event Template

The LRM (§9.7.6) defines the grammar for repeat event control as follows:

```
repeat_event_control ::= repeat ( expression ) @ (event_expression)
```

The repeat_event template is a primary template belonging to no classes. It contains the attributes shown in Table 109.

Table 109: repeat_event Template

Attribute	Kind	Limit_Kind
event_control	template	event_control
repeat_expression	template	EXPRESSION

Repeat Statement Template

The repeat_statement template is a primary template belonging to the SEQUENTIAL_STATEMENT class. It contains the attributes shown in Table 110.

Table 110: repeat_statement Template

Attribute	Kind	Limit_Kind
blocking_assignment	template	blocking_assignment
case_statement	template	case_statement
casex_statement	template	casex_statement
casez_statement	template	casez_statement
conditional_statement	template	conditional_statement
disable_statement	template	disable_statement
event_trigger	template	event_trigger
expression	template	EXPRESSION
for_statement	template	for_statement
forever_statement	template	forever_statement
non_blocking_assignment	template	non_blocking_assignment
par_block	template	par_block
procedural_continuous_assign	template	procedural_continuous_assign
procedural_continuous_deassign	template	procedural_continuous_deassign
procedural_continuous_force	template	procedural_continuous_force
procedural_continuous_release	template	procedural_continuous_release
procedural_timing_control_statement	template	procedural_timing_control_statement
process_statement	template	process_statement
repeat_statement	template	repeat_statement
seq_block	template	seq_block
statement	template	SEQUENTIAL_STATEMENT
system_task_enable	template	system_task_enable
task_enable	template	task_enable

Table 110: repeat_statement Template (Continued)

Attribute	Kind	Limit_Kind
wait_statement	template	wait_statement
while_statement	template	while_statement

Resetall Template

The resetall template is a primary template belonging to no classes. It contains the attribute shown in Table 111.

Table 111: resetall Template

Attribute	Kind	Limit_Kind
enclosing_filename	template	ID

Selected_Member Template

The selected_member template is a primary template belonging to no classes. It contains the attributes shown in Table 112.

Table 112: selected_member Template

Attribute	Kind	Limit_Kind
prefix_name	template	EXPRESSION
suffix_name	template	EXPRESSION

Sensitivity_List Template

The sensitivity_list template is a primary template belonging no classes. It contains the attributes shown in Table 113.

Table 113: sensitivity_list Template

Attribute	Kind	Limit_Kind
comma	local	N/A
duplicated_sensitivity_list_expression	local	N/A
duplicated_sensitivity_list_edge_expression	local	N/A
star	local	N/A

Seq Block Template

The LRM (§9.8.1) defines the grammar for sequential block as follows:

```
seq_block ::= begin [ : block_identifier { block_item_declaration } ]
           {statement} end

block_item_declaration ::= parameter_declaration
                         | reg_declaration
                         | integer_declaration
                         | real_declaration
                         | time_declaration
                         | realtime_declaration
                         | event_declaration
```

The seq_block template is a primary template belonging to the SEQUENTIAL_STATEMENT class. It contains the attributes shown in Table 114.

Table 114: seq_block Template

Attribute	Kind	Limit_Kind
blocking_assignment	template	blocking_assignment
case_statement	template	case_statement
casex_statement	template	casex_statement
casez_statement	template	casez_statement
conditional_statement	template	conditional_statement
disable_statement	template	disable_statement
end_comment	limit	end_comment
event_trigger	template	event_trigger
identifier	template	ID
for_statement	template	for_statement
forever_statement	template	forever_statement
non_blocking_assignment	template	non_blocking_assignment
par_block	template	par_block
procedural_continuous_assign	template	procedural_continuous_assign
procedural_continuous_deassign	template	procedural_continuous_deassign
procedural_continuous_force	template	procedural_continuous_force

Table 114: seq_block Template (Continued)

Attribute	Kind	Limit_Kind
procedural_continuous_release	template	procedural_continuous_release
procedural_timing_control_statement	template	procedural_timing_control_statement
process_statement	template	process_statement
repeat_statement	template	repeat_statement
seq_block	template	seq_block
system_task_enable	template	system_task_enable
task_enable	template	task_enable
wait_statement	template	wait_statement
while_statement	template	while_statement
parameter_declaration	template	parameter_declaration
reg_declaration	template	reg_declaration
integer_declaration	template	integer_declaration
real_declaration	template	real_declaration
time_declaration	template	time_declaration
realtime_declaration	template	realtime_declaration
event_declaration	template	event_declaration
unused_declaration	local	DECLARATIVE_ITEM
statement_format	template	statement_format

Use the unused_declaration attribute to control whether a given named begin/end block should contain unused variables (for example, regs or nets). You can also use this attribute with the VeRSL limit command to control the type of the unused variable. For example:

```
no unused_declaration in seq_block
limit unused_declaration to integer_declaration
```

Example:

```
module m_no_end_comment (a, y); // FAIL
input a;
output y;
reg y;
endmodule

force end_comment in module_declaration
message "No comment after 'end...' keyword "
severity WARNING
```

Shortint Declaration Template

The shortint declaration template contains the attributes shown in Table 115.

Table 115: shortint_declaration Template

Attribute	Kind	Limit_Kind
identifier	template	ID
expression	template	EXPRESSION
signed	local	N/A
unsigned	local	N/A
static	local	N/A
automatic	local	N/A
unpacked_dimension	template	range
unpacked_dimension_count	max/min	N/A

Shortreal Declaration Template

The shortreal_declaration template contains the attributes shown in Table 116.

Table 116: shortreal_declaration Template

Attribute	Kind	Limit_Kind
identifier	template	ID
expression	template	EXPRESSION
signed	local	N/A
unsigned	local	N/A
static	local	N/A
automatic	local	N/A
unpacked_dimension	template	range
unpacked_dimension_count	max/min	N/A

Specify Block Template

The LRM (§13.1) defines the grammar for specify block as follows:

```
specify_block ::= specify [specify_item] endspecify

specify_item ::= specparam_declaration
               | path_declaration
               | system_timing_check

path_declaration ::= simple_path_declaration
                   | edge_sensitive_path_declaration
                   | state_dependent_path_declaration.
```

The specify_block template is a primary template belonging to the CONCURRENT_STATEMENT class. It contains the attribute shown in **Table 117**.

Table 117: specify_block Template

Attribute	Kind	Limit_Kind
statement_format	template	statement_format

Statement Format Template

The statement_format template is a secondary template belonging to no classes. It contains the attribute shown in Table 118.

Table 118: statement_format Template

Attribute	Kind	Limit_Kind
line_count	template	ID

Use the line_count attribute to count the number of lines in Verilog constructs that have a statement_format attribute. Here is an example for a module declaration:

```
template SF is statement_format
       max_line_count is 8
end

limit statement_format in module_declaration to SF
       message "Max line count in module declaration should be 8"
       severity ERROR
```

Struct Literal Template

You can use the struct_literal template to check the use of struct literals.

Struct Declaration Template

The struct_declaration template contains the attributes shown in Table 119.

Table 119: struct_declaration Template

Attribute	Kind	Limit_Kind
static	local	N/A
automatic	local	N/A
packed	local	N/A
packed_dimension	template	range
packed_dimension_count	max/min	N/A
signed	local	N/A
unsigned	local	N/A

Table 119: struct_declaration Template (Continued)

Attribute	Kind	Limit_Kind
struct_union_member	template	DECLARATIVE_ITEM
member_count	max/min	N/A

Synchronous Initialization Template

The synchronous_initialization template does not correspond to any part of the formal definition of the Verilog language. Instead, you use this template to constrain hardware synchronous resets inferred from the Verilog code.

The synchronous_initialization template is a primary template belonging to no classes. It contains the attributes shown in **Table 120**.

Table 120: synchronous_initialization Template

Attribute	Kind	Limit_Kind
identifier	template	ID
is_load	local	N/A
is_reset	local	N/A
is_set	local	N/A
expression	template	EXPRESSION
gated_initialization	max/min	N/A
object_definition	template	object_item
connectivity_path	template	connectivity_path
gated_in_unit	template	ID

System Function Template

The LRM (§14) defines the grammar for system function call as follows:

```
system_function ::= name_of_ system_function [(expression{, expression})]

name_of_ system_function ::= $identifier
```

The system_function template is a primary template belonging to the EXPRESSION class. It contains the attributes shown in **Table 121**.

Table 121: system_function Template

Attribute	Kind	Limit_Kind
identifier	template	ID
expression	template	EXPRESSION
expression_count	max/min	N/A

System Task Enable Template

The LRM (§14) defines the grammar for system task enable as follows:

```
system_task_enable ::= system_task_name [(expression{, expression})]

system_task_name ::= $identifier
```

The system_task_enable template is a primary template belonging to the SEQUENTIAL_STATEMENT class. It contains the attributes shown in **Table 122**.

Table 122: system_task_enable Template

Attribute	Kind	Limit_Kind
arg_decl_outside_stmt_scope	local	N/A
expression	template	EXPRESSION
identifier	template	ID
expression_count	max/min	N/A
operand_size_match	local	N/A
operand_size_match_no_carry	local	N/A

Use the operand_size_match_no_carry attribute to control whether the left- and right-hand sides of a binary operation are the same size. This attribute differs from the operand_size_match attribute in that it does not take into account the carry bit from addition or subtraction. For example:

```
force operand_size_match_no_carry in continuous_asign
wire a, b, c;
assign a = b+c; // rule does not fire here
```

Use the arg_decl_outside_stmt scope attribute to control if the arguments of system_task_enable are all wires declared in the same module.

Task Declaration Template

The LRM (§10.2.1) defines the grammar for task declaration as follows:

```
task_declaration ::=task [ automatic ] task_identifier ;
        { task_item_declaration }
        { statement }
        endtask
        | task [ automatic ] task_identifier ( task_port_list ) ;
        { block_item_declaration }
        { statement }
        endtask
task_item_declaration ::=block_item_declaration
        | { attribute_instance } input_declaration ;
        | { attribute_instance } output_declaration ;
        | { attribute_instance } inout_declaration ;
task_port_list ::= task_port_item { , task_port_item }
task_port_item ::= { attribute_instance } input_declaration
        | { attribute_instance } output_declaration
                        | { attribute_instance } inout_declaration
```

The task_declaration template is a primary template belonging to the OBJECT_ITEM class. It contains the attributes shown in **Table 123**.

Table 123: task_declaration Template

Attribute	Kind	Limit_Kind
automatic	local	N/A
end_comment	limit	end_comment
end_label	local	N/A
event_declaration	template	event_declaration
identifier	template	ID
inout_declaration	template	inout_declaration
input_count	max/min	N/A
input_declaration	template	input_declaration
integer_declaration	template	integer_declaration
output_declaration	template	output_declaration

Table 123: task_declaration Template (Continued)

Attribute	Kind	Limit_Kind
parameter_declaration	template	parameter_declaration
realtime_declaration	template	realtime_declaration
real_declaration	template	real_declaration
reg_declaration	template	reg_declaration
time_declaration	template	time_declaration
blocking_assignment	template	blocking_assignment
case_statement	template	case_statement
casex_statement	template	casex_statement
casez_statement	template	casez_statement
conditional_statement	template	conditional_statement
disable_statement	template	disable_statement
event_trigger	template	event_trigger
for_statement	template	for_statement
forever_statement	template	forever_statement
non_blocking_assignment	template	non_blocking_assignment
par_block	template	par_block
procedural_continuous_assign	template	procedural_continuous_assign
procedural_continuous_deassign	template	procedural_continuous_deassign
procedural_continuous_force	template	procedural_continuous_force
procedural_continuous_release	template	procedural_continuous_release
procedural_timing_control_statement	template	procedural_timing_control_statement
process	template	process
process_statement	template	process_statement
repeat_statement	template	repeat_statement
seq_block	template	seq_block

Table 123: task_declaration Template (Continued)

Attribute	Kind	Limit_Kind
system_task_enable	template	system_taskenable
task_enable	template	task_enable
wait_statement	template	wait_statement
while_statement	template	while_statement
side_effect	local	N/A
unused_declaration	local	DECLARATIVE_ITEM
global_signals_read	local	N/A
header_comment	template	header_comment
interface_identifier	template	ID
statement_format	template	statement_format

- Use the side_effect attribute to control whether writing to variables declared outside the task is allowed.
- Use the unused_declaration attribute to control whether a given task contains unused variables (for example, ports or registers). You can also use this attribute with a VeRSL limit command to control the type of the unused variable. For example:

```
no unused_declaration in task_declaration
limit unused_declaration to integer_declaration
```

Example:

```
module m_no_end_comment (a, y); // FAIL
input a;
output y;
reg y;
endmodule

force end_comment in module_declaration
message "No comment after 'end...' keyword "
severity WARNING
```

Task Enable Template

The LRM (§10.2.2) defines the grammar for task enable as follows:

```
task_enable ::= task_identifier [(expression {,expression}];
```

The task_enable template is a primary template belonging to the SEQUENTIAL_STATEMENT class. It contains the attributes shown in **Table 124**.

Table 124: task_enable Template

Attribute	Kind	Limit_Kind
identifier	template	ID
expression	template	EXPRESSION
expression_count	max/min	N/A
object_definition	template	OBJECT_TEM
operand_size_match_no_carry	local	N/A
port_connection	template	port_connection

- Use the port_connection attribute to control the port association in a task enable. See also the “Port Connection Template” on page 182.
- Use the operand_size_match_no_carry attribute to control whether the left- and right-hand sides of a binary operation are the same size. This attribute differs from the operand_size_match attribute in that it does not take into account the carry bit from addition or subtraction. For example:

```
force operand_size_match_no_carry in continuous_asign

wire a, b, c;
assign a = b+c; // rule does not fire here
```

Text Macro Definition Template

The LRM (§16.3) defines the grammar for text macro as follows:

```
text_macro_definition ::= define text_macro_name macro_text

text_macro_name ::= text_macro_identifier [(list_of_formal_arguments)]

list_of_formal_arguments ::= formal_argument_identifier
                        {,formal_argument_identifier}
```

The text_macro_definition template is a primary template belonging to no classes. It contains the attributes shown in **Table 125**.

Table 125: text_macro_definition Template

Attribute	Kind	Limit_Kind
text_macro_identifier	template	ID
macro_text	template	ID
enclosing_filename	template	ID

- Use the text_macro_identifier attribute to control the name of the macro.
- Use the macro_text attribute to control the macro text.

Time Declaration Template

The LRM (§3.9) defines the grammar for time declaration as follows:

```
time_declaration ::= time list_of_register_identifiers ;

list_of_register_identifiers ::= register_ name {, register _name}

register_name ::= register_identifier
                | memory_identifier [ upper_limit_constant_expression :
                lower_limit_constant_expression ]
```

The time_declaration template is a primary template belonging to the OBJECT_ITEM class. It contains the attributes shown in Table 126.

Table 126: time_declaration Template

Attribute	Kind	Limit_Kind
dimension_count	max/min	N/A
comment	template	comment

Table 126: time_declaration Template (Continued)

Attribute	Kind	Limit_Kind
declarative_region	template	REGION
is_initialization	local	N/A
identifier	template	ID
is_load	local	N/A
is_read	local	N/A
is_reset	local	N/A
is_set	local	N/A
memory_range	template	range
driving_expression	template	EXPRESSION
one_declaration_per_line	local	N/A
signals_driven	template	DRIVEN_OBJECT

- Use the comment attribute to control the comments associated with variable declarations.
- Use the declarative_region attribute to control the region where the variable is declared.
- Use the driving_expression attribute to control the expressions accepted as drivers for the current variable.
- Use the one_declaration_per_line attribute to control whether multiple variables are declared on the same line or not.
- Use the signals_driven attribute to control the signals driven by the current variable.
- Use the is_read attribute to make sure all bits of the specified signal, variable, or port are read at least once somewhere in the Verilog code. When you force this attribute, Leda verifies that the signal is read. For example:

```
force is_read in time_declaration
```

Time Units Declaration Template

The time_units_declaration template contains the attributes shown in Table 127.

Table 127: time_units_declaration Template

Attribute	Kind	Limit_Kind
timeunit	template	time_literal
timeprecision	template	time_literal

You can use the time_units_declaration template to limit the timescale used in Verilog source files, as shown in the following VeRSL code. In this example, Leda fires an error when it finds Verilog source files with a time unit or precision that is not 100 ps. You can set the desired time unit or precision by modifying the limit values in the T4 and T2 templates, respectively.

```
ruleset TIMESCALE is

template T4 is literal
set value_type to integer_literal_type
limit value to "100"
end

template T2 is time_literal
limit literal to T4
set timeunit_value to ps
end

template T1 is time_units_declaration
limit timeunit to T2
end

template T3 is time_units_declaration
limit timeprecision to T2
end

limit time_units_declaration to T1
message "time unit should be 100ps" severity ERROR

limit time_units_declaration to T3
message "time precision should be 100ps" severity ERROR

end ruleset
```

Type Declaration Template

The grammar for type declaration is:

```
type_declaration ::=typedef data_type type_declaration_identifier ;
| typedef interface_identifier { [ constant_expression ] } .
type_identifier
type_declaration_identifier ;
```

.The type_declaration template contains the attributes shown in **Table 128**.

Table 128: type_declaration Template

Attribute	Kind	Limit_Kind
identifier	template	ID
signed	local	N/A
unsigned	local	N/A
packed_dimension	template	range
packed_dimension_count	max/min	N/A
byte	local	N/A
char	local	N/A
shortint	local	N/A
int	local	N/A
longint	local	N/A
integer	local	N/A
bit	local	N/A
logic	local	N/A
reg	local	N/A
time	local	N/A
shortreal	local	N/A
real	local	N/A
realtime	local	N/A
void	local	N/A
enum	local	N/A

Table 128: type_declaration Template (Continued)

Attribute	Kind	Limit_Kind
struct	local	N/A
union	local	N/A
data_type	template	DECLARATIVE_ITEM

Unconnected Drive Template

The unconnected_drive template is a primary template belonging to no classes. It contains the attributes shown in Table 129.

Table 129: unconnected_drive Template

Attribute	Kind	Limit_Kind
pull0	local	N/A
pull1	local	N/A
enclosing_filename	template	ID

Use the pull0 and pull1 attributes with force or no commands to force or forbid the presence of pullups with values 0 and 1, respectively.

UDP Declaration Template

The LRM (§8.1) defines the grammar for UDP declaration as follows:

```
udp_declaration ::= primitive udp_identifier ( udp_port_list ) ;
                    udp_port_declaration { , udp_port_declaration }
                    udp_body
                    endprimitive

udp_portlist ::= output_port_identifier, input_port_identifier { ,
                input_port_identifier}

udp_port_declaration ::= output_declaration
                       | input_declaration
                       | reg_declaration

udp_body ::= combinational_body | sequential_body

combinational_body ::= table combinational_entry { combinational_entry}
                      endtable

combinational_entry ::= level_input_list : output_symbol ;

sequential _body ::= [udp_initial_statement]
                      table sequential _entry { sequential _entry}
                      endtable

udp_initial_statement ::= initial udp_output_port_identifier = init_val;

init_val::=1'b0 | 1'b1 | 1'bx | 1'bX | 1'B0 | 1'B1 | 1'Bx | 1'BX | 1 | 0

sequential _entry ::= seq_input_list : current_state : next_state ;

seq_input_list ::= level_input_list | edgeinput_list

level_input_list ::= level_symbol { level_symbol }

edge_input_list ::= { level_symbol } edge_indicator  { level_symbol }

edgeindicator ::= { level_symbol level_symbol } | edge_symbol

current_state ::= level_symbol

next_state ::= output_symbol | -

output_symbol ::= 0 | 1 | x | X

level_symbol ::= 0 | 1 | x | X | ? | b | B
```

The udp_declaration template is a primary template belonging to the OBJECT_ITEM class. It contains the attributes shown in **Table 130**.

Table 130: udp_declaration Template

Attribute	Kind	Limit_Kind
end_comment	limit	end_comment
identifier	template	ID
header_comment	template	header_comment
statement_format	template	statement_format
top_level	local	N/A

- Example (the argument of the -top option is a primitive name):

```
leda -top MUX2x1 <file1.v> <file2.v> ...

primitive MUX2x1 (Z, Hab, Bay, Sel); // FAIL
    output Z;
    input Hab, Bay, Sel;

table
// Hab Bay Sel Z
          1 0 0 : 0 ;
          1 0 1 : 1 ;
          1 1 0 : 1 ;
          1 1 1 : 1 ;
          0 ? ? : 0 ;
    endtable
endprimitive

template NOT_TOP_UDP is udp_declaration
    no top_level
end

limit udp_declaration to NOT_TOP_UDP
message "Top level module is a primitive"
severity WARNING
```

- Example:

```
module m_no_end_comment (a, y); // FAIL
input a;
output y;
reg y;
endmodule

force end_comment in module_declaration
message "No comment after 'end...' keyword "
severity WARNING
```

UDP Instantiation Template

The LRM (§8.1) defines the grammar for UDP instance as follows:

```
udp_instantiation ::= udp_identifier [drive_strength] [delay2]
                      udp_instance { ,
                      udp_instance }

udp_instance ::= [name_of_udp_instance]
                      (output_port_connection , input_port_connection { ,
                      input_port_connection} )

name_of_udp_instance ::= udp_instance_identifier [range]
```

The udp_instantiation template is a primary template belonging to the CONCURRENT_STATEMENT class.

Union Declaration Template

The union_declaration template contains the attributes shown in Table 131.

Table 131: union_declaration Template

Attribute	Kind	Limit_Kind
static	local	N/A
automatic	local	N/A
packed	local	N/A
packed_dimension	template	range
signed	local	N/A
unsigned	local	N/A
struct_union_member	template	DECLARATIVE_ITEM
member_count	max/min	N/A

Upward Reference Template

The upward_ref template contains the attributes shown in Table 132.

Table 132: upward_ref Template

Attribute	Kind	Limit_Kind
root_ref	local	N/A

- Use the root_ref attribute to check the use of root reference ($root) in the design. For example:

```
no root_ref in upward_ref
```

Variable Declaration Template

The grammar for variable declaration is:

```
variable_declaration ::= [ lifetime ] data_type
list_of_variable_identifiers_or_assignments ;
lifetime ::= static | automatic
```

.The variable_declaration template contains the attributes shown in Table 133.

Table 133: variable_declaration Template

Attribute	Kind	Limit_Kind
identifier	template	ID
expression	template	EXPRESSION
data_type	template	DECLARATIVE_ITEM
enum	local	N/A
struct	local	N/A
union	local	N/A
static	local	N/A
automatic	local	N/A
unpacked_dimension	template	range
unpacked_dimension_count	max/min	N/A

Void Declaration Template

. The void_declaration template contains the attributes shown in Table 134.

Table 134: void_declaration Template

Attribute	Kind	Limit_Kind
identifier	template	ID
static	local	N/A
automatic	local	N/A

Table 134: void_declaration Template (Continued)

Attribute	Kind	Limit_Kind
expression	template	EXPRESSION
signed	local	N/A
unsigned	local	N/A
unpacked_dimension	template	range
unpacked_diemnsion_count	max/min	N/A

Wait Statement Template

The LRM (§9.7.5) defines the grammar for wait statement as follows:

```
wait_statement ::= wait (expression) statement_or_null
statement_or_null ::= statement| ;
```

The wait_statement template is a primary template belonging to the SEQUENTIAL_STATEMENT class. It contains the attributes shown in Table 135.

Table 135: wait_statement Template

Attribute	Kind	Limit_Kind
blocking_assignment	template	blocking_assignment
case_statement	template	case_statement
casex_statement	template	casex_statement
casez_statement	template	casez_statement
conditional_statement	template	conditional_statement
disable_statement	template	disable_statement
event_trigger	template	event_trigger
expression	template	EXPRESSION
for_statement	template	for_statement
forever_statement	template	forever_statement
non_blocking_assignment	template	non_blocking_assignment
null_statement	template	null_statement
par_block	template	par_block

Table 135: wait_statement Template (Continued)

Attribute	Kind	Limit_Kind
procedural_continuous_assign	template	procedural_continuous_assign
procedural_continuous_deassign	template	procedural_continuous_deassign
procedural_continuous_force	template	procedural_continuous_force
procedural_continuous_release	template	procedural_continuous_release
procedural_timing_control_statement	template	procedural_timing_control_statement
process_statement	template	process_statement
repeat_statement	template	repeat_statement
seq_block	template	seq_block
statement	template	SEQUENTIAL_STATEMENT
system_task_enable	template	system_task_enable
task_enable	template	task_enable
wait_statement	template	wait_statement
while_statement	template	while_statement

While Statement Template

The while_statement template is a primary template belonging to the SEQUENTIAL_STATEMENT class. It contains the attributes shown in Table 136.

Table 136: while_statement Template

Attribute	Kind	Limit_Kind
blocking_assignment	template	blocking_assignment
case_statement	template	case_statement
casex_statement	template	casex_statement
casez_statement	template	casez_statement
conditional_statement	template	conditional_statement
disable_statement	template	disable_statement
event_trigger	template	event_trigger
expression	template	EXPRESSION
for_statement	template	for_statement
forever_statement	template	forever_statement
non_blocking_assignment	template	non_blocking_assignment
par_block	template	par_block
procedural_continuous_assign	template	procedural_continuous_assign
procedural_continuous_deassign	template	procedural_continuous_deassign
procedural_continuous_force	template	procedural_continuous_force
procedural_continuous_release	template	procedural_continuous_release
procedural_timing_control_statement	template	procedural_timing_control_statement
process_statement	template	process_statement
repeat_statement	template	repeat_statement
seq_block	template	seq_block
statement	template	SEQUENTIAL_STATEMENT
system_task_enable	template	system_task_enable
task_enable	template	task_enable

Table 136: while_statement Template (Continued)

Attribute	Kind	Limit_Kind
wait_statement	template	wait_statement
while_statement	template	while_statement

A
Template x Attribute SpecDex

About the SpecDex

The Specifier Index (SpecDex) is a double cross-referencing tool that you can use to search for information in this manual about the VeRSL templates and attributes. SpecDex is indexed two ways:

- TEMPLATE x Attribute
- ATTRIBUTE x Template

SpecDex is designed to be used as an online tool for developers writing VeRSL code.

For example, to find all the attributes that use the Always_construct template, go to the TEMPLATE x Attribute index and scroll down until you find the Always_construct template. All of the attributes of the Always_construct template are listed by page number. Click on any attribute page number to hyperlink to the information about that attribute in this manual. A traditional index is also included at the end of the manual.

TEMPLATE x Attribute

A

Asynchronous_initialization template

B

Blocking_assignment template

Byte_declaration template

C

Casex_statement template

Casez_statement template

Char_declaration template

Charge_strength template

Clock template

Cmos_switch template

Comment template

Concatenation template

Conditional_compilation template

Conditional_expression template

Conditional_statement template

Connectivity_path template

Constant_declaration template

Continuous_assign template

D

Disable_statement template

Do_while__statement template

E

Enable_gate template

End_comment template

text attribute 110

Enum_declaration template

automatic attribute 110
bit attribute 111
byte attribute 111
char attribute 111
enum_member attribute 111
int attribute 111
integer attribute 111
logic attribute 111
longint attribute 111
member_count attribute 111
packed attribute 111
packed_dimension attribute 111
reg attribute 111
shortint attribute 111
signed attribute 110
static attribute 110
unsigned attribute 111

Enum_expression template

packed_dimension attribute 111

Enum_member template

packed_dimension attribute 111

Event_control template

comma attribute 112
expression attribute 112
expression_count attribute 112

Event_declaration template

Event_trigger template

F

File_layout template

Flipflop template

For_statement template

Forever_statement template

Fsm template

Function_call template

Function_declaration template

H

Header_comment template

I

Identifier template

Include_compiler_directive template

Initial_construct template

Inout_declaration template

Input_declaration template

Int_declaration template

Integer_declaration template

Interface_declaration template

Interface_port_declaration template

J

Jump_statement template

L

Latch template

Literal template

Local_parameter_declaration template

Logic_cost template

Logic_declaration template

Longint_declaration template

M

Memory_addressing template

Mintypmax_expression template

Modport_declaration template

Module_declaration template

Module_instantiation template

Mos_switch template

Mux template

N

N_input_gate template

Non_blocking_assignment template

Nounconnected_drive template

O

Output_declaration template

P

Par_block template

Parameter_assignment template

Parameter_declaration template

Parameter_override template

Part_select template

Pass_en_switch template

Pass_switch template

Port template

Port_connection template

Posedge_event template

Procedural_continuous_assign template

Procedural_continuous_deassign template

Procedural_continuous_force template

Procedural_continuous_release template

Procedural_timing_control_statement template

R

Realtime_declaration template

Reg_assignment template

Reg_declaration template

Repeat_event template

Repeat_statement template

S

Shortint_declaration template

Shortreal_declaration template

Specify_block template

Statement_format template

Struct_declaration template

Synchronous_initialization template

System_function template

System_task_enable template

T

Task_declaration template

Task_enable template

Test_signal template

Text_macro_definition template

Time_declaration template

Time_units_declaration template

Type_declaration template

U

Udp_declaration template

Unconnected_drive template

Union_declaration template

V

Variable_declaration template

Void_declaration template

W

Wait_statement template

While_statement template

B
Attribute x Template SpecDex

About the SpecDex

The Specifier Index (SpecDex) is a double cross-referencing tool that you can use to search for information in this manual about the VeRSL templates and attributes. SpecDex is indexed two ways:

- TEMPLATE x Attribute
- ATTRIBUTE x Template

SpecDex is designed to be used as an online tool for developers writing VeRSL code.

For example, to find all the templates that use the file_name attribute, go to the ATTRIBUTE x Template index and scroll down until you find the file_name attribute. All of the templates in which file_name occurs are listed by page number. Click on any template page number to hyperlink to the information about that attribute in this manual. A traditional index is also included at the end of the manual.

ATTRIBUTE x Template

A

B

block_comment attribute

block_count attribute

blocking_assignment attribute

C

case_statement attribute

casex_statement attribute

casez_statement attribute

char attribute

comment_line attribute

comparator_cost attribute

complete_port_connection attribute

complete_sensitivity attribute

conditional_statement attribute

connectivity_path attribute

consistent_range attribute

constant_declaration attribute

constant_expression attribute

continue attribute

continuous_assign attribute

control_at_start attribute

control_src_count attribute

D

decoder_cost attribute

default attribute

default_as_last attribute

delay_control attribute

delay_value attribute

delay2 attribute

delay3 attribute

dimension_count attribute

disable_control attribute

disable_statement attribute

divide_cost attribute

do_while_statement attribute

dot_name_port_connection attribute

drive_strength attribute

driver_declaration attribute

E

enum attribute

Constant_declaration template 104
Local_parameter_delcaration template 146
Parameter_declaration template 177
Type_declaration template 215
Variable_declaration template 220

enum_identifier attribute

Constant_declaration template 104

enum_member attribute

Enum_declaration template 111

error_id attribute

Identifier template 127

evaluation_time attribute

Binary_operation template 77
Concatenation template 98
Conditional_expression template 100
Mintypmax_expression template 148

event_control attribute

Blocking_assignment template 83
Case_statement template 87
Casex_statement template 91
Casez_statement template 93
Disable_statement template 108
Non_blocking_assignment template 170
Procedural_timing_control_statement template 189
Repeat_event template 198

event_declaration attribute

event_trigger attribute

expression attribute

expression_count attribute

extention_bits attribute

extern_task_declaration attribute

F

fall_delay attribute

false_alt attribute
Conditional_statement template 102

file_layout attribute
Module_declaration template 152

file_length attribute
Module_declaration template 154

file_name attribute
Include_compiler_directive template 128

fixed_value attribute
Clock template 95
Data_signal template 56

flipflop attribute
Always_construct template 72
Bit_select template 80
Design template 57
Initial_construct template 129
Memory_addressing template 147
Name template 163
Part_select template 178

flipflop_as_source attribute
Connectivity_path template 54

for_statement attribute
Always_construct template 70
Case_statement template 86
Casex_statement template 90

forever_statement attribute

G

H

I

if_statement_count

iff_expression attribute

ignore_in_default attribute

implicit_net attribute

implicit_type attribute

instance_identifier attribute

int attribute

int_declaration attribute

integer attribute

integer_declaration attribute

is_instance attribute

is_load attribute

is_partly_assigned attribute

is_partly_used attribute

is_port attribute

is_read attribute

is_reset attribute

is_set attribute

is_type attribute

item_count attribute

iterator_modification attribute

L

latch attribute

latch_as_source attribute

left_expression attribute
Binary_operation template 77
Conditional_expression template 100

limit_id attribute
Identifier template 127

line_count attribute
Statement_format template 205

load_count attribute
Design template 56

local_parameter_declaration attribute
Interface_declaration template 136

logic attribute
Constant_declaration template 104
Enum_declaration template 111
Local_parameter_delcaration template 145
Parameter_declaration template 177
Type_declaration template 215

logic_declaration attribute
Interface_declaration template 137

logic_level attribute
Design template 57

longint attribute
Constant_declaration template 104
Enum_declaration template 111

M

memory_range attribute
Integer_declaration template 135
Reg_declaration template 196
Time_declaration template 213

meta_stability attribute
Design template 57

middle_expression attribute
Conditional_expression template 100

min_expression attribute
Mintypmax_expression template 148

minus_cost attribute
Logic_cost template 61

missing_signals_in_sensitivity_list attribute
Always_construct template 72

mixed_assignment attribute
Always_construct template 71

mixed_async_sync_resetline attribute
Design template 57

mixed_clock attribute
Design template 57

modified_sensitivity_list_variable attribute
Always_construct template 71

modport_declaration attribute
Interface_declaration template 136

modport_function_declaration attribute
Modport_declaration template 150

modport_task_declaration attribute
Modport_declaration template 150

module_declaration attribute
Module_declaration template 152

module_identifier attribute
Module_instantiation template 157

module_instantiation attribute
Interface_declaration template 137
Module_declaration template 152

modulus_cost attribute
Logic_cost template 61

moore attribute
Fsm template 119

mos_switch attribute
Interface_declaration template 137
Module_declaration template 153

msb_constant_expression attribute
Range template 192

N

nand attribute

nand_cost attribute

negedge attribute

net_declaration attribute

net_lvalue attribute

nmos attribute

non_blocking_assignment attribute

null_statement attribute

O

object_definition attribute

one_assignment_per_line attribute

one_declaration_per_line attribute

operand_size_match attribute

operand_size_match_no_carry attribute

output_declaration attribute

outputs_driven attribute

overflow attribute

P

packed attribute

packed_dimension attribute

packed_dimension_count attribute

par_block attribute

port_order attribute
Module_declaration template 154

port_reference attribute
Port template 185

port_reference_declaration attribute
Module_instantiation template 157

posedge attribute
Event_control template 112

power_cost attribute
Logic_cost template 61

pragma attribute
Case_statement template 87
Casex_statement template 90
Casez_statement template 93

prefix_name attribute
Selected_member template 200

priority attribute
Case_statement template 86
Casex_statement template 91
Casez_statement template 93
Conditional_statement template 102

procedural_continuous_assign attribute
Always_construct template 71
Case_statement template 87

procedural_continuous_deassign attribute

procedural_continuous_force attribute

procedural_continuous_release attribute

procedural_timing_control attribute

procedural_timing_control_statement attribute

process attribute

process_statement attribute

pull_gate attribute

pull0 attribute

pull1 attribute

pulldown attribute

R

rcmos attribute

reach_memory attribute

read_write attribute

real attribute

real_declaration attribute

realtime attribute

realtime_declaration attribute

recursion_type attribute

redundancy_in_sensitivity_list attribute

redundant_case_item attribute

redundant_statement attribute

reg attribute

reg_declaration attribute

reg_lvalue attribute

registered_inputs attribute

registered_outputs attribute

remainder_cost attribute

repeat_event attribute

repeat_expression attribute

repeat_statement attribute

reset_at_hierarchical_boundary attribute

reset_count attribute

return attribute

rtranif1 attribute

S

scalared attribute

sensitivity_element_is_fully_used attribute

sensitivity_element_is_incomplete attribute

sensitivity_list attribute

seq_block attribute

signals_driven attribute

signed attribute

statement_format attribute

static attribute

synchronous_initialization_signal attribute

synchronous_load_signal attribute

synchronous_reset_signal attribute

synchronous_set_signal attribute

system_task_enable attribute

T

task_declaration attribute

task_enable attribute

text attribute

text_macro_identifier attribute

text_macro_name attribute

time attribute

time_declaration attribute

time_units_declaration attribute

timeprecision attribute

timeunit attribute
Time_units_declaration template 214

top_level attribute
Udp_declaration template 218

top_module attribute
Module_declaration template 154

top_unit attribute
Design template 56

tran attribute
Pass_switch template 181

tranif0 attribute
Pass_en_switch template 181

tranif1 attribute
Pass_en_switch template 181

transition_in_default attribute
Fsm template 119

tri attribute
Net_declaration template 166

tri0 attribute
Net_declaration template 166

tri1 attribute
Net_declaration template 166

triand attribute
Net_declaration template 166

trior attribute
Net_declaration template 166

trireg attribute
Net_declaration template 166

tristate attribute
Inout_declaration template 131
Input_declaration template 132
Net_declaration template 167
Output_declaration template 172
Reg_declaration template 197

true_alt attribute
Conditional_statement template 102

truncating_extra_bits attribute
Literal template 141

truncating_leading_bits attribute
Literal template 140

turn_off_delay attribute
Delay3 template 107

typ_expression attribute
Mintypmax_expression template 148

type_declaration attribute
Interface_declaration template 136

type_declaration_identifier attribute

U

udp_instantiation attribute

union attribute

unique attribute

unit_count attribute

unpacked_dimension attribute

unpacked_dimension_count

unpacked_dimension_count attribute

unresolved attribute

unsigned attribute

unused_declaration attribute

upward_reference attribute

use_db_name attribute

V

W

wand attribute

while_statement attribute

X

Index

F

G

H

I

J

L

M

N

O

P

R

S

T

U

V

W