Leda
VRSL 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 VHDL Rule Specification Language (VRSL). If you are unfamiliar with VRSL, 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 VRSL and VHDL.

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 and appendixes:

Preface	Describes the manual and lists the typographical conventions and symbols used in it; tells how to get technical assistance.
Chapter 1 VHDL Rule Specification Language	Overview and detailed description of the VHDL Rule Specification Language (VRSL).
Chapter 2 Templates and Attributes	An API-like reference for all of the VRSL templates and attributes that you use to write rules.
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
VHDL Rule Specification Language

Introduction

This chapter provides reference information for the VHDL Rule Specification Language (VRSL). You use VRSL to write custom coding rules when there isn't a prepackaged rule that meets your design team's needs.

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 VRSL?

VRSL is a macro-based language that you use to write rules that check VHDL code for errors or anomalies. You write rules using prebuilt templates and attributes that match certain kinds of VHDL code, and a simple set of commands. There are six VRSL 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 *IEEE Standard 1076-1993 VHDL Language Reference Manual* (LRM). The LRM is the basis upon which this manual was created. Most of the VRSL keywords correspond to VHDL grammar rules and clauses, and can be found in the LRM. However, to avoid excessively long rule chains for VHDL structures such as expressions, VRSL does not always follow the LRM grammar. These deviations are noted in this manual where they apply.

VRSL'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 VHDL rules defined in the LRM and rules for a given coding style, the former are referred to as VHDL_rules in this manual.

VRSL Templates and Attributes

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

A template defines a model of how the VHDL code should appear. Templates are basic elements of VRSL code that you use to build rules or even other templates. Templates are all prepackaged (VRSL primary template or VRSL 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 VRSL commands, but you cannot create new templates or attributes yourself.

Each template has a set of attributes or characteristics of VHDL code that you can use with it. When you define a template to model the VHDL code you want to constrain, you select one or more attributes from this set and use VRSL 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.

VRSL Templatesets and Rulesets

VRSL contains two types of high-level semantic units: templatesets and rulesets. A templateset is like a VHDL package. It contains a set of template declarations. No commands are allowed in templatesets, but they can contain other templateset units. A ruleset is similar to a VHDL architecture. Rulesets can contain template declarations, commands, and other templateset units.

VRSL Rules

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

```
command attribute context message severity
```

Table 2 describes each of these components.

Table 2: VRSL Rule Components

Rule Component	Description	Type
command	Keyword that specifies how to constrain a VHDL item.	Required
attribute	Keyword that specifies the VHDL item to be constrained.	Required
context	Keyword that refines the focus of where the VHDL 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 VRSL rule:

- "VRSL Commands" on page 18
- "VRSL Attributes" on page 18
- "VRSL Context" on page 21
- "VRSL Message" on page 21
- "VRSL Severity" on page 22

VRSL Commands

Table 3 explains how each of the VRSL commands works.

Table 3: VRSL Commands

Command	Use
no	The specified VHDL_rule or clause must not be present. If it is present, the Checker flags a rule violation.
force	The specified VHDL_rule or clause must be present. If it is not present, the Checker flags a rule violation.
limit	The specified VHDL_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 VRSL command that you can use with conditional logic (for example, if ..., then ...).
set	The specified VHDL_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 VHDL_rule may appear. If the maximum number is exceeded, the Checker flags a rule violation.
min	The minimum number of times that the specified VHDL_rule must appear. If the minimum number is not met, the Checker flags a rule violation.

VRSL Attributes

An attribute represents the VHDL item that you want to constrain. There are several hundred attributes in VRSL, 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 VRSL commands, as explained in Table 4.

Table 4: Types of VRSL 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

Table 4: Types of VRSL Attributes (Continued)

Type	Description	Use Only with These Commands
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

Following are some example VRSL rules that show the attributes in **bold** for easy identification:

No Wait Statements

```
no wait_statement
```

In this example, if a wait statement appears anywhere in the design entity, the rule is violated.

No Rising Edge in Condition of Wait Statements

```
no RISING_EDGE_1 in condition of wait_statement
```

In this example. if an expression in the condition clause of a wait_statement matches the template RISING_EDGE, the rule is violated. Note that, in most cases, the attribute comes immediately after the command. However, this example, where you specify that something should not match, is an exception to this general rule.

Limit Condition to Rising Edge

```
limit condition in wait_statement to RISING_EDGE
```

In this example, if the expression in the condition of each wait statement does not match the template RISING_EDGE, the rule is violated.

Max Dimension Count in Constrained Array

```
max dimension_count in constrained_array_definition is 1
```

In this example, if the number of dimensions in a constrained array declaration is greater than 1, the rule is violated.

Attribute-Related Compilation Errors

Because VRSL constrains the kinds of attributes that you can use with the different commands, you can get a compilation error if you use an attribute with a command for which it is not allowed. For example, the following VRSL rule:

```
limit dimension_count in constrained_array_definition to 1 -- Invalid rule
```

produces a compilation error because you cannot use the dimension_count attribute with the limit command.

VRSL Context

The context specifies where the VRSL 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 VRSL rules that show the context in **bold** for easy identification:

```
no wait_statement in subprogram_body
no RISING_EDGE_1 in condition of wait_statement
limit condition in wait_statement to RISING_EDGE
max dimension_count in constrained_array_definition is 1
```

Notice 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.

Note that some contexts are not valid for a given attribute. For example:

```
no wait_statement in package_declaration -- Invalid rule
```

produces an error message because wait statements cannot appear in package declarations. If you instead specify the context as "all":

```
no wait_statement in all-- Valid rule
```

Leda checks all appropriate contexts.

VRSL Message

The message part of a VRSL 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 31.

VRSL Severity

Use the severity portion of a VRSL 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 VHDL 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 use the default value, which is note.

VRSL Examples

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

A First Example

This first example uses the VHDL wait_statement. The VHDL_rule for the wait_statement is as follows (LRM §8.1):

```
wait_statement ::=
       [ label : ] wait [ sensitivity_clause ] [ condition_clause ]
[timeout_clause ] ;
sensitivity_clause ::=  on sensitivity_list
sensitivity_list ::=  signal_name  { , signal_name  }
condition_clause ::=  until condition
condition ::=  boolean_expression
timeout_clause ::=  for time_expression
```

Notice that the wait_statement is subdivided into four clauses: label, sensitivity, condition, and timeout.

Note

Although tokens such as label are not exactly clauses, throughout this document this term is used to mean any subelement of a VHDL_rule.

Leda must be able to refine all of these clauses with different degrees of granularity. For example:

- Limit condition clauses to expressions modeling rising edges
- Forbid timeout clauses

For this first rule, assume that an expression template called RISING_EDGE_1 has been declared in the templateset CLOCK_EDGES to model rising-edge expressions.

```
ruleset SYNTHESIS is
       use templateset CLOCK_EDGES
       limit condition in wait_statement to RISING_EDGE_1
             message "SYN_8.1-3 Illegal condition in wait statement"
             severity error
end ruleset
```

To ensure that the condition clause is always present and that it matches the RISING_EDGE expression template, you could write the following commands:

```
force condition in wait_statement
       message "Condition must be present in wait statement"
       severity error
limit condition in wait_statement to RISING_EDGE_1
       message "SYN_8.1-3 Illegal condition in wait statement"
       severity error
```

Similarly, the second rule (forbid timeout clauses) is relevant to all wait statements. You can describe this in the command part as follows:

```
no timeout in wait_statement
       message "SYN8.1-4 Timeout clauses are illegal"
       severity error
```

You can define the same rules in a wait_statement template as follows:

```
template WAIT is wait_statement
       limit condition to RISING_EDGE_1
       no timeout
end
```

You could then use this template to restrict the use of wait statements in different contexts. For example:

```
limit wait_statement in process_statement to WAIT
       message "Illegal wait statement in process"
       severity error
no wait_statement in subprogram_body
       message "Wait statements are illegal in subprograms"
       severity error
```

Conditional Commands

In some cases, depending on the outcome of one rule, you may need to apply a different set of rules. To do that you can use conditional logic (for example, if ..., then ,,,) with VRSL limit commands. Note that limit commands are the only VRSL commands that you can use with conditional logic.

Suppose you want the following rules:

- Architectures may only have names “RTL” or “STRUCT”
- Component instantiations are not allowed in RTL architectures
- Process statements are not allowed in STRUCT architectures

To limit architecture body identifiers, you can declare the following templates:

```
template RTL_ARCH is architecture_body
       limit identifier to "RTL"
end
template STRUCT_ARCH is architecture_body
       limit identifier to "STRUCTURAL"
end
```

and then define the command restricting the architecture body as follows:

```
limit architecture_body in all to RTL_ARCH, STRUCT_ARCH
       message "Illegal name for architecture_body"
       severity error
```

Next, you define rules for limiting the contents of an architecture body depending on which template was matched. For the example above, you could use conditional commands as follows:

```
limit architecture_body in all to RTL_ARCH, STRUCT_ARCH
    message "Illegal name for architecture_body"
    severity error
    if RTL_ARCH then
          no component_instantiation_statement
                message "Component instantiation statements are illegal"
                severity error
    end if
    if STRUCT_ARCH then
          no process_statement
                message "Processes are illegal"
                severity error
    end if
```

So, if RTL_ARCH matches, the rule forbids component instantiation statements. If STRUCT_ARCH matches, the rule forbids process statements.

ANDing Template Matching

The previous example showed how to use the limit command to match a piece of VHDL code with one of a set of templates. Suppose, instead, that you want to match the VHDL code with all templates in a list. If you wanted to limit port map aspects so that:

- Only named associations are used, and
- No others clauses are used

you could write the rule as follows:

```
template PORT_MAP_ASSOC is association_list
       force named_association
       no others
end
limit port_map_aspect in component_instantiation_statement
       to PORT_MAP_ASSOC
       message "Illegal port map aspect"
       severity Error
```

The disadvantage of this coding style is that you have to specify a very general message with the limit rule. For more specific messages, you can use the allof keyword to indicate that you want to match against every template in the list. Then for each mismatch, you can specify a different message, as follows:

```
template NAMED_ASSOC is association_list
       force named_association
end
template NO_OTHERS is association_list
       no others
end

limit port_map_aspect in component_instantiation_statement
to allof
       NAMED_ASSOC message "Use named association in port maps",
       NO_OTHERS   message "Others clause is illegal in port maps"
       severity error
```

Aggregate Attributes

Some coding styles require that VHDL statements or clauses appear in a certain order. For example, some synthesis tools can optimize processes written according to a certain profile. Consider the following VHDL code:

```
FF_ARST: process (CLK,RST)
begin
       if RST='1' then
             Q<='0';
       elsif CLK='1' AND CLK' event then
             Q<=D;
       end if;
end process FF_ARST;
```

This represents a flip-flop with an asynchronous reset. The process contains a sensitivity list (no waits) and a single statement in its outermost region. The if statement has two conditions (no else), and these two conditions are expressions of a predefined format: the first is a high value on RST and the second is a rising edge on CLK.

Because you often need to assign different expressions to different conditions of an if statement, VRSL has the concept of aggregate attributes. These are sets of attributes that can be tested against different templates for each item. For example, the following command limits if statement conditions to those described above:

```
limit condition_s in if_statement to
       first HIGH_LEVEL, then RISING_EDGE_1
       message "Illegal condition in if statement"
       severity error
```

In this example, HIGH_LEVEL and RISING_EDGE_1 are predefined expression templates that are located in the CLOCK_EDGES templateset.

Aggregate attributes always have an "_s" suffix. Other aggregate attributes include:

- association_element_s
- conditional_waveforms_s
- element_association_s
- index_constraint_s
- selected_waveforms_s
- statement_profile_s
- waveform_element_s

You can use aggregate attributes as ordinary attributes by assigning a single list of templates and removing the first keyword. For example, the following code means that every condition alternative is checked against both predefined templates.

```
limit condition_s in if_statement to HIGH_LEVEL, RISING_EDGE_1
```

You can also test different occurrences of the same VHDL structure against the same set of templates. Suppose in the example above you want to restrict the statements in the outermost statement region of the process to any number of variable assignment statements (corresponding to initializations) followed by the if statement. You can write this rule as follows:

```
limit statement_profile_s in process_statement to
       first {VAR_ASS_STM}, then IF1
       message "Illegal process profile"
       severity error
```

where VAR_ASS_STM is a previously defined template for variable_assignment_statements and IF1 is an if_statement template limiting the condition_s attribute as shown above.

Putting braces around the template name**, {**VAR_ASS_STM**},** means that all VHDL statements are checked against VAR_ASS_STM until a mismatch occurs. If the VHDL statement that causes the mismatch doesn't match IF1, you get an error. If it does match,

then Leda moves on to the next set of templates. For example, suppose you want every element in an association list to match with the same association_element template.You can write the VRSL rule for this as follows:

```
template ASSOC_ELT is association_element
       no formal_function
end
template ASSOC_LIST is association_list
       limit association_element_s to {ASSOC_ELT}
end
limit association_list to ASSOC_LIST
       message "Illegal association_list"
       severity error
```

The braces around {ASSOC_ELT} in this example mean that the VHDL code is not restricted. You should only use braces around a template names with the last “then” alternative.

Parameterizing Rules Using Regular Expressions

Naming conventions are not part of VHDL, but such rules frequently appear in coding standards. To support rules for naming convention rules, 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 labels of component instantiation statements so that they have one of the following formats:

- <instantiated entity name>0 or
- <instantiated entity name>_<index>
- <instantiated entity name>_<more_info>_<index>

where <index> is an integer value and <more_info> is a user-defined string.

The following VRSL command correctly tests the instantiation statements:

```
limit label in component_instantiation_statement to
                    "^<entity>0$",
                    "^<entity>_[0-9]*$",
                    "^<entity>_[.]*_[0-9]*$"
```

where <entity> is a predefined parameter that is replaced by the name of the enclosing entity.

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) specifies that clock signals must have the prefix c*l*k_. In the RMM policy, this rule is written 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 used or that used another naming convention for clock signals. You can extend the rule definition using VRSL to include a variable parameter as follows:

```
template CLOCK_ID is clock
  limit identifier to "<clock_name>"
end
```

You then must specify a value for the <clock_name> macro for the 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*.

You can use predefined macros in regular expressions. For example, Leda replaces the predefined parameter <entity> in a regular expression with the name of the enclosing entity, if any. If you write the following rule:

```
--the architecture's name must be prefixed by the entity name
template AB is architecture_body
       limit identifier to "<entity>_[a-zA-Z0-9]+"
end
limit architecture_body to AB...
```

and have an entity called TOP, then Leda flags the architecture name RTL as an error, but accepts the name TOP_RTL. Similarly, you can use the following predefined parameters in regular expressions:

- architecture
- configuration

- package
- library
- component (name of component in component instantiation)
- formal (name of formal parameter in a map aspect)
- target (name of left-hand side in an assignment statement)

Constrain Max/Min Attributes to Predefined Values

You use max and min commands to control the number of elements in a given VHDL clause. For example, suppose you want ranges always to be descending downto 0 (for example, 5 downto 0...). You can write a rule to specify this as follows:

```
max low_bound in range is 0
```

You can also constrain a max/min attribute to a value defined in a macro. This rule then becomes a generic rule that can be used by multiple users, each with their own parameters. To change the value of the macro, use a rule_set_parameter command in your config.tcl file. For more information, see the *Leda User Guide*.

The VRSL code itself stays the same, so you don't need to recompile the rule. For example, you could use a macro to implement the previous rule as follows:

```
max low_bound in range is "<range_low_value>"
```

and in the config.tcl file:

```
rule_set_parameter -rule rule_label -parameter range_low_value -value 0
```

Note

You must write the macros within inverted commas in the VRSL code, but do not treat them as regular expressions.

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 VRSL 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 VRSL 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 VRSL, 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.

The following VRSL example shows how to specify the location of HTML help documents:

```
limit file_name in architecture_body to "<entity>_<architecture>.vhd"
       message "Illegal file name for architecture"
             html_document"file.html#address"
             html_note    "note.html#address"
       severity error
```

Note that the HTML file locations are part of the message command. As such, they must only appear after the keyword "message" in a VRSL rule.

Parameterizing Error Messages

You can make error messages for Leda rules more informative by using parameters in the 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 VRSL 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 VRSL command (no, force, limit, set, max, min). For limitations on the use of **<%item>**, see "Limitations on use of <%item>" on page 33.

	Example VRSL 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 VRSL code, Leda cannot determine a more specific context.	Any VRSL command (no, force, limit, set, max, min).

	Example VRSL 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 VHDL 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 VRSL 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 VRSL command (no, force, limit, set, max, min).

	Example VRSL 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 only works 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:

- alias_declaration
- asynchronous_reset
- attribute_declaration

- attribute_name
- clock
- complete_sensitivity
- component_declaration
- component_instantation_statement
- configuration_declaration
- constant_declaration
- element_declaration
- entity_declaration
- file_declaration
- file_layout
- flipflop
- formal_parameter
- function_call
- gated_clock
- gated_reset
- generate_statement
- generic_declaration
- indexed_name
- interface_file_declaration
- interface_variable_declaration
- latch
- missing_signals_in_sensitivity_list
- mixed_async_sync_resetline
- multiplexed_clock
- object_definition
- operator
- package_declaration
- port_declaration

- redundancy_in_sensitivity_list
- registered_outputs
- selected_name
- shared_variable_declaration
- signal_declaration
- simple_name
- slice_name
- subprogram_declaration
- subtype_declaration
- synchronous_reset
- type_declaration
- variable_declaration

Formal Definition of VRSL

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

A VRSL description is a set of library units. There must be at least one ruleset unit, as this is where the commands are defined. This is summarized in the syntax:

```
policy ::= {leda_library_unit}
leda_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, similar to a VHDL package declaration. 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 used 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, similar to a VHDL architecture. 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 VHDL_rules (rules based on 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 VHDL_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
                 | aggregate_attribute
```

The attribute kind 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
                                aggregate_template_decl_limit_command
complete_template_decl_limit_command ::=
                          limit template_kind to one_of_limit_list
aggregate_template_decl_limit_command ::=
       limit aggregate_attribute to template_aggregate_list
template_aggregate_list ::= first template_id_list
                            {then template_id_list }
                            [others template_id_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.

When an attribute is constrained to templates of type identifier, the string used in the limit_id attribute of this template can replace the identifier template itself. For example, the two commands below are identical.

```
template ID is identifier
        limit limit-id to ”<entity>.vhd”
end

limit file_name in entity_declaration to ID...
limit file_name in entity_declaration to ”<entity>.vhd”...
```

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

In this VRSL template declaration example, the left_expression and right_expression attributes have the same limitations. They can only point to expressions of type simple_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 ::= evaluation_period
                        | literal_type
evaluation_period::= unresolved
                   | locally_static_evaluation
                   | globally_static_evaluation
                   | dynamic_evaluation
literal_type ::= any_literal_type
                | abstract_literal_type
                | physical_literal_type
                | enumeration_literal_type
                | bit_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 max and min commands to control the number of elements in a given VHDL clause. For example, suppose if you want a range always to be descending to 0 (for example, 5 downto 0...). You can write this rule as follows:

```
template DESCENDING_RANGE is range
       no    to
       max   low_bound is 0
       min   low_bound is 0
end
```

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

```
template SEQ_PSS is process_statement
       max wait_statement is 1
end
```

Command Part

You define subset rules in the command part of a rule_file. The syntax 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_address_string"
html_app ::= "html_address_string"
```

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

```
no wait_statement in subprogram_body
```

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

```
template SUBP_WAIT is wait_statement
       no timeout
end
no SUBP_WAIT in wait_statement of subprogram_body
```

This example rule means that wait statements are allowed in subprogram bodies 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 VHDL 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
                | aggregate_limit_command
complete_limit_command ::= [label :] limit template_kind
[command_context] to limit_list
       [message] [severity]
       {conditional_rule_block}
aggregate_limit_command ::= [label :] limit aggregate_attribute
[command_context] to template_aggregate_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 VHDL_rule in the input VHDL 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_attribute  is number
                [message][severity]
```

You use the max and min commands to control the number of elements in a given VHDL clause. For example, suppose you want a range always to be descending to 0 (for example, 5 downto 0...). You can write this rule as follows:

```
max low_bound in range is 0
```

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 process, you could write the following rule:

```
max wait_statement in process_statement is 1
```

VRSL Primitives

The following primitive types are defined in VRSL:

Literal_Type

Enumerates the different types that a literal value can take. This enumerated type is pointed to by the value_type attribute of the literal template. The literal_type values are:

- Any_Literal_Type—An abstract literal can only be a decimal or based number. For example: 12, 123-456, 12-0, 6.023E+24, 2#11, 16#F.FF#E+2
- Abstract_Literal_Type—A physical literal is an abstract literal followed by a unit name. For example: 10ns.
- Physical_Literal_Type—A physical literal is an abstract literal followed by a unit name; for example: 10ns.
- Enumerated_Literal_Type—An enumerated literal is an element of an enumeration type. For example: TRUE or '1'.
- String_Literal_Type—A string literal is a string of characters. For example: "hello world".
- Bit_String_Literal_Type—A bit string literal is a group of character literals. For example: "1111" or B "0011".

Evaluation_Periods

Enumerates the different evaluation times for expressions. This enumerated type is pointed to by the evaluation_time attribute. The evaluation_periods values are:

- Undefined
- Locally_Static_Evaluation—Locally static expressions are those that can be evaluated at compile time. For example: 1+2.
- Globally_Static_Evaluation—Globally static expressions are those that can only be evaluated after elaboration when generic values are known and functions have been executed. For example: foo(3).
- Dynamic_Evaluation—Dynamic expressions are those that can only be evaluated during simulation. For example, those that include signals or variables.

Error_Status

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

- Note
- Warning
- Error
- Fatal

Recursion_Type

Enumerates the different recursion types allowed by the attribute recursion_type. The recursion_type values are:

- all_recursion
- static_recursion
- no_recursion

Edge_Type

Enumerates the different values the edge attribute can have. This attribute is used by clock and reset nodes to model the active edge of the corresponding signal. The edge_type values are:

- Undefined_Edge
- Rising_Edge
- Falling_Edge
- Both_Edges
- High_Level
- Low_Level

VRSL Classes

Each VRSL template belongs to one or more classes, or different types of VHDL code. Classes are also used to specify a group of template kinds that are valid for a given attribute. Each template belongs to one or more classes. Classes are used to indicate 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 may also accept string literals as values. For example, for the following template description:

```
template STD_BIT_ID is identifier
       limit limit_id to "STD.STANDARD.BIT"
end
template STD_BIT is type_declaration
       limit identifier to STD_BIT_ID
end
```

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

```
template STD_BIT is type_declaration
       limit identifier to "STD.STANDARD.BIT"
end
```

The following classes are defined in VRSL:

- “TYPE_MARK” on page 54
- “TYPE_DEFINITION” on page 54

CONCURRENT_STATEMENT

The CONCURRENT_STATEMENT class includes the following templates:

- concurrent_assertion_statement
- block_statement
- component_instantiation_statement
- conditional_signal_assignment
- generate_statement
- procedure_call_statement
- process_statement
- selected_signal_assignment

DECLARATIVE_ITEM

The DECLARATIVE_ITEM class includes the following templates:

- alias_declaration
- attribute_declaration
- attribute_specification
- component_declaration
- constant_declaration
- disconnection_specification
- file_declaration
- group_declaration
- group_template_declaration
- name_prefix
- shared_variable_declaration
- signal_declaration
- subprogram_body

- subprogram_declaration
- subtype_declaration
- type_declaration
- use_clause
- variable_declaration

DISCRETE_RANGE

The DISCRETE_RANGE class includes the following templates:

- subtype_indication
- range

DRIVEN_OBJECT

The DRIVEN_OBJECT class includes the following templates:

- port_declaration
- formal_parameter
- signal_declaration

EXPRESSION

The EXPRESSION class includes the following templates:

- aggregate
- allocator
- binary_operation
- expression
- function_call
- literal

  SUBCLASS: NAME

  The NAME subclass includes the following templates:

  - attribute_name
  - indexed_name

- selected_name
- simple_name
- slice_name

ID

The ID class includes the following template:

- identifier

NAME

The NAME class includes the following templates:

- attribute_name
- indexed_name
- selected_name
- simple_name
- slice_name

NAMED_ENTITY

The NAMED_ENTITY class includes the following templates:

- design_unit
- function_call

  SUBCLASS: TYPE_MARK

  The TYPE_MARK subclass includes all VRSL templates.

  SUBCLASS: OBJECT_ITEM

  The OBJECT_ITEM subclass includes all VRSL templates.

  SUBCLASS: DECLARATION_ITEM

  The DECLARATION_ITEM subclass includes all VRSL templates.

OBJECT_ITEM

The OBJECT_ITEM class includes the following templates:

- alias_declaration
- architecture_body
- attribute_declaration
- component_declaration
- constant_declaration
- configuration_declaration
- entity_declaration
- file_declaration
- formal_parameter
- interface_constant_declaration
- interface_file_declaration
- name_prefix
- port_declaration
- generic_declaration
- package_declaration
- shared_variable_declaration
- signal_declaration
- subprogram_declaration
- variable_declaration

REGION_PART

The REGION_PART class includes the following templates:

- architecture_body
- block_configuration
- block_specification
- block_statement
- component_declaration

- configuration_declaration
- entity_declaration
- for_loop_statement
- generate_statement
- loop_statement
- package_body
- package_declaration
- process_statement
- subprogram_body
- subprogram_declaration
- while_loop_statement

SEQUENTIAL_STATEMENT

The SEQUENTIAL_STATEMENT class includes the following templates:

- assertion_statement
- case_statement
- exit_statement
- for_loop_statement
- if_statement
- loop_statement
- next_statement
- procedure_call_statement
- report_statement
- return_statement
- signal_assignment_statement
- variable_assignment_statement
- wait_statement
- while_loop_statement

STATEMENT

The STATEMENT class includes the following subclasses:

- CONCURRENT_STATEMENT

  The CONCURRENT_STATEMENT subclass includes the following templates

  - concurrent_assertion_statement
  - block_statement
  - component_instantiation_statement
  - conditional_signal_assignment
  - generate_statement
  - procedure_call_statement
  - process_statement
  - selected_signal_assignment

- SEQUENTIAL_STATEMENT

  The SEQUENTIAL_STATEMENT subclass includes the following templates

  - assertion_statement
  - case_statement
  - for_loop_statement
  - if_statement
  - loop_statement
  - procedure_call_statement
  - report_statement
  - signal_assignment_statement
  - variable_assignment_statement
  - wait_statement
  - while_loop_statement

TARGET

The TARGET class includes the following templates:

- aggregate
- indexed_name
- selected_name
- simple_name
- slice_name

TYPE_MARK

The TYPE_MARK class includes the following templates:

- identifier
- name_prefix
- type_declaration
- subtype_declaration

Note that attributes of type TYPE_MARK can also take string literals as parameters.

TYPE_DEFINITION

The TYPE_DEFINITION class includes the following templates:

- access_type_definition
- constrained_array_definition
- enumeration_type_definition
- file_type_definition
- floating_type_definition
- integer_type_definition
- physical_type_definition
- record_type_definition
- unconstrained_array_definition

2 Templates and Attributes

Introduction

This chapter presents an API-like reference for the VRSL 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 also 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:

Connectivity Path Template

A lot of rules concern what appears on the path of control signals. This can be anything from gated clocks, to counting the number of inverters, to forbidding buffers. Each control signal therefore has a connectivity_path attribute which points to a set of templates of this type. Some design for test (DFT) rules concern what appears on the path of data or control signals. Also, some rules imply the analysis of a data signal 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 domain. Combined with the flipflop_as_source or latch_as_source attributes, this attribute was used 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 VRSL 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, 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. You can only use the local attributes in this template with the force and no commands. The design template also 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 just 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
top_unit	template	REGION_PART
asynchronous_feedback	local	N/A
asynchronous_logic	local	N/A
drivers_per_signal	max/min	N/A

Table 8: design Template (Continued)

Attribute	Kind	Limit_Kind
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
load_count	max/min	N/A
multiplexed_clock	local	N/A
reset_count	max/min	N/A
gated_reset	local	N/A
initialization_count	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
set_count	max/min	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 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.
- 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. 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
```

Flipflop Template

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
cased_identifier	template	ID
identifier	template	ID
tinput	template	EXPRESSION
output	template	EXPRESSION
object_definition	template	OBJECT_ITEM
synchronous_initialization	template	synchronous_initialization

Table 9: flipflop Template (Continued)

Attribute	Kind	Limit_Kind
clock	template	clock
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 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 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	template	asynchronous_initialization
cased_identifier	template	ID
identifier	template	ID
input	template	EXPRESSION
output	template	EXPRESSION
object_definition	template	OBJECT_ITEM
synchronous_initialization	template	synchronous_initialization
clock	template	clock
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_971 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 specify 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 program 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 VRSL 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 VHDL constructs to ensure that they correspond to acceptable values, ranges, or conventions.
- **Hardware-based rules**—use to control the hardware semantics of VHDL. Certain VHDL 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:

- "Access Type Definition Template" on page 70
- "Aggregate Template" on page 70
- "Alias Declaration Template" on page 71
- "Allocator Template" on page 72

Access Type Definition Template

The LRM (§3.3.2) defines the grammar for access type definition as follows:

```
access_type_definition ::=  access subtype_indication
```

Access types are associated with the allocation and deallocation of objects. The access_type_definition template is a primary template belonging to no classes. It contains the attribute shown in **Table 13**.

Table 13: access_type_definition Template

Attribute	Kind	Limit_Kind
subtype_indication	template	subtype_indication

Aggregate Template

The LRM (§7.3.2) defines the grammar for aggregate as follows:

```
aggregate ::= ( element_association { , element_association } )
```

The aggregate template is a primary template belonging to the EXPRESSION class. It contains the attributes shown in **Table 14**.

Table 14: aggregate Template

Attribute	Kind	Limit_Kind
choice	template	EXPRESSION
positional_association	local	N/A
named_association	local	N/A
multiple_choices	local	N/A
record_aggregate	local	N/A

Table 14: aggregate Template (Continued)

Attribute	Kind	Limit_Kind
others	local	N/A
element_count	max/min	N/A
element_association_s	aggregate	element_association

You can use the choice attribute and any of the local attributes to constrain every element_association template. Or, you can add these constraints to each element_association template individually to achieve the same effect.

Alias Declaration Template

The LRM (§4.3.3) defines the grammar for alias declaration as follows:

```
alias_declaration ::=
alias alias_designator [ : subtype_indicator ] is name [ signature ] ;
alias_designator ::= identifier | character_literal | operator_symbol
```

The alias_declaration template is a primary template belonging to the OBJECT_ITEM class. It contains the attributes shown in **Table 15**.

Table 15: alias_declaration Template

Attribute	Kind	Limit_Kind
identifier	template	ID
subtype_indication	template	subtype_indication
name	template	NAME
declarative_region	template	REGION_PART
cased_identifier	template	ID
character_literal	local	N/A
operator_symbol	local	N/A
one_declaration_per_line	local	N/A
comment	local	N/A

Use the cased_identifier attribute just like the identifier attribute, but in cases where the identifier is case-sensitive. Consider the following templates:

```
template T1 is alias_declaration
       limit identifier to "^[a-z][a-z0-9_]*$"
end
template T2 is alias_declaration
       limit cased_identifier to "^[a-z][a-z0-9_]*$"
end
```

The id1 identifier matches both templates, whereas the ID1 identifier only matches the first template. The identifier attribute is not case-sensitive because VHDL is not case-sensitive.

Use the one_declaration_per_line attribute to force a single line for each declaration.

Use the comment attribute to force a comment on the same line as the declaration.

Allocator Template

The LRM (§7.3.6) defines the grammar for allocator as follows:

```
allocator ::= new subtype_indication | new qualified_expression
```

The allocator template is a secondary template belonging to the EXPRESSION class. It contains the attributes shown in **Table 16**.

Table 16: allocator Template

Attribute	Kind	Limit_Kind
subtype_indication	template	subtype_indication
expression	template	EXPRESSION

Architecture Body Template

The LRM (§1.2) defines the grammar for architecture body as follows:

```
architecture_body ::=
       architecture identifier of entity_name is
             architecture_declarative_part
       begin
             architecture_statement_part
       end [ architecture ] [ architecture_simple_name ] ;
```

The LRM (§1.2.1) defines the grammar for the architecture declarative part as follows:

```
architecture_declarative_part ::=
       { block_declarative_item }
block_declarative_item ::=
          subprogram_declaration
       | subprogram_body
       | type_declaration
       | subtype_declaration
       | constant_declaration
       | signal_declaration
       | shared_variable_declaration
       | file_declaration
       | alias_declaration
       | attribute_declaration
       | attribute_specification
       | configuration_specification
       | disconnection_specification
       | use_clause
       | group_template_declaration
       | group declaration
```

The LRM (§1.2.2) defines the grammar for the architecture statement part as follows:

```
architecture_statement_part ::= { concurrent_statement }
```

Architecture bodies define the body of a design entity. The architecture_body template is a primary template belonging to no classes. It contains the attributes shown in **Table 17**.

Table 17: architecture_body Template

Attribute	Kind	Limit_Kind
identifier	template	ID
cased_identifier	template	ID
subprogram_declaration	template	subprogram_declaration
subprogram_body	template	subprogram_body

Table 17: architecture_body Template (Continued)

Attribute	Kind	Limit_Kind
type_declaration	template	type_declaration
statement_format	template	statement_format
subtype_declaration	template	subtype_declaration
constant_declaration	template	constant_declaration
related_unit	local	N/A
shared_variable_declaration	template	shared_variable_declaration
signal_declaration	template	signal_declaration
file_declaration	template	file_declaration
alias_declaration	template	alias_declaration
component_declaration	template	component_declaration
attribute_declaration	template	attribute_declaration
attribute_specification	template	attribute_specification
configuration_specification	template	configuration_specification
disconnection_specification	template	disconnection_specification
use_clause	template	use_clause
group_template_declaration	template	group_template_declaration
group_declaration	template	group_declaration
concurrent_assertion_statement	template	concurrent_assertion_statement
concurrent_procedure_call_statement	template	concurrent_procedure_call_statement
process_statement	template	process_statement
block_statement	template	block_statement
selected_signal_assignment	template	selected_signal_assignment
conditional_signal_assignment	template	conditional_signal_assignment
component_instantiation_statement	template	component_instantiation_statement
generate_statement	template	generate_statement

Table 17: architecture_body Template (Continued)

Attribute	Kind	Limit_Kind
design_unit	template	design_unit
header_comment	template	header_comment
entity_declaration	template	entity_declaration
top_architecture	template	architecture_body
file_name	template	ID
ncs_file_name	template	ID
file_layout	template	file_layout
unused_declaration	local	N/A
line_count	max/min	N/A
file_length	max/min	N/A
clock_signal	max/min	N/A
synchronous_reset_signal	max/min	N/A
asynchronous_reset_signal	max/min	N/A
statement_profile_s	aggregate	CONCURRENT_STATEMENT
declaration_profile_s	aggregate	DECLARATIVE_ITEM
fsm	template	fsm

Employ the use_clause attribute to limit use_clauses inside the declaration. To constrain use clauses (and library clauses) appearing before the declaration, use the design_unit template instead.

There are some application-specific attributes associated with this template that make it easier to write commonly-used rules. For example:

- Use the entity_declaration attribute to constrain the architecture entity.
- Use the top_architecture attribute to constrain the top-level architecture. To make this work, you must specify the name of the top-level architecture using the –top option of the command-line Checker.

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

```
limit file_name in architecture_body to "<architecture>.vhd"
```

- Use the unused_declaration attribute to test for unused declarations. For example:

```
no unused_declaration in architecture_body
```

- Use the line_count attribute to specify the maximum or minimum lines of code in a unit.
- Use the clock_signal attribute to specify the maximum or minimum number of clock signals in an architecture.
- Use the asynchronous_reset_signal attribute to specify the maximum or minimum number of asynchronous reset signals in an architecture.
- Use the synchronous_reset_signal attribute to specify the maximum or minimum number of synchronous reset signals in an architecture.
- Use the statement_profile_s attribute to specify the order of statements. For example:

```
limit statement_profile_s in architecture_body to
    first       COMP_INST_STM
    then        PROCESS_STM
```

- Use the declaration_profile_s attribute to specify the order of declarations. For example:

```
limit declaration_profile_s in architecture_body to
    first       type_declaration
    then        subtype_declaration
    then        {}
```

Use the cased_identifier attribute just like the identifier attribute, but in cases when the identifier is case-sensitive. Consider the following templates:

```
template T1 is architecture_body
       limit identifier to "^[a-z][a-z0-9_]*$"
end
template T2 is architecture_body
       limit cased_identifier to "^[a-z][a-z0-9_]*$"
end
```

The identifier id1 matches both templates, whereas the identifier ID1 only matches the first template. The attribute identifier is not case-sensitive because VHDL is not case-sensitive.

- Use the related_unit attribute to check that the architecture or the configuration you can find in a file is related to the entity described in that same file.

Assertion Statement Template

The LRM (§8.2) defines the grammar for assertion statement as follows:

```
assert_statement ::= [ label : ] assertion ;

assertion ::= assert condition
              [ report expression ]
              [ severity expression ]
```

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

Table 18: assertion_statement Template

Attribute	Kind	Limit_Kind
label	template	ID
cased_label	template	ID
condition	template	EXPRESSION
report_expression	template	EXPRESSION
severity_expression	template	EXPRESSION
declarative_region	template	REGION_PART
postponed	local	N/A

Use the cased_label attribute just like the label attribute, but in cases where the label is case-sensitive. Consider the following templates:

```
template T1 is report_statement
       limit label to "^[a-z][a-z0-9_]*$"
end
template T2 is report_statement
       limit cased_label to "^[a-z][a-z0-9_]*$"
end
```

The lab1 label matches both templates, whereas the LAB1 label only matches the first template. The label attribute is not case-sensitive because VHDL is not case-sensitive.

Association Element Template

The LRM (§4.3.2.2) defines the grammar for association element as follows:

```
association_element ::= [ formal_part => ] actual_part
formal_part ::=          formal_designator
                         | function_name ( formal_designator )
                         | type_mark ( formal_designator )
formal_designator ::= generic_name | port_name | parameter_name
actual_part ::=           actual_designator
                         | function_name ( actual_designator )
                         | type_mark ( actual_designator )
actual_designator ::=     expression
                         | signal_name
                         | variable_name
                         | open
```

The association_element template is a primary template belonging to no classes. It contains the attributes shown in **Table 19.**

Table 19: association_element Template

Attribute	Kind	Limit_Kind
actual_designator	template	EXPRESSION
named_association	local	N/A
positional_association	local	N/A
range_overflow	local	N/A
open	local	N/A
others	local	N/A
formal_function	local	N/A
formal_type_mark	local	N/A
actual_function	local	N/A
actual_type_mark	local	N/A

Some of the attributes in association list templates also appear in association element templates, allowing finer control over these elements. To force positional (or named) association, you can use a command like the following:

```
force positional_association in association_element
```

Use the formal_function and formal_type_mark attributes to force or prohibit the presence of function calls or type conversions in formal parameters.

Use the actual_function and actual_type_mark attributes to force or prohibit the presence of function calls or type conversions in actual parameters.

Use the open attribute to force or prohibit the use of the “open” keyword in actual parameters.

Association List Template

The LRM (§4.3.2.2) defines the grammar for association list as follows:

```
association_list ::= association_element { , association_element }
```

When you call a subprogram or instantiate a component, you use VHDL association lists to establish the correspondence between formal or local generic, port, or parameter names on the one hand, and local or actual names or expressions on the other.

The association_list template is a primary template belonging to the no classes. It contains the attributes shown in **Table 19.**

Table 20: association_list Template

Attribute	Kind	Limit_Kind
named_association	local	N/A
positional_association	local	N/A
open	local	N/A
read_write	local	N/A
others	local	N/A
parameter_count	max/min	N/A
association_element_s	aggregate	association_element

Use the parameter_count attribute to set a maximum or minimum number of elements for an association list. For example:

```
min parameter_count in association_list is 2
```

Use the named_association and the positional_association attributes to set named or positional associations on each element of an association list.

Use the open attribute to control the use of the open VHDL keyword in the association list. For example, the rule:

```
no open in association_list severity error
```

means that the open keyword is prohibited in association lists.

Use the read_write attribute to control the use of actual parameters in the association list. Consider the following rule:

```
no read_write in association_list severity error
```

This rule ensures that the same actual parameter is not assigned to formals of both mode in and mode out or inout.

Use the association_element_s attribute to point to templates of type association_element. This gives you extra control over individual elements in the list.

Asynchronous Initialization Template

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

Table 21: asynchronous_initialization Template

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

Leda recognizes all RTL reset expressions. However, if you want to limit the expression type, use the expression attribute.

Use the object_definition attribute to limit the type of the reset signal.

Use the cased_identifier attribute just like the identifier attribute, but in cases where the identifier is case-sensitive. Consider the following templates:

```
template T1 is asynchronous_initialization
       limit identifier to "^[a-z][a-z0-9_]*$"
end
       template T2 is asynchronous_initialization
limit cased_identifier to "^[a-z][a-z0-9_]*$"
end
```

The id1 identifier matches both templates, whereas the ID1 identifier only matches the first template. The identifier attribute is not case-sensitive because VHDL is not case-sensitive.

Note

If you use a complex expression (containing more than one object) as a clock or reset, Leda automatically deactivates naming convention rules. This is because Leda assumes that the clock or reset is the output of a logic gate inferred by the complex expression, and not simply one of the objects in the expression.

Attribute Declaration Template

The LRM (§4.4) defines the grammar for attribute declaration as follows:

```
attribute_declaration ::= attribute identifier : type_mark ;
```

The attribute_declaration template is a primary template belonging to the OBJECT_ITEM class. It contains the attributes shown in **Table 22.**

Table 22: attribute_declaration Template

Attribute	Kind	Limit_Kind
cased_identifier	template	ID
identifier	template	ID
type_mark	template	TYPE_MARK
declarative_region	template	REGION_PART
one_declaration_per_line	local	N/A
comment	local	N/A

Use the cased_identifier attribute just like the identifier attribute, but in cases where the identifier is case-sensitive. Consider the following templates:

```
template T1 is attribute_declaration
       limit identifier to "^[a-z][a-z0-9_]*$"
end
template T2 is attribute_declaration
       limit cased_identifier to "^[a-z][a-z0-9_]*$"
end
```

The id1 identifier matches both templates, whereas the ID1 identifier only matches the first template. The attribute identifier is not case-sensitive because VHDL is not case-sensitive.

Use the one_declaration_per_line attribute to force a single line for each declaration.

Use the comment attribute to force a comment on the same line as the declaration.

Attribute Name Template

The LRM (§6.6) defines the grammar for attribute name as follows:

```
attribute_name ::=
   prefix [ signature ]' attribute_designator [ ( expression ) ]
   attribute_designator ::=  attribute_simple_name
```

The attribute_name template is a primary template belonging to the EXPRESSION, NAME, and TARGET classes. It contains the attributes shown in **Table 23**.

Table 23: attribute_name Template

Attribute	Kind	Limit_Kind
name_prefix	template	EXPRESSION
named_entity_prefix	template	NAMED_ENTITY
expression	template	EXPRESSION
attribute_designator	template	attribute_declaration

The named_entity_prefix attribute ensures full compatibility with the LRM, something the name_prefix attribute does not do. Use the named_entity_prefix attribute to constrain prefixes that are types. For example:

```
integer'image(p)
```

Attribute Specification Template

The LRM (§5.1) defines the grammar for attribute specification as follows:

```
attribute_specification ::= attribute attribute_designator of
       entity_specification is expression ;
entity_specification ::= entity_name_list : entity_class
entity_class ::=
        entity     | architecture   | configuration
       | procedure | function       | package
       | type      | subtype        | constant
       | signal    | variable       | component
       | label     | literal        | units
       | group     | file
entity_name_list ::=
        entity_designator { , entity_designator }
       | others
       | all
entity_designator ::=  entity_tag [ signature ]
entity_tag ::= simple_name | character_literal | operator_symbol
```

The attribute_specification template is a primary template belonging to no classes. It contains the attributes shown in **Table 24**.

Table 24: attribute_specification Template

Attribute	Kind	Limit_Kind
cased_label	template	ID
label	template	ID
attribute_designator	template	attribute_declaration
entity_designator	template	NAME
expression	template	EXPRESSION
others	local	N/A
all	local	N/A
entity	local	N/A
architecture	local	N/A
configuration	local	N/A
procedure	local	N/A
function	local	N/A
package	local	N/A

Table 24: attribute_specification Template (Continued)

Attribute	Kind	Limit_Kind
type	local	N/A
subtype	local	N/A
constant	local	N/A
signal	local	N/A
variable	local	N/A
component	local	N/A
literal	local	N/A
units	local	N/A
group	local	N/A
file	local	N/A

Use the cased_label just like the label attribute, but in cases where the label is case-sensitive. Consider the following templates:

```
template T1 is attribute_specification
       limit label to "^[a-z][a-z0-9_]*$"
end
template T2 is attribute_specification
       limit cased_label to "^[a-z][a-z0-9_]*$"
end
```

The lab1 label matches both templates, whereas the LAB1 label only matches the first template. The label attribute is not case-sensitive because VHDL is not case-sensitive.

Binary Operation Template

The binary_operation template is a primary template belonging to the EXPRESSION class. It contains the attributes shown in Table 25.

Table 25: binary_operation Template

Attribute	Kind	Limit_Kind
left_expression	template	EXPRESSION
right_expression	template	EXPRESSION
type_mark	template	TYPE_MARK
operand_size_mismatch	local	N/A
operator_symbol	template	ID
value	template	literal
evaluation_time	set	N/A
operand_size_match	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 these enumerated literals:

- undefined
- locally_static_evaluation
- globally_static_evaluation
- dynamic_evaluation

For example:

```
set evaluation_time to locally_static_evaluation
```

In addition to literal templates, you can use the value attribute for either integer values or character strings representing the literal value. For example:

```
limit value to 0
limit value to "0"
```

Binding Indication Template

The LRM (§5.2.1) defines the grammar for binding indication as follows:

```
binding_indication ::=
        [ use entity_aspect ]
        [ generic_map_aspect ]
        [ port_map_aspect ]
entity_aspect ::=
          entity entity_name [ (architecture_identifier) ]
        | configuration configuration_name
        | open
generic_map_aspect ::=
          generic map (generic_association_list)
port_map_aspect ::=
          port map (port_association_list)
```

The binding_indication template is a secondary template belonging to no classes. It contains the attributes shown in **Table 26**.

Table 26: binding_indication Template

Attribute	Kind	Limit_Kind
generic_map_aspect	template	association_list
port_map_aspect	template	association_list
entity_aspect_name	template	ID
entity	local	N/A
configuration	local	N/A

Use the entity_aspect_name attribute to configure the name of the entity you want to constrain. For example, to prohibit the use of “WORK” in the entity name, you could write the following rule:

```
template NO_WORK_IN_ENTITY_NAME is binding_indication
       limit entity_aspect_name to "^[^w][^o][^r][^k]"
end
template NO_WORK_IN_CONFIG_SPEC is configuration_specification
       limit binding_indication to NO_WORK_IN_CONFIG_SPEC
end
limit configuration_specification in all to NO_WORK_IN_CONFIG_SPEC
```

Use the entity and configuration attributes to force or prohibit entity or configuration entity aspects.

Block Configuration Template

The LRM (§1.3.1) defines the grammar for block configuration as follows:

```
block_configuration ::=
       for block_specification
             { use_clause }
             { configuration_item }
       end for ;
configuration_item ::=
             block_configuration
             | component_configuration
```

The block_configuration template is a primary template belonging to the REGION_PART class. It contains the attributes shown in **Table 27**.

Table 27: block_configuration Template

Attribute	Kind	Limit_Kind
block_specification	template	block_specification
use_clause	template	use_clause
block_configuration	template	block_configuration
component_configuration	template	component_configuration

Block Specification Template

The LRM (§1.3.1) defines the grammar for block specification as follows:

```
block_specification ::=
          architecture_name
        | block_statement_label
        | generate_statement_label [ ( index_specification ) ]

index_specification ::=
          discrete_range
        | static_expression
```

The block_specification template is a primary template belonging to the REGION_PART class. It contains the attributes shown in Table 28.

Table 28: block_specification Template

Attribute	Kind	Limit_Kind
index_specification	template	INDEX_SPEC
architecture_name	local	N/A
block_statement_label	local	N/A
generate_statement_label	local	N/A

To limit the index_specification attribute, use a template that belongs to the INDEX_SPEC class. For example:

```
template RANGE_1 is range
       no null_range
       no downto
       set evaluation_time to locally_static_evaluation
end
limit index_specification in block_specification to RANGE_1
       severity error
```

Block Statement Template

The LRM (§9.1) defines the grammar for block statement as follows:

```
block_statement ::=
       block_label :
             block [ ( guard_expression ) ] [ is ]
                   block_header
                   block_declarative_part
             begin
                   block_statement_part
             end block [ block_label ] ;
block_header ::=
       [ generic_clause  [ generic_map_aspect ; ] ]
       [ port_clause  [ port_map_aspect ; ] ]
block_declarative_part ::=
       { block_declarative_item }
block_statement_part ::=
       { concurrent_statement}
```

The block_statement template is a primary template belonging to the SEQUENTIAL_STATEMENT class. It contains the attributes shown in **Table 29**.

Table 29: block_statement Template

Attribute	Kind	Limit_Kind
cased_label	template	ID
label	template	ID
guard_expression	template	EXPRESSION
generic_declaration	template	interface_constant_declaration
generic_map_aspect	template	association_list
port_declaration	template	interface_signal_declaration
port_map_aspect	template	association_list
subprogram_declaration	template	subprogram_declaration
subprogram_body	template	subprogram_body
type_declaration	template	type_declaration
subtype_declaration	template	subtype_declaration
constant_declaration	template	constant_declaration
shared_variable_declaration	template	shared_variable_declaration

Table 29: block_statement Template (Continued)

Attribute	Kind	Limit_Kind
signal_declaration	template	signal_declaration
file_declaration	template	file_declaration
alias_declaration	template	alias_declaration
attribute_declaration	template	attribute_declaration
attribute_specification	template	attribute_specification
disconnection_specification	template	disconnection_specification
component_declaration	template	component_declaration
configuration_specification	template	configuration_specification
use_clause	template	use_clause
group_template_declaration	template	group_template_declaration
group_declaration	template	group_declaration
assertion_statement	template	assertion_statement
procedure_call_statement	template	procedure_call_statement
process_statement	template	process_statement
block_statement	template	block_statement
selected_signal_assignment	template	selected_signal_assignment
conditional_signal_assignment	template	conditional_signal_assignment
component_instantiation_statement	template	component_instantiation_statement
generate_statement	template	generate_statement
declarative_region	template	REGION_PART
statement_format	template	statement_format
statement_profile_s	aggregate	CONCURRENT_STATEMENT
declaration_profile_s	aggregate	DECLARATIVE_ITEM

Use the cased_label attribute just like the label attribute, but in cases where the label is case-sensitive. Consider the following templates:

```
template T1 is block_statement
       limit label to "^[a-z][a-z0-9_]*$"
end
template T2 is block_statement
       limit cased_label to "^[a-z][a-z0-9_]*$"
end
```

The lab1 label matches both templates, whereas the LAB1 label only matches the first template. The label attribute is not case-sensitive because VHDL is not case-sensitive.

Case Statement Template

The LRM (§8.8) defines the grammar for case statement as follows:

```
case_statement ::=
        [ case_label : ]
              case expression is
                    case_statement_alternative
                    {case_statement_alternative}
              end case[ case_label ] ;
case_statement_alternative ::= when choices => { sequence_of_statements }
```

The case_statement template is a primary template belonging to the SEQUENTIAL_STATEMENT class. It contains the attributes shown in Table 30.

Table 30: case_statement Template

Attribute	Kind	Limit_Kind
cased_label	template	ID
label	template	ID
expression	template	EXPRESSION
choice	template	EXPRESSION
assertion_statement	template	assertion_statement
report_statement	template	report_statement
procedure_call_statement	template	procedure_call_statement
if_statement	template	if_statement
signal_assignment_statement	template	signal_assignment_statement
variable_assignment_statement	template	variable_assignment_statement

Table 30: case_statement Template (Continued)

Attribute	Kind	Limit_Kind
case_statement	template	case_statement
loop_statement	template	loop_statement
for_loop_statement	template	for_loop_statement
while_loop_statement	template	while_loop_statement
next_statement	template	next_statement
exit_statement	template	exit_statement
return_statement	template	return_statement
wait_statement	template	wait_statement
null_statement	local	N/A
others	local	N/A
declarative_region	template	REGION_PART
statement_format	template	statement_format
multiple_choices	local	N/A
alternative_count	max/min	N/A
statement_profile_s	aggregate	SEQUENTIAL_STATEMENT

Use the multiple_choices attribute to control the presence of more than one choice for the same alternative.

Use the others attribute to control the presence of the others alternative. Note that, if all alternatives are not explicit and there is no others clause, there is a VHDL syntax error.

Use the cased_label attribute just like the label attribute, but in cases where the label is case-sensitive. Consider the following templates:

```
template T1 is case_statement
       limit label to "^[a-z][a-z0-9_]*$"
end
template T2 is case_statement
       limit cased_label to "^[a-z][a-z0-9_]*$"
end
```

The lab1 label matches both templates, whereas the LAB1 label only matches the first template. The label attribute is not case-sensitive because VHDL is not case-sensitive.

Clock Template

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

Table 31: clock Template

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

The clock template recognizes all RTL clock expressions. If you want to constrain the expression type for clocks, use the expression attribute. For example, you could use the expression attribute if the following code was not acceptable:

```
clk='1' and not clk'STABLE
```

Use the object_definition attribute to limit the type of the clock signal.

Use the fixed_value attribute to detect clocks with fixed values.

Use the edge attribute to control the active edge of the clock. You can set the edge attribute to any of the following values:

- Rising_Edge
- Falling_Edge
- Both_Edges
- High_Level
- Low_Level

For example, the following VRSL code sets the active clock edge to rising:

```
template MY_CLOCK is clock
       set edge to rising_edge
end
```

Use the data attribute to constrain the use of the clock signal as data.

Use the gated_in_unit attribute to control the whereabouts of gated clocks with greater precision. Using this attribute, you can write post-elaboration rules that control gated clocks.

Use the cased_identifier attribute just like the identifier attribute, but in cases where the identifier is case-sensitive. Consider the following templates:

```
template T1 is clock
       limit identifier to "^[a-z][a-z0-9_]*$"
end
template T2 is clock
       limit cased_identifier to "^[a-z][a-z0-9_]*$"
end
```

The id1 identifier matches both templates, whereas the ID1 identifier only matches the first template. The attribute identifier is not case-sensitive because VHDL is not case-sensitive.

Note

If you use a complex expression (containing more than one object) as a clock or reset, Leda automatically deactivates the naming convention rules. This is because Leda assumes that the clock or reset is the output of a logic gate inferred by the complex expression and not simply one of the objects in the expression.

Comment Template

The comment template is a primary template belonging to no classes. It contains the attributes shown in **Table 32.**

Table 32: comment Template

Attribute	Kind	Limit_Kind
enclosing_filename	template	ID
text	template	ID

Component Configuration Template

The LRM (§1.3.1) defines the grammar for component configuration as follows:

```
component_configuration ::=
        for component_specification
              [ binding_indication ; ]
              [ block_configuration ]
        end for;
```

The component configuration template is a primary template belonging to the REGION_PART class. It contains the attributes shown in **Table 33**.

Table 33: component_configuration Template

Attribute	Kind	Limit_Kind
component_specification	template	component_specification
binding_indication	template	binding_indication
block_configuration	template	block_configuration

Component Declaration Template

The LRM (§4.5) defines the grammar for component declaration as follows:

```
component_declaration ::=
        component identifier [ is ]
              [ local_generic_clause]
              [ local_port_clause]
        end component [ component_simple_name ]
```

The component_declaration template is a primary template belonging to the OBJECT_ITEM class. It contains the attributes shown in **Table 34**.

Table 34: component_declaration Template

Attribute	Kind	Limit_Kind
cased_identifier	template	ID
identifier	template	ID
generic_declaration	template	interface_constant_declaration
port_declaration	template	interface_signal_declaration

Table 34: component_declaration Template (Continued)

Attribute	Kind	Limit_Kind
declarative_region	template	REGION_PART
port_order	local	N/A
port_count	max/min	N/A

Use the port_order attribute to define the order in which ports must appear. Specify the order according to the mode of the ports. You must specify the order using the following strings (otherwise Leda flags an error):

- in
- out
- inout
- buffer
- linkage

You can separate these strings using white space or commas, as shown in the following VRSL example, which uses white space to separate the specified “out inout in” port order:

```
limit port_order in entity_declaration to "out inout in"
message "ERROR: Ports are not in correct order"
severity ERROR
```

Using this example, if the specified port order is not followed or there is a port of mode buffer or linkage, the Leda Checker flags an error, and generates the specified message.

Use the cased_identifier attribute just like the identifier attribute, but in cases where the identifier is case-sensitive. Consider the following templates:

```
template T1 is component_declaration
       limit identifier to "^[a-z][a-z0-9_]*$"
end
template T2 is component_declaration
       limit cased_identifier to "^[a-z][a-z0-9_]*$"
end
```

The id1 identifier matches both templates, whereas the ID1 identifier only matches the first template. The identifier attribute is not case-sensitive because VHDL is not case-sensitive.

Component Instantiation Statement Template

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

```
component_instantiation_statement ::=
       instantiation_label :
             instantiated_unit
                   [ generic_map_aspect ]
                   [ port_map_aspect ] ;
instantiated_unit ::=
       [component ] component_name
       | entity entity_name [ ( architecture_identifier ) ]
       | configuration configuration_name
```

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

Table 35: component_instantiation_statement Template

Attribute	Kind	Limit_Kind
cased_label	template	ID
label	template	ID
unit_name	template	ID
generic_map_aspect	template	association_list
port_map_aspect	template	association_list
declarative_region	template	REGION_PART
entity	local	N/A
configuration	local	N/A
consistent_port_order	local	N/A
use_db_name	local	N/A
db_instantiation	local	N/A

- Use the unit_name attribute to configure the name of the instantiated component. For example, to ensure that the instantiated component's name corresponds to an existing entity, you could write the following rule:

```
limit unit_name in component_instantiation_statement to "^<entity>$"
```

- Use the entity and configuration attributes to force or forbid the presence of the entity and configuration VHDL keywords.

- Use the use_db_name attribute to ensure that the label name of the module instantiation does not collide with the name of a DB cell specified in the $link_library and found in a directory in $search_path.
- 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 component_instantiation_statement

architecture A of E is

begin
    db_cell : comp( );        // Fail -- use_db_name rule
    lab: db_cell( );          // Fail -- no no db_instantiation rule

end A
```

- Use the cased_label attribute just like the label attribute, but in cases where the label is case-sensitive. Consider the following templates:

```
template T1 is component_instantiation_statement
       limit label to "^[a-z][a-z0-9_]*$"
end
       template T2 is component_instantiation_statement
       limit cased_label to "^[a-z][a-z0-9_]*$"
end
```

The lab1 label matches both templates, whereas the LAB1 label only matches the first template. The label attribute is not case-sensitive because VHDL is not case-sensitive.

Component Specification Template

The LRM (§5.2) defines the grammar for component specification as follows:

```
component_specification ::= instantiation_list : component_name
```

The component_specification template is a primary template belonging to no classes. It contains the attributes shown in **Table 36**.

Table 36: component_specification Template

Attribute	Kind	Limit_Kind
cased_label	template	ID
label	template	ID

Table 36: component_specification Template

Attribute	Kind	Limit_Kind
others	local	N/A
all	local	N/A
postponed	local	N/A

Use the cased_label attribute just like the label attribute, but in cases where the label is case-sensitive. Consider the following templates:

```
template T1 is component_specification
       limit label to "^[a-z][a-z0-9_]*$"
end
template T2 is component_specification
       limit cased_label to "^[a-z][a-z0-9_]*$"
end
```

The lab1 label matches both templates, whereas the LAB1 label only matches the first template. The label attribute is not case-sensitive because VHDL is not case-sensitive.

Use the others and all attributes to force or forbid the presence of the others and all VHDL keywords in an instantiation list.

Concurrent Assertion Statement Template

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

```
concurrent_assertion_statement ::= [ label : ]  [ postponed ] assertion ;
```

The concurrent_assertion_statement template is a primary template belonging to the CONCURRENT_STATEMENT class. It contains the attributes shown in **Table 37**.

Table 37: concurrent_assertion_statement Template

Attribute	Kind	Limit_Kind
cased_label	template	ID
label	template	ID
condition	template	EXPRESSION
report_expression	template	EXPRESSION

Table 37: concurrent_assertion_statement Template (Continued)

Attribute	Kind	Limit_Kind
severity_expression	template	EXPRESSION
declarative_region	template	REGION_PART
postponed	local	N/A

Use the cased_label attribute just like the label attribute, but in cases where the label is case-sensitive. Consider the following templates:

```
template T1 is concurrent_assertion_statement
       limit label to "^[a-z][a-z0-9_]*$"
end
template T2 is concurrent_assertion_statement
       limit cased_label to "^[a-z][a-z0-9_]*$"
end
```

The lab1 label matches both templates, whereas the LAB1 label only matches the first template. The label attribute is not case-sensitive because VHDL is not case-sensitive.

Concurrent Procedure Call Statement Template

The LRM (§9.3) defines the grammar for concurrent procedure call as follows:

```
concurrent_procedure_call_statement ::= [ label : ]  [ postponed ]
  procedure_call;
```

The concurrent_procedure_call_statement template is a primary template belonging to the CONCURRENT_STATEMENT class. It contains the attributes shown in **Table 38**.

Table 38: concurrent_procedure_call_statement Template

Attribute	Kind	Limit_Kind
cased_identifier	template	ID
cased_label	template	ID
label	template	ID
identifier	template	ID
actual_parameter_part	template	association_list
declarative_region	template	REGION_PART
postponed	local	N/A

Use the cased_identifier attribute just like the identifier attribute, but in cases where the identifier is case-sensitive. Consider the following templates:

```
template T1 is clock
       limit identifier to "^[a-z][a-z0-9_]*$"
end
template T2 is clock
       limit cased_identifier to "^[a-z][a-z0-9_]*$"
end
```

The id1 identifier matches both templates, whereas the ID1 identifier only matches the first template. The identifier attribute is not case-sensitive because VHDL is not case-sensitive.

Similarly, you use the cased_label attribute just like the label attribute, but in cases where the label is case-sensitive. Consider the following templates:

```
template T1 is concurrent_procedure_call_statement
       limit label to "^[a-z][a-z0-9_]*$"
end
template T2 is concurrent_procedure_call_statement
       limit cased_label to "^[a-z][a-z0-9_]*$"
end
```

The lab1 label matches both templates, whereas the LAB1 label only matches the first template. The label attribute is not case-sensitive because VHDL is not case-sensitive.

Conditional Signal Assignment Template

The LRM (§9.5.1) defines the grammar for conditional signal assignment as follows:

```
conditional_signal_assignment ::=
   target  <=  options conditional_waveforms ;
```

The conditional_signal_assignment template is a primary template belonging to the CONCURRENT_STATEMENT class. It contains the attributes shown in **Table 39**.

Table 39: conditional_signal_assignment Template

Attribute	Kind	Limit_Kind
cased_label	template	ID
label	template	ID
target	template	TARGET
reject_expression	template	EXPRESSION
operand_size_mismatch	local	N/A
postponed	local	N/A

Table 39: conditional_signal_assignment Template (Continued)

Attribute	Kind	Limit_Kind
range_overflow	local	N/A
statement_format	template	statement_format
transport	local	N/A
inertial	local	N/A
guarded	local	N/A
read_write	local	N/A
waveform_count	max/min	N/A
conditional_waveforms_s	aggregate	conditional_waveforms

Use the transport, inertial, and guarded attributes to force or forbid the presence of the corresponding VHDL keywords (transport, inertial, and guarded).

Use the waveform_count attribute to set the maximum number of waveforms allowed.

Use the cased_label attribute just like the label attribute, but in cases where the label is case-sensitive. Consider the following templates:

```
template T1 is conditional_signal_assignment
       limit label to "^[a-z][a-z0-9_]*$"
end
template T2 is conditional_signal_assignment
       limit cased_label to "^[a-z][a-z0-9_]*$"
end
```

The lab1 label matches both templates, whereas the LAB1 label only matches the first template. The label attribute is not case-sensitive because VHDL is not case-sensitive.

Conditional Waveforms Template

The LRM (§9.5.1) defines the grammar for conditional waveforms as follows:

```
conditional_waveforms ::=
       { waveform when condition else }
        waveform [ when condition ]
```

This template is used in the conditional_signal_assignment template.

The conditional_waveforms template is a secondary template belonging to no classes. It contains the attributes shown in **Table 40**.

Table 40: conditional_waveforms Template

Attribute	Kind	Limit_Kind
waveform	template	waveform
condition	template	EXPRESSION

Configuration Declaration Template

The LRM (§2.3) defines the grammar for configuration declaration as follows:

```
configuration_declaration ::=
       configuration identifier of entity_name is
             configuration_declarative_part
             block_configuration
       end [ configuration ][ configuration_simple_name ] ;
configuration_declarative_part::= { configuration_declarative_item}
configuration_declarative_item::=
          use_clause | attribute_specification | group_declaration
```

The configuration_declaration template is a primary template belonging to the REGION_PART and OBJECT_ITEM classes. It contains the attributes shown in **Table 41**.

Table 41: configuration_declaration Template

Attribute	Kind	Limit_Kind
cased_identifier	template	ID
identifier	template	ID
use_clause	template	use_clause
attribute_specification	template	attribute_specification
group_declaration	template	group_declaration
block_configuration	template	block_configuration
design_unit	template	design_unit
header_comment	template	header_comment
top_configuration	template	configuration_declaration
file_name	template	ID
ncs_file_name	template	ID
file_layout	template	file_layout
line_count	max/min	N/A
file_length_count	max/min	N/A
related_unit	local	N/A

Employ the use-clause attribute to constrain use-clauses declared inside a unit. To constrain use clauses (and library clauses) appearing before a unit, employ the design_unit template instead.

This template includes the following application-specific attributes that make it easier to write commonly-used rules:

- Use the top_configuration attribute to constrain the top-level configuration.

  To make this work, you must specify the name of the top-level architecture using the –top option of the command-line Checker.

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

```
limit file_name in configuration_declaration to "config.vhd"
```

- Use the line_count attribute to specify the maximum or minimum lines of code allowed in a unit.

Use the cased_identifier attribute just like the identifier attribute, but in cases where the identifier is case-sensitive. Consider the following templates:

```
template T1 is configuration_declaration
       limit identifier to "^[a-z][a-z0-9_]*$"
end
template T2 is configuration_declaration
       limit cased_identifier to "^[a-z][a-z0-9_]*$"
end
```

The id1 identifier matches both templates, whereas the ID1 identifier only matches the first template. The identifier attribute is not case-sensitive because VHDL is not case-sensitive.

- Use the related_unit attribute to check that the architecture or the configuration you can find in a file is related to the entity described in that same file.

Configuration Specification Template

The LRM (§5.2) defines the grammar for configuration specification as follows:

```
configuration_specification ::=
       for component_specification binding_indication ;
component_specification ::= instantiation_list : component_name
instantiation_list ::=
        instantiation_label { , instantiation_label }
       | others
       | all
```

The configuration_specification template is a primary template belonging to no classes. It contains the attributes shown in **Table 42**.

Table 42: configuration_specification Template

Attribute	Kind	Limit_Kind
binding_indication	template	binding_indication
others	local	N/A
all	local	N/A
consistent_port_order	local	N/A

Use the others and all attributes to force or forbid the presence of the corresponding VHDL keywords (others and all).

Constant Declaration Template

The LRM (§4.3.1.1) defines the grammar for constant declaration as follows:

```
constant_declaration ::=
       constant identifier_list : subtype_indication [ := expression ] ;
```

The constant_declaration template is a primary template belonging to the OBJECT_ITEM class. It contains the attributes shown in **Table 43.**

Table 43: constant_declaration Template

Attribute	Kind	Limit_Kind
cased_identifier	template	ID
identifier	template	ID
subtype_indication	template	subtype_indication
default	template	EXPRESSION

Table 43: constant_declaration Template (Continued)

Attribute	Kind	Limit_Kind
declarative_region	template	REGION_PART
deferred	local	N/A
one_declaration_per_line	local	N/A
comment	local	N/A

Use the cased_identifier attribute just like the identifier attribute, but in cases where the identifier is case-sensitive. Consider the following templates:

```
template T1 is constant _declaration
       limit identifier to "^[a-z][a-z0-9_]*$"
end
template T2 is constant _declaration
       limit cased_identifier to "^[a-z][a-z0-9_]*$"
end
```

The id1 identifier matches both templates, whereas the ID1 identifier only matches the first template. The identifier attribute is not case-sensitive because VHDL is not case-sensitive.

Use the one_declaration_per_line attribute to force a single line for each declaration.

Use the comment attribute to force a comment on the same line as the declaration.

Constrained Array Definition Template

The LRM (§3.2.1) defines the grammar for constrained array definition as follows:

```
constrained_array_definition ::=
       array index_constraint of element_subtype_indication
index_constraint ::=  ( discrete_range { , discrete_range } )
discrete_range ::=  discrete_subtype_indication | range
```

The constrained_array_definition template is a primary template belonging to no classes. It contains the attributes shown in **Table 44**.

Table 44: constrained_array_definition Template

Attribute	Kind	Limit_Kind
subtype_indication	template	subtype_indication
dimension_count	max/min	N/A
index_constraint_s	aggregate	DISCRETE_RANGE

Use the dimension_count attribute to specify the maximum or minimum number of allowed dimensions. For example, the command:

```
max dimension_count in constrained_array_definition is 1
```

means that only 1-dimensional arrays are allowed.

To allow more than one dimension, set different constraints on the index expressions for each dimension. Make sure each aggregate points to a set of range templates.

Design Unit Template

The LRM (§11.1) defines the grammar for design unit as follows:

```
design_unit ::= context_clause library_unit
context_clause ::= {context_item}
context_item ::=   library_clause
                   | use_clause
```

The design_unit template is a secondary template belonging to no classes. It contains the attributes shown in **Table 45**.

Table 45: design_unit Template

Attribute	Kind	Limit_Kind
library_clause	template	ID
library_name	template	ID
use_clause	template	use_clause
use_work	local	N/A

Use the library_clause attribute to constrain the libraries imported into the design.

Use the library_name attribute to constrain the name of the library containing the unit.

Disconnection Specification Template

The LRM (§5.3) defines the grammar for disconnection specifications as follows:

```
disconnection_specification ::=
       disconnect guarded_signal_specification after time_expression
guarded_signal_specification ::= guarded_signal_list : type_mark
signal_list ::=
        signal_name { ,signal_name }
       | others
       | all
```

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

Table 46: disconnection_specification Template

Attribute	Kind	Limit_Kind
type_mark	template	TYPE_MARK
others	local	N/A
all	local	N/A

Element Association Template

The LRM (§7.3.2) defines the grammar for element association as follows:

```
element_association ::= [ choices => ]  expression
choices ::= choice {  | choice }
choice ::=
        simple_expression
       | discrete_range
       | element_simple_name
       | others
```

The element_association template is a secondary template belonging to no classes. It contains the attributes shown in **Table 47**.

Table 47: element_association Template

Attribute	Kind	Limit_Kind
choice	template	EXPRESSION
positional_association	local	N/A
range_overflow	local	N/A

Table 47: element_association Template (Continued)

Attribute	Kind	Limit_Kind
named_association	local	N/A
multiple_choices	local	N/A
others	local	N/A

Element Declaration Template

The LRM (§3.2.2) defines the grammar for element declaration as follows:

```
element_declaration ::= identifier_list : element_subtype_definition ;
```

The element_declaration template is a secondary template belonging to no classes. It contains the attributes shown in **Table 48**.

Table 48: element_declaration Template

Attribute	Kind	Limit_Kind
cased_identifier	template	ID
identifier	template	ID
subtype_indication	template	subtype_indication

Use the cased_identifier attribute just like the identifier attribute, but in cases where the identifier is case-sensitive. Consider the following templates:

```
template T1 is element_declaration
       limit identifier to "^[a-z][a-z0-9_]*$"
end
template T2 is element_declaration
       limit cased_identifier to "^[a-z][a-z0-9_]*$"
end
```

The id1 identifier matches both templates, whereas the ID1 identifier only matches the first template. The identifier attribute is not case-sensitive because VHDL is not case-sensitive.

Entity Declaration Template

The LRM (§1.1.1) defines the grammar for entity declaration as follows:

```
entity_declaration ::=
       entity identifier is
             entity_header
             entity_declarative_part
       [begin
             entity_statement_part ]
end [ entity ] [ entity_simple_name ] ;
```

The LRM (§1.1.1) defines the grammar for entity header as follows:

```
entity_header ::=
       [ formal_generic_clause ]
       [ formal_port_clause ]
generic_clause ::=
       generic ( generic_interface_list ) ;
port_clause ::=
       port ( port_interface_list ) ;
```

The LRM (§1.1.2) defines the grammar for entity declarative part as follows:

```
entity_declarative_part ::=
       { entity_declarative_item }
entity_declarative_item ::=
          subprogram_declaration
       | subprogram_body
       | type_declaration
       | subtype_declaration
       | constant_declaration
       | signal_declaration
       | shared_variable_declaration
       | file_declaration
       | alias_declaration
       | attribute_declaration
       | attribute_specification
       | disconnection_specification
       | use_clause
       | group_template_declaration
       | group declaration
```

The LRM (§1.1.3) defines the grammar for entity statement part as follows:

```
entity_statement_part ::=
       { entity_statement }
entity_statement ::=
          concurrent_assertion_statement
       | passive_concurrent_procedure_call
       | passive_process_stm
```

The entity_declaration template is a primary template belonging to the REGION_PART and OBJECT_ITEM classes. It contains the attributes shown in **Table 49**.

Table 49: entity_declaration Template

Attribute	Kind	Limit_Kind
cased_identifier	template	ID
identifier	template	ID
generic_declaration	template	interface_constant_declaration
port_declaration	template	interface_signal_declaration
subprogram_declaration	template	subprogram_declaration
subprogram_body	template	subprogram_body
type_declaration	template	type_declaration
subtype_declaration	template	subtype_declaration
constant_declaration	template	constant_declaration
signal_declaration	template	signal_declaration
shared_variable_declaration	template	shared_variable_declaration
use_db_name	local	N/A
file_declaration	template	file_declaration
alias_declaration	template	alias_declaration
attribute_declaration	template	attribute_declaration
attribute_specification	template	attribute_specification
disconnection_specification	template	disconnection_specification
use_clause	template	use_clause
group_template_declaration	template	group_template_declaration
group_declaration	template	group_declaration
assertion_statement	template	assertion_statement
procedure_call_statement	template	procedure_call_statement
process_statement	template	process_statement
design_unit	template	design_unit

Table 49: entity_declaration Template (Continued)

Attribute	Kind	Limit_Kind
header_comment	template	header_comment
top_entity	template	entity_declaration
file_name	template	ID
ncs_file_name	template	ID
file_layout	template	file_layout
port_order	local	N/A
port_count	max/min	N/A
line_count	max/min	N/A
declaration_profile_s	aggregate	DECLARATIVE_ITEM
file_length	max/min	N/A

Employ the use_clause attribute to constrain the number of use-clauses declared inside the entity declaration (after the entity keyword). To constrain the number of use clauses (and library clauses) appearing before the entity keyword, use the design_unit template instead.

There are some application-specific attributes associated with this template that make it easier to write commonly-used rules. For example:

- Use the top_entity attribute to constrain the top-level entity.

  To make this work, you must specify the name of the top-level entity using the –top switch with the command-line Checker.

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

```
limit file_name in entity_declaration to "<entity>.vhd"
```

- Use the file_layout attribute to control the organization of the containing file.
- Use the port_order attribute to specify the order in which ports must appear. Specify the order according to the mode of the ports using any of these strings: “in”, “out”, “inout”, “buffer”, and “linkage”. Any other strings cause Leda to flag an error. Separate the port strings using blanks or commas. For example:

```
limit port_order in entity_declaration to "out inout in"
   message "ERROR: Ports are not in correct order" severity ERROR
```

In this example, if the order is not followed or a port of mode buffer or linkage appears, Leda flags an error and prints the specified message.

- Use the use_db_name attribute to ensure that the name of the module does not collide with the name of a DB cell specified in the $link_library and found in a directory in $search_path.
- Use the line_count attribute to specify the maximum or minimum lines of code allowed in a unit.
- Use the declaration_profile_s attribute to specify the order in which declarations must appear. For example:

```
limit declaration_profile_s in entity_declaration to
    first       type_declaration
    then        subtype_declaration
    then        {}
```

Use the cased_identifier attribute just like the identifier attribute, but in cases where the identifier is case-sensitive. Consider the following templates:

```
template T1 is entity_declaration
       limit identifier to "^[a-z][a-z0-9_]*$"
end
template T2 is entity_declaration
       limit cased_identifier to "^[a-z][a-z0-9_]*$"
end
```

The id1 identifier matches both templates, whereas the ID1 identifier only matches the first template. The identifier attribute is not case-sensitive because VHDL is not case-sensitive.

Enumeration Type Definition Template

The LRM (§3.1.1) defines the grammar for enumeration type definition as follows:

```
enumeration_type_definition ::=
       ( enumeration_literal { , enumeration_literal } )
enumeration_literal ::=  identifier | character_literal
```

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

Table 50: enumeration_type Template

Attribute	Kind	Limit_Kind
cased_identifier	template	ID
identifier	template	ID
character_literal	local	N/A
element_count	min_max	N/A

For the command:

```
no character_literal in enumeration_type_definition
severity error
```

Leda flags an error for the following:

```
type BAD_ENUM is (NO,CHARACTER,LITERALS,ALLOWED,'C');
```

but not for this:

```
type GOOD_ENUM is (NO,CHARACTER,LITERALS,ALLOWED);
```

Use the cased_identifier attribute just like the identifier attribute, but in cases where the identifier is case-sensitive. Consider the following templates:

```
template T1 is enumeration_type_definition
       limit identifier to "^[a-z][a-z0-9_]*$"
end
template T2 is enumeration_type_definition
       limit cased_identifier to "^[a-z][a-z0-9_]*$"
end
```

The id1 identifier matches both templates, whereas the ID1 identifier only matches the first template. The identifier attribute is not case-sensitive because VHDL is not case-sensitive.

Exit Statement Template

The LRM (§8.11) defines the grammar for exit statement as follows:

```
exit_statement ::= [ label : ] exit [ loop_label ] [ when condition ] ;
```

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

Table 51: exit_statement Template

Attribute	Kind	Limit_Kind
cased_label	template	ID
label	template	ID
loop_label	template	ID
condition	template	EXPRESSION
declarative_region	template	REGION_PART

Use the cased_label attribute just like the label attribute, but in cases where the label is case-sensitive. Consider the following templates:

```
template T1 is exit_statement
       limit label to "^[a-z][a-z0-9_]*$"
end
template T2 is exit_statement
       limit cased_label to "^[a-z][a-z0-9_]*$"
end
```

The lab1 label matches both templates, whereas the LAB1 label only matches the first template. The label attribute is not case-sensitive because VHDL is not case-sensitive.

Expression Template

The VRSL representation of expressions deviates from the LRM representation because the LRM is so detailed that it leads to long chains of design rules to express even simple expressions.

The expression template is a primary template belonging to the EXPRESSION class. It contains the attributes shown in **Table 52**.

Table 52: expression Template

Attribute	Kind	Limit_Kind
operator_symbol	template	ID
value	template	literal
operand	template	NAME
qualified	local	N/A
type_conversion	local	N/A
evaluation_time	set	N/A

Use the operator_symbol attribute to specify operators that are acceptable in expressions. You can specify character strings or templates of class ID that represent the acceptable operators. For example:

```
limit operator_symbol in expression to "+"
```

Use the operand attribute to write rules that constrain all operands in an expression.

Use the evaluation_time attribute to specify any of the following enumerated literals:

- undefined
- locally_static_evaluation
- globally_static_evaluation
- dynamic_evaluation

For example:

```
set evaluation_time to locally_static_evaluation
```

Use the value attribute to specify an integer or character string that represents the literal value. For example:

```
limit value to 0
limit value to "0"
```

Use the type_conversion attribute to control the types of conversion functions allowed, if any.

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 use case statements to define the states and transitions. Leda cannot identify FSMs defined using if statements. Leda recognizes 1-process, 2-process, and 3-process FSM templates 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 allowed in an FSM. Note that Leda only counts states that are reachable. Therefore, the number of states is not equal to the number of different values that the state variable can have. You can also use the state_count 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 state variables (for example, the name and type).

Use the block_count attribute to constrain the number of processes used to infer an FSM (for example. 1-, 2,- or 3-block FSMs).

Use the moore attribute to specify that only Moore-style FSMs are allowed, as shown in the following example:

```
force moore in fsm
```

Use the mealy attribute to specify that only Mealy-style FSMs are allowed, as shown in the following example:

```
force mealy in fsm
```

Use the transition_in_default attribute to force 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 general coding guidelines policy. For more information, see the *Leda General Coding Rules Guide*.

File Declaration Template

The LRM (§4.3.1.4) defines the grammar for file declaration as follows:

```
file_declaration ::= file identifier_list : subtype_indication
   [ file_open_information ] ;
file_open_information ::= [ open file_open_kind_expression ] is
   file_logical_name
file_logical_name ::= string_expression
```

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

Table 54: file_declaration Template

Attribute	Kind	Limit_Kind
cased_identifier	template	ID
identifier	template	ID
subtype_indication	template	subtype_indication
logical_name	template	simple_name
open_kind	template	simple_name
declarative_region	template	REGION_PART
one_declaration_per_line	local	N/A
comment	local	N/A

Use the cased_identifier attribute just like the identifier attribute, but in cases where the identifier is case-sensitive. Consider the following templates:

```
template T1 is file_declaration
       limit identifier to "^[a-z][a-z0-9_]*$"
end
template T2 is file_declaration
       limit cased_identifier to "^[a-z][a-z0-9_]*$"
end
```

The id1 identifier matches both templates, whereas the ID1 identifier only matches the first template. The identifier attribute is not case-sensitive because VHDL is not case-sensitive.

Use the one_declaration_per_line attribute to force a single line for each declaration.

Use the comment attribute to force a comment on the same line as the declaration.

File Layout Template

Use the file_layout template to control the layout of the file containing your VHDL code. Note that you can also set some of these same things using the VHDL Beautifier that is part of the Checker tool. For more information on using the VHDL Beautifier, see the *Leda User Guide*.

The file_layout template is a primary template belonging to the no classes. It contains the attributes shown in **Table 55**.

Table 55: file_layout Template

Attribute	Kind	Limit_Kind
characters_per_linee	max/min	N/A
cased_identifier	template	ID
header_comment	template	header_comment
identifier	template	ID
unit_count	max/min	N/A
file_length	max/min	N/A
related_units	local	N/A

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

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

Use the file_length attribute to control the number of lines allowed in a file.

Use the cased_identifier attribute just like the identifier attribute, but in cases where the identifier is case-sensitive. Consider the following templates:

```
template T1 is file_layout
       limit identifier to "^[a-z][a-z0-9_]*$"
end
template T2 is file_layout
       limit cased_identifier to "^[a-z][a-z0-9_]*$"
end
```

The id1 identifier matches both templates, whereas the ID1 identifier only matches the first template. The identifier attribute is not case-sensitive because VHDL is not case-sensitive.

Use the characters_per_line attribute to allow the number of characters on each line to be constrained.

File Type Definition Template

The LRM (§3.4) defines the grammar for file type definition as follows:

```
file_type_definition ::= file of type_mark
```

The file_type_definition template is a primary template belonging to no classes. It contains the attribute shown in **Table 56**.

Table 56: file_type_definition Template

Attribute	Kind	Limit_Kind
type_mark	template	TYPE_MARK

Flipflop Template

The flipflop template is a primary template belonging to no classes. It contains the attributes shown in **Table 57**.

Table 57: flipflop Template

Attribute	Kind	Limit_Kind
asynchronous_initialization	limit	asynchronous_initialization
cased_identifier	template	ID
identifier	template	ID
input	template	EXPRESSION

Table 57: flipflop Template (Continued)

Attribute	Kind	Limit_Kind
output	template	EXPRESSION
synchronous_initialization	limit	synchronous_initialization
clock	template	clock
data_signal	template	data_signal
has_clock_as_data	local	N/A

Use the identifier attribute to control the names of flipflops inferred in the code. You can also do this using the output attribute.

Use the input attribute to control the expressions driving a flipflop.

Use the cased_identifier attribute just like the identifier attribute, but in cases where the identifier is case-sensitive. Consider the following templates:

```
template T1 is flipflop
       limit identifier to "^[a-z][a-z0-9_]*$"
end
template T2 is flipflop
       limit cased_identifier to "^[a-z][a-z0-9_]*$"
end
```

The id1 identifier matches both templates, whereas the ID1 identifier only matches the first template. The identifier attribute is not case-sensitive because VHDL is not case-sensitive.

Floating Type Definition Template

The LRM (§3.1.4) defines the grammar for floating point type as follows:

```
floating_point_type ::= range_constraint
```

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

Table 58: floating_type_definition Template

Attribute	Kind	Limit_Kind
range	template	range

For Loop Statement Template

The LRM (§8.9) defines the grammar for loop statement as follows:

```
loop_statement ::=
       [ loop_label : ]
             [ iteration_scheme ] loop
                   sequence_of_statements
             end loop [ loop_label ] ;
iteration_scheme ::=
        while condition
       | for loop_parameter_specification
parameter_specification ::=
       identifier in discrete_range
```

The for_loop_statement template is a primary template belonging to the SEQUENTIAL_STATEMENT class. It contains the attributes shown in **Table 59**.

Table 59: for_loop_statement Template

Attribute	Kind	Limit_Kind
cased_label	template	ID
label	template	ID
subtype_indication	template	subtype_indication
assertion_statement	template	assertion_statement
report_statement	template	report_statement
procedure_call_statement	template	procedure_call_statement
if_statement	template	if_statement
signal_assignment_statement	template	signal_assignment_statement
variable_assignment_statement	template	variable_assignment_statement
case_statement	template	case_statement
loop_statement	template	loop_statement
for_loop_statement	template	for_loop_statement
while_loop_statement	template	while_loop_statement
next_statement	template	next_statement
exit_statement	template	exit_statement
return_statement	template	return_statement

Table 59: for_loop_statement Template (Continued)

Attribute	Kind	Limit_Kind
wait_statement	template	wait_statement
null_statement	local	N/A
declarative_region	template	REGION_PART
statement_format	template	statement_format
statement_profile_s	aggregate	SEQUENTIAL_STATEMENT
null_statement	local	N/A
statement_profile_s	aggregate	SEQUENTIAL_STATEMENT
evaluation_time	set	N/A

Use the cased_label attribute just like the label attribute, but in cases where the label is case-sensitive. Consider the following templates:

```
template T1 is for_loop_statement
       limit label to "^[a-z][a-z0-9_]*$"
end
template T2 is for_loop_statement
       limit cased_label to "^[a-z][a-z0-9_]*$"
end
```

The lab1 label matches both templates, whereas the LAB1 label only matches the first template. The label attribute is not case-sensitive because VHDL is not case-sensitive.

Formal Parameter Template

The LRM (§2.1) defines the grammar for formal parameter as follows:

```
formal_parameter_list ::= parameter_interface_list
```

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

Table 60: formal_parameter Template

Attribute	Kind	Limit_Kind
cased_identifier	template	ID
identifier	template	ID
subtype_indication	template	subtype_indication
default	template	EXPRESSION
in	local	N/A
out	local	N/A
inout	local	N/A
constant	local	N/A
one_declaration_per_line	local	N/A
signal	local	N/A
comment	local	N/A
variable	local	N/A
file	local	N/A

The in, out, and inout attributes correspond to the different modes allowed for formal subprogram parameters. For example, the following command:

```
no out in formal_parameter severity error
```

means that parameters of mode “out” are not allowed.

The constant, signal, variable, and file attributes correspond to the different classes allowed for formal subprogram parameters. For example, the following command:

```
force signal in formal_parameter severity error
```

means that only parameters of class “signal” are allowed.

Use the cased_identifier attribute just like the identifier attribute, but in cases where the identifier is case-sensitive. Consider the following templates:

```
template T1 is formal_parameter
       limit identifier to "^[a-z][a-z0-9_]*$"
end
template T2 is formal_parameter
       limit cased_identifier to "^[a-z][a-z0-9_]*$"
end
```

The id1 identifier matches both templates, whereas the ID1 identifier only matches the first template. The identifier attribute is not case-sensitive because VHDL is not case-sensitive.

Use the one_declaration_per_line attribute to force a single line for each declaration.

Use the comment attribute to force a comment on the same line as the declaration.

Function Call Template

The LRM (§7.3.3) defines the grammar for function call as follows:

```
function_call ::= function_name [ ( actual_parameter_part ) ]
actual_parameter_part ::= parameter_association_list
```

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

Table 61: function_call Template

Attribute	Kind	Limit_Kind
cased_identifier	template	ID
identifier	template	ID
actual_parameter_part	template	association_list

Use the identifier attribute to control calls to functions outside the current unit. For example, to signal an error if the function RISING_EDGE in the package STD_LOGIC_1164 is called, you could write this rule as follows:

```
template BAD_FUNC_NAME is identifier
       limit error_id to "IEEE.STD_LOGIC_1164.RISING_EDGE"
end
limit identifier in function_call to BAD_FUNC_NAME
       message "Illegal function call" severity error
```

With this rule activated, Leda flags an error if the error_id attribute matches.

Use the actual_parameter_part attribute to point to association list templates that enable different constraints to be put on the parameters. For example, to impose named association on parameter lists unless there are only two or less parameters, you could write this rule as follows:

```
template AP1 is association_list
  min parameter_count is 2
end
limit actual_parameter_part in function_call to AP1 severity note
       if AP1 then
             force named_association in association_list
                   message "Named association must be used here"
                   severity error
       end if
```

Because it is not an error if AP1 is not matched, this example uses a severity of “note” to catch the mismatches. Leda then continues testing.

Use the cased_identifier attribute just like the identifier attribute, but in cases where the identifier is case-sensitive. Consider the following templates:

```
template T1 is function_call
       limit identifier to "^[a-z][a-z0-9_]*$"
end
template T2 is function_call
       limit cased_identifier to "^[a-z][a-z0-9_]*$"
end
```

The id1 identifier matches both templates, whereas the ID1 identifier only matches the first template. The identifier attribute is not case-sensitive because VHDL is not case-sensitive.

Generate Statement Template

The LRM (§9.7) defines the grammar for generate statement as follows:

```
generate_statement ::=
       generate_label :
             generation_scheme generate
                   [ { block_declarative_item }
             begin ]
                   { concurrent_statement}
             end generate [ generate_label ] ;
generation_scheme ::=
        for generate_parameter_specification
       | if condition
parameter_specification ::=
          identifier in discrete_range
```

The generate_statement template is a primary template belonging to the CONCURRENT_STATEMENT class. It contains the attributes shown in **Table 62**.

Table 62: generate_statement Template

Attribute	Kind	Limit_Kind
cased_label	template	ID
label	template	ID
discrete_range	template	range
condition	template	EXPRESSION
subprogram_declaration	template	subprogram_declaration
subprogram_body	template	subprogram_body
type_declaration	template	type_declaration
subtype_declaration	template	subtype_declaration
constant_declaration	template	constant_declaration
shared_variable_declaration	template	shared_variable_declaration
signal_declaration	template	signal_declaration
file_declaration	template	file_declaration
alias_declaration	template	alias_declaration
component_declaration	template	component_declaration
attribute_declaration	template	attribute_declaration

Table 62: generate_statement Template (Continued)

Attribute	Kind	Limit_Kind
attribute_specification	template	attribute_specification
disconnection_specification	template	disconnection_specification
configuration_specification	template	configuration_specification
use_clause	template	use_clause
group_template_declaration	template	group_template_declaration
group_declaration	template	group_declaration
assertion_statement	template	assertion_statement
procedure_call_statement	template	procedure_call_statement
process_statement	template	process_statement
block_statement	template	block_statement
selected_signal_assignment	template	selected_signal_assignment
conditional_signal_assignment	template	conditional_signal_assignment
component_instantiation_statement	template	component_instantiation_statement
generate_statement	template	generate_statement
declarative_region	template	REGION_PART

Use the condition attribute with if_generate statements and the discrete_range attribute with for_generate statements.

Use the cased_label attribute just like the label attribute, but in cases where the label is case-sensitive. Consider the following templates:

```
template T1 is generate_statement
       limit label to "^[a-z][a-z0-9_]*$"
end
template T2 is generate_statement
       limit cased_label to "^[a-z][a-z0-9_]*$"
end
```

The lab1 label matches both templates, whereas the LAB1 label only matches the first template. The label attribute is not case-sensitive because VHDL is not case-sensitive.

Generic Declaration Template

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

Table 63: generic_declaration Template

Attribute	Kind	Limit_Kind
cased_identifier	template	ID
identifier	template	ID
subtype_indication	template	subtype_indication
default	template	EXPRESSION
declarative_region	template	REGION_PART
constant	local	N/A
in	local	N/A
one_declaration_per_line	local	N/A
comment	local	N/A

Use the “constant” and “in” attributes to control the presence of the constant and in VHDL keywords. For example, the following command:

```
force in in generic_declaration
```

means that the “in” keyword must be present for generic declarations.

Use the cased_identifier attribute just like the identifier attribute, but in cases where the identifier is case-sensitive. Consider the following templates:

```
template T1 is generic_declaration
       limit identifier to "^[a-z][a-z0-9_]*$"
end
template T2 is generic_declaration
       limit cased_identifier to "^[a-z][a-z0-9_]*$"
end
```

The id1 identifier matches both templates, whereas the ID1 identifier only matches the first template. The identifier attribute is not case-sensitive because VHDL is not case-sensitive.

Use the one_declaration_per_line attribute to force a single line for each declaration.

Use the comment attribute to force a comment on the same line as the declaration.

Group Declarations

The LRM (§4.7) defines the grammar for group declaration as follows:

```
group_declaration ::=
group identifier  : group_template_name (group_constituent_list) ;
group_constituent_list ::= group_constituent { , group_constituent }
group_constituent ::= name  | character_literal
```

You cannot limit a group_declaration using the current version of Leda.

Group Template Declarations

The LRM (§4.6) defines the grammar for group template declaration as follows:

```
group_template_declaration ::=
       group identifier  is  (entity_class_entry_list)
entity_class_entry_list ::=
       entity_class_entry { , entity_class_entry }
entity_class_entry ::= entity_class [ <>]
```

You cannot limit a group_template_declaration using the current version of Leda.

Header Comment Template

Use the header_comment template to control the comments that come before a VHDL unit or subprogram and make sure that each field is present. The header_comment template is a secondary template belonging to no classes. It contains the attribute shown in **Table 64**.

Table 64: 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. Consider the following template:

```
--
-- unit : architecture A of E
-- author : Me
-- date : November 1998
--
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

limit header_comment in entity_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
```

Note

Observe the use of the allof keyword to associate several messages with the same command.

Identifier Template

The LRM (§13.3) defines the grammar for identifier as follows:

```
identifier ::= basic_identifier | extended_identifier
```

The identifier template is a primary template belonging to the ID class. It contains the attributes shown in the Table 65.

Table 65: identifier Template

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

Use character strings with the limit_id attribute to specify the acceptable value, as shown in the following example:

```
limit limit_id to "^STD_LOGIC_1164$"
```

Use the error_id attribute to indicate character strings that cannot be present. For example, Leda flags an error for the following rule when a signal declaration has the name BAD_SIG_NAME.

```
template BAD_SIG_ID is identifier
       limit error_id to "BAD_SIG_NAME"
end
limit identifier in signal_declaration to BAD_SIG_ID
       severity error
```

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

For more information on writing naming convention rules, see “Parameterizing Rules Using Regular Expressions” on page 28.

If Statement Template

The LRM (§8.7) defines the grammar for if statement as follows:

```
if_statement ::=
        [ if_label : ]
              if condition then
                    sequence_of_statements
              { elsif condition then
                    sequence_of_statements }
              [ else
                    sequence_of_statements }
        end if [ if_label ] ;
```

The if_statement template is a primary template belonging to the SEQUENTIAL_STATEMENT class. It contains the attributes shown in **Table 66**.

Table 66: if_statement Template

Attribute	Kind	Limit_Kind
cased_label	template	ID
label	template	ID
assertion_statement	template	assertion_statement
report_statement	template	report_statement
procedure_call_statement	template	procedure_call_statement
if_statement	template	if_statement
indentation_depth	max/min	N/A
signal_assignment_statement	template	signal_assignment_statement
variable_assignment_statement	template	variable_assignment_statement
case_statement	template	case_statement
loop_statement	template	loop_statement
for_loop_statement	template	for_loop_statement
while_loop_statement	template	while_loop_statement
next_statement	template	next_statement
exit_statement	template	exit_statement
return_statement	template	return_statement
wait_statement	template	wait_statement

Table 66: if_statement Template (Continued)

Attribute	Kind	Limit_Kind
null_statement	local	N/A
else	local	N/A
declarative_region	template	REGION_PART
statement_format	template	statement_format
condition_count	max/min	N/A
condition_s	aggregate	EXPRESSION
statement_profile_s	aggregate	SEQUENTIAL_STATEMENT

You can constrain all sequential statements and the statement profile within an if statement. Use the else clause to force or forbid the use of the else alternative.

Use the condition_count attribute to constrain the number of conditions in one if statement. For example:

```
if .... then
elsif ... then
else ... then
end if
```

In this example, there are three conditions in a single if statement. There are no nested if statements. So, the following rule is violated by this example:

```
max condition_count in if_statement is 2
```

Use the indentation_depth attribute to constrain the number of nested if's in an if statement. For example:

```
if ... then
    if ... then
        if ... then
        end if
    end if
end if
```

In this example, there are three nested if's. Note that the current if is also counted. So, the following rule is violated by this example:

```
max indentation_depth in if_statement is 2
```

Use the cased_label attribute just like the label attribute, but in cases where the label is case-sensitive. Consider the following templates:

```
template T1 is if_statement
       limit label to "^[a-z][a-z0-9_]*$"
end
template T2 is if_statement
       limit cased_label to "^[a-z][a-z0-9_]*$"
end
```

The lab1 label matches both templates, whereas the LAB1 label only matches the first template. The label attribute is not case-sensitive because VHDL is not case-sensitive.

Indexed Name Template

The LRM (§6.4) defines the grammar for indexed name as follows:

```
indexed_name ::= prefix ( expression { , expression } )
```

The indexed_name template is a secondary template belonging to the EXPRESSION, NAME, and TARGET classes. It contains the attributes shown in **Table 67**.

Table 67: indexed_name Template

Attribute	Kind	Limit_Kind
cased_identifier	template	ID
identifier	template	ID
object_definition	template	OBJECT_ITEM
range_overflow	local	N/A
expression	template	EXPRESSION
flipflop	template	flipflop
latch	template	latch

Use the object_definition attribute to control the type of definition associated with a simple_name. For example, to consider just the ports in a given expression, you could write the following rule:

```
template CLK_NAME is simple_name
       limit object_definition to port_declaration
end
```

There are some application-specific attributes associated with this template that make it easier to write commonly-used rules. For example:

- Use the flipflop attribute to control whether or not the object infers a flipflop
- Use the latch attribute to control whether or not the object infers a latch

Use the cased_identifier attribute just like the identifier attribute, but in cases where the identifier is case-sensitive. Consider the following templates:

```
template T1 is indexed_name
       limit identifier to "^[a-z][a-z0-9_]*$"
end
template T2 is indexed_name
       limit cased_identifier to "^[a-z][a-z0-9_]*$"
end
```

The id1 identifier matches both templates, whereas the ID1 identifier only matches the first template. The identifier attribute is not case-sensitive because VHDL is not case-sensitive.

Integer Type Definition Template

The LRM (§3.1.2) defines the grammar for integer type definition as follows:

```
integer_type_definition ::= range_constraint
```

The integer_type_definition template is a primary template belonging to no classes. It contains the attribute shown in **Table 68**.

Table 68: integer_type_definition Template

Attribute	Kind	Limit_Kind
range	template	range

Interface File Declaration Template

The LRM (§4.3.2) defines the grammar for interface file declaration as follows:

```
interface_file_declaration ::= file identifier_list subtype_indication
```

The interface_file_declaration template is a primary template belonging to the OBJECT_ITEM class. It contains the attributes shown in Table 69.

Table 69: interface_file_declaration Template

Attribute	Kind	Limit_Kind
cased_identifier	template	ID
identifier	template	ID
subtype_indication	template	subtype_indication
declarative_region	template	REGION_PART
file	local	N/A
one_declaration_per_line	local	N/A
comment	local	N/A

Use the cased_identifier attribute just like the identifier attribute, but in cases where the identifier is case-sensitive. Consider the following templates:

```
template T1 is interface_file_declaration
       limit identifier to "^[a-z][a-z0-9_]*$"
end
template T2 is interface_file_declaration
       limit cased_identifier to "^[a-z][a-z0-9_]*$"
end
```

The id1 identifier matches both templates, whereas the ID1 identifier only matches the first template. The identifier attribute is not case-sensitive because VHDL is not case-sensitive.

Use the one_declaration_per_line attribute to force a single line for each declaration.

Use the comment attribute to force a comment on the same line as the declaration.

Interface Variable Declaration Template

The LRM (§4.3.2) defines the grammar for interface variable declaration as follows:

```
interface_variable_declaration ::=
   [variable] identifier_list : [ mode ] subtype_indication
   [ := static_expression ]
mode ::=  in | out | inout | buffer | linkage
```

The interface_variable_declaration template is a primary template belonging to the OBJECT_ITEM class. It contains the attributes shown in **Table 70**.

Table 70: interface_variable_declaration Template

Attribute	Kind	Limit_Kind
cased_identifier	template	ID
identifier	template	ID
subtype_indication	template	subtype_indication
default	template	EXPRESSION
declarative_region	template	REGION_PART
variable	local	N/A
in	local	N/A
out	local	N/A
inout	local	N/A
buffer	local	N/A
linkage	local	N/A
one_declaration_per_line	local	N/A
comment	local	N/A

Use the cased_identifier attribute just like the identifier attribute, but in cases where the identifier is case-sensitive. Consider the following templates:

```
template T1 is interface_variable_declaration
       limit identifier to "^[a-z][a-z0-9_]*$"
end
template T2 is interface_variable_declaration
       limit cased_identifier to "^[a-z][a-z0-9_]*$"
end
```

The id1 identifier matches both templates, whereas the ID1 identifier only matches the first template. The identifier attribute is not case-sensitive because VHDL is not case-sensitive.

Use the one_declaration_per_line attribute to force a single line for each declaration.

Use the comment attribute to force a comment on the same line as the declaration.

Latch Template

The latch template is a primary template belonging to no classes. It contains the attributes shown in **Table 71**.

Table 71: latch Template

Attribute	Kind	Limit_Kind
asynchronous_initialization	limit	asynchronous_initialization
cased_identifier	template	ID
identifier	template	ID
tinput	template	EXPRESSION
output	template	EXPRESSION
synchronous_initialization	limit	synchronous_initialization
clock	template	clock
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. Note that you can accomplish the same thing using the output attribute. For example, to ensure that latches all have an identifier with the suffix “_latch”, you could define the following template:

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

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

Use the cased_identifier attribute just like the identifier attribute, but in cases where the identifier is case-sensitive. Consider the following templates:

```
template T1 is latch
       limit identifier to "^[a-z][a-z0-9_]*$"
end
template T2 is latch
       limit cased_identifier to "^[a-z][a-z0-9_]*$"
end
```

The id1 identifier matches both templates, whereas the ID1 identifier only matches the first template. The identifier attribute is not case-sensitive because VHDL is not case-sensitive.

Literal Template

The LRM (§7.3.1) defines the grammar for literal template as follows:

```
literal ::=        numeric_literal
                   | enumeration_literal
                   | string_literal
                   | bit_string_literal
                   | null
numeric_literal ::= abstract_literal | physical_literal
```

The literal template is a primary template belonging to the EXPRESSION class. It contains the attributes shown in **Table 72**.

Table 72: literal Template

Attribute	Kind	Limit_Kind
value	template	literal
null	local	N/A
use_exponent	local	N/A
base	set	N/A
base_specifier	set	N/A
value_type	set	N/A
evaluation_time	set	N/A

You can use integer values or character strings with the value attribute, as shown in the following examples:

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

In the first example, the integer value zero is represented, whereas in the second case, the enumerated literal “0” or 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:

- any_literal_type
- abstract_literal_type
- physical_literal_type
- enumerated_literal_type
- string_literal_type
- bit_string_literal_type

You can set the evaluation_time attribute to any of the following enumerated literals:

- undefined
- locally_static_evaluation
- globally_static_evaluation
- dynamic_evaluation

For example, the following rule sets the evaluation_time attribute to the locally_static_evaluation enumerated literal.

```
set evaluation_time to locally_static_evaluation
```

Loop Statement Template

The LRM (§8.9) defines the grammar for loop statement as follows:

```
loop_statement ::=
       [ loop_label : ]
             [ iteration_scheme ] loop
                   sequence_of_statements
             end loop [ loop_label ] ;
iteration_scheme ::=
          while condition
       | for loop_parameter_specification
parameter_specification ::=
       identifier in discrete_range
```

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

Table 73: loop_statement Template

Attribute	Kind	Limit_Kind
cased_label	template	ID
label	template	ID
assertion_statement	template	assertion_statement
report_statement	template	report_statement
procedure_call_statement	template	procedure_call_statement
if_statement	template	if_statement
signal_assignment_statement	template	signal_assignment_statement
variable_assignment_statement	template	variable_assignment_statement
case_statement	template	case_statement
loop_statement	template	loop_statement
for_loop_statement	template	for_loop_statement
while_loop_statement	template	while_loop_statement
next_statement	template	next_statement
exit_statement	template	exit_statement
return_statement	template	return_statement
wait_statement	template	wait_statement

Table 73: loop_statement Template (Continued)

Attribute	Kind	Limit_Kind
null_statement	local	N/A
declarative_region	template	REGION_PART
statement_format	template	statement_format
statement_profile_s	aggregate	SEQUENTIAL_STATEMENT

Use the cased_label attribute just like the label attribute, but in cases where the label is case-sensitive. Consider the following templates:

```
template T1 is loop_statement
       limit label to "^[a-z][a-z0-9_]*$"
end
template T2 is loop_statement
       limit cased_label to "^[a-z][a-z0-9_]*$"
end
```

The lab1 label matches both templates, whereas the LAB1 label only matches the first template. The label attribute is not case-sensitive because VHDL is not case-sensitive.

Next Statement Template

The LRM (§8.10) defines the grammar for next statement as follows:

```
next_statement ::= [ label : ] next [ loop_label ] [ when condition ] ;
```

The next_statement template is a primary template belonging to the SEQUENTIAL_STATEMENT class. It contains the attributes shown in **Table 74**.

Table 74: next_statement Template

Attribute	Kind	Limit_Kind
cased_label	template	ID
label	template	ID
loop_label	template	ID
condition	template	EXPRESSION
declarative_region	template	REGION_PART

Use the cased_label attribute just like the label attribute, but in cases where the label is case-sensitive. Consider the following templates:

```
template T1 is loop_statement
       limit label to "^[a-z][a-z0-9_]*$"
end
template T2 is loop_statement
       limit cased_label to "^[a-z][a-z0-9_]*$"
end
```

The lab1 label matches both templates, whereas the LAB1 label only matches the first template. The label attribute is not case-sensitive because VHDL is not case-sensitive.

Package Body Template

The LRM (§2.6) defines the grammar for package body as follows:

```
package_body ::=
       package body package_simple_name is
             package_body_declarative_part
       end [ package body ] [ package_simple_name ] ;

package_body_declarative_part ::=
       { package_body_declarative_item }

package_body_declarative_item ::=
          subprogram_declaration
       | subprogram_body
       | type_declaration
       | subtype_declaration
       | constant_declaration
       | shared_variable_declaration
       | file_declaration
       | alias_declaration
       | use_clause
       | group_template_declaration
       | group declaration
```

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

Table 75: package_body Template

Attribute	Kind	Limit_Kind
subprogram_declaration	template	subprogram_declaration
subprogram_body	template	subprogram_body
type_declaration	template	type_declaration
subtype_declaration	template	subtype_declaration
constant_declaration	template	constant_declaration
shared_variable_declaration	template	shared_variable_declaration
file_declaration	template	file_declaration
alias_declaration	template	alias_declaration
use_clause	template	use_clause
group_template_declaration	template	group_template_declaration

Table 75: package_body Template (Continued)

Attribute	Kind	Limit_Kind
group_declaration	template	group_declaration
design_unit	template	design_unit
header_comment	template	header_comment
statement_format	template	statement_format
file_name	template	ID
ncs_file_name	template	ID
file_layout	template	file_layout
line_count	max/min	N/A
file_length	max/min	N/A

Employ the use-clause attribute to limit use-clauses inside the declaration. To constrain use and library clauses appearing before the declaration, employ the design_unit template instead.

There are some application-specific attributes associated with this template that make it easier to write commonly-used rules. For example:

- Use the file_name attribute to configure the name of containing file, as shown in the following example:

```
limit file_name in package_body to "<package>.Body.vhd"
```

- Use the line_count attribute to specify the maximum or minimum lines of code in a unit.

Package Declaration Template

The LRM (§2.5) defines the grammar for package declaration as follows:

```
package_declaration ::=
       package identifier is
             package_declarative_part
       end [ package ] [ package_smple_name ] ;

package_declarative_part ::=
       { package_declarative_item }

package_declarative_item ::=
          subprogram_declaration
        | type_declaration
        | subtype_declaration
        | constant_declaration
        | signal_declaration
        | shared_variable_declaration
        | file_declaration
        | alias_declaration
        | component_declaration
        | attribute_declaration
        | attributepecification
        | disconnectionpecification
        | use_clause
        | group_template_declaration
        | group declaration
```

The package_declaration template is a primary template belonging to the REGION_PART and OBJECT_ITEM classes. It contains the attributes shown in **Table 76**.

Table 76: package_declaration Template

Attribute	Kind	Limit_Kind
cased_identifier	template	ID
identifier	template	ID
subprogram_declaration	template	subprogram_declaration
type_declaration	template	type_declaration
subtype_declaration	template	subtype_declaration
constant_declaration	template	constant_declaration
shared_variable_declaration	template	shared_variable_declaration

Table 76: package_declaration Template (Continued)

Attribute	Kind	Limit_Kind
signal_declaration	template	signal_declaration
file_declaration	template	file_declaration
alias_declaration	template	alias_declaration
component_declaration	template	component_declaration
attribute_declaration	template	attribute_declaration
attribute_specification	template	attribute_specification
declaration_profile_s	aggregate	DECLARATIVE_ITEM
disconnection_specification	template	disconnection_specification
file_length	max/min	N/A
use_clause	template	use_clause
group_template_declaration	template	group_template_declaration
group_declaration	template	group_declaration
header_comment	template	header_comment
design_unit	template	design_unit
statement_format	template	statement_format
file_name	template	ID
ncs_file_name	template	ID
file_layout	template	file_layout
line_count	max/min	N/A

Employ the use-clause attribute to limit the use-clauses inside the declaration. To constrain use and library clauses appearing before the declaration, use the design_unit template instead.

There are some application-specific attributes associated with this template that make it easier to write commonly-used rules. For example:

- Use the file_name attribute to configure the name of the containing file, as shown in the following example:

```
limit file_name in package_declaration to "<package>.vhd"
```

- Use the line_count attribute to specify the maximum or minimum lines of code in a unit.

Use the cased_identifier attribute just like the identifier attribute, but in cases where the identifier is case-sensitive. Consider the following templates:

```
template T1 is package_declaration
       limit identifier to "^[a-z][a-z0-9_]*$"
end
template T2 is package_declaration
       limit cased_identifier to "^[a-z][a-z0-9_]*$"
end
```

The id1 identifier matches both templates, whereas the ID1 identifier only matches the first template. The identifier attribute is not case-sensitive because VHDL is not case-sensitive.

Physical Type Definition Template

The LRM (§3.1.3) defines the grammar for physical type definition as follows:

```
physical_type_definition ::=
       range_constraint
             units
                   primary_unit_declaration
                   { secondary_unit_declaration }
             end units [ physical_type_simple_name]

primary_unit_declaration ::= identifier

secondary_unit_declaration ::= identifier = physical_literal ;

physical_literal ::= [ abstract_literal ] unit_name
```

The physical_type_definition template is a primary template belonging to no classes. It contains the attributes shown in **Table 77**.

Table 77: physical_type_definition Template

Attribute	Kind	Limit_Kind
cased_identifier	template	ID
range	template	range
identifier	template	ID

Use the cased_identifier attribute just like the identifier attribute, but in cases where the identifier is case-sensitive. Consider the following templates:

```
template T1 is physical_type_definition
       limit identifier to "^[a-z][a-z0-9_]*$"
end
template T2 is physical_type_definition
       limit cased_identifier to "^[a-z][a-z0-9_]*$"
end
```

The id1 identifier matches both templates, whereas the ID1 identifier only matches the first template. The identifier attribute is not case-sensitive because VHDL is not case-sensitive.

Port Declaration Template

The port_declaration template is a primary template belonging to the OBJECT_ITEM class. It contains the attributes shown in Table 78.

Table 78: port_declaration Template

Attribute	Kind	Limit_Kind
cased_identifier	template	ID
identifier	template	ID
subtype_indication	template	subtype_indication
default	template	EXPRESSION
declarative_region	template	REGION_PART
signal	local	N/A
in	local	N/A
out	local	N/A
inout	local	N/A
buffer	local	N/A
linkage	local	N/A
bus	local	N/A
is_clock	local	ID
is_reset	local	ID
signals_driven	template	DRIVEN_OBJECT

Table 78: port_declaration Template (Continued)

Attribute	Kind	Limit_Kind
driving_expression	template	EXPRESSION
flipflop	template	flipflop
latch	template	latch
tristate	local	N/A
one_declaration_per_line	local	N/A
outputs_driven	max/min	N/A
register	local	N/A
comment	local	N/A
connections	max/min	N/A

Note

The port_declaration template contains the same set of attributes as the interface_signal_declaration template.

To force the presence of the “in” mode in every port declaration, you could write:

```
force in in port_declaration
```

Because “in” is the default mode, the others must be present.

Use the declarative_region attribute to limit the regions where this item can be declared. You can use the declarative_regions attribute to point to a list of templates of the REGION_PART class.

Use the register and bus attributes to control the presence of the register and bus VHDL keywords.

There are some application-specific attributes associated with this template that make it easier to write commonly-used rules. For example:

- Use the signals_driven attribute to control the signals driven by the current signal.
- Use the driving_expression attribute to control the expressions accepted as drivers for the current signal.
- Use the flipflop attribute to control whether or not the signal infers a flipflop.
- Use the latch attribute to control whether or not the signal infers a latch.

- Use the tristate attribute to constrain tristate signals.
- Use the connections attribute to control the number of times the signal drives or is driven by other signals.
- Use the outputs_driven attribute to constrain the number of directly driven outputs. For example, in the following code, the number of outputs driven by d1 is 1, whereas the number driven by d2 is 0.

```
q1<=d1;         --d1 drives port q1 directly
if cond then    --infers mux
    q2<=d2;     --d2 is input to mux driving port q2
else
    q2<='1';
end if;
```

To write a rule forbidding the use of feedthrough ports, you could use the following template:

```
template FEEDTHRU_PORT is port_declaration
       max outputs_driven is 0
end
```

Use the cased_identifier attribute just like the identifier attribute, but in cases where the identifier is case-sensitive. Consider the following templates:

```
template T1 is interface_signal_declaration
       limit identifier to "^[a-z][a-z0-9_]*$"
end
template T2 is interface_signal_declaration
       limit cased_identifier to "^[a-z][a-z0-9_]*$"
end
```

The id1 identifier matches both templates, whereas the ID1 identifier only matches the first template. The identifier attribute is not case-sensitive because VHDL is not case-sensitive.

Use the one_declaration_per_line attribute to force a single line for each declaration.

Use the comment attribute to force a comment on the same line as the declaration.

Procedure Call Statement Template

The LRM (§8.6) defines the grammar for procedure call statement as follows:

```
procedure_call_statement ::=  [ label : ] procedure_call ;
procedure_call ::= procedure_name [ ( actual_parameter_part ) ]
```

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

Table 79: procedure_call_statement Template

Attribute	Kind	Limit_Kind
label	template	ID
cased_identifier	template	ID
identifier	template	ID
actual_parameter_part	template	association_list
declarative_region	template	REGION_PART

Use the cased_identifier attribute just like the identifier attribute, but in cases where the identifier is case-sensitive. Consider the following templates:

```
template T1 is procedure_call_statement
       limit identifier to "^[a-z][a-z0-9_]*$"
end
template T2 is procedure_call_statement
       limit cased_identifier to "^[a-z][a-z0-9_]*$"
end
```

The id1 identifier matches both templates, whereas the ID1 identifier only matches the first template. The identifier attribute is not case-sensitive because VHDL is not case-sensitive.

Process Statement Template

The LRM (§9.2) defines the grammar for process statement as follows:

```
process_statement ::=
       [ process_label : ]
            [ postponed ]process [ ( sensitivity_list ) ] [ is ]
                 process_declarative_part
            begin
                 process_statement_part
            end [ postponed ] process [ process_label ] ;
process_declarative_part ::=
       { process_declarative_item }
process_declarative_item ::=
        subprogram_declaration
       | subprogram_body
       | type_declaration
       | subtype_declaration
       | constant_declaration
       | variable_declaration
       | file_declaration
       | alias_declaration
       | attribute_declaration
       | attribute_specification
       | use_clause
       | group_type_declaration
       | group declaration
process_statement_part ::=
       { sequential_statement }
```

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

Table 80: process_statement Template

Attribute	Kind	Limit_Kind
cased_label	template	ID
label	template	ID
sensitivity	template	EXPRESSION
subprogram_declaration	template	subprogram_declaration
subprogram_body	template	subprogram_body
type_declaration	template	type_declaration
subtype_declaration	template	subtype_declaration
constant_declaration	template	constant_declaration

Table 80: process_statement Template (Continued)

Attribute	Kind	Limit_Kind
variable_declaration	template	variable_declaration
file_declaration	template	file_declaration
alias_declaration	template	alias_declaration
attribute_declaration	template	attribute_declaration
attribute_specification	template	attribute_specification
use_clause	template	use_clause
group_template_declaration	template	group_template_declaration
group_declaration	template	group_declaration
unused_declaration	local	N/A
assertion_statement	template	assertion_statement
report_statement	template	report_statement
procedure_call_statement	template	procedure_call_statement
if_statement	template	if_statement
signal_assignment_statement	template	signal_assignment_statement
variable_assignment_statement	template	variable_assignment_statement
case_statement	template	case_statement
loop_statement	template	loop_statement
for_loop_statement	template	for_loop_statement
while_loop_statement	template	while_loop_statement
next_statement	template	next_statement
exit_statement	template	exit_statement
return_statement	template	return_statement
wait_statement	template	wait_statement
null_statement	local	N/A
postponed	local	N/A

Table 80: process_statement Template (Continued)

Attribute	Kind	Limit_Kind
header_comment	template	header_comment
latch	template	latch
flipflop	template	flipflop
fsm	local	N/A
synchronous_reset	template	synchronous_reset
asynchronous_reset	template	asynchronous_reset
statement_format	template	statement_format
declarative_region	template	REGION_PART
combinatorial	local	N/A
complete_sensitivity	local	N/A
missing_signals_in_sensitivity_list	local	N/A
redundancy_in_sensitivity_list	local	N/A
initialize_signals	local	N/A
fully_assign_signals	local	N/A
fully_assign_variables	local	N/A
initialize_variables	local	N/A
maximum_variable_usage	local	N/A
dead_code	local	N/A
sensitivity_count	max/min	N/A
clock_expression_count	max/min	N/A
clock_signal	max/min	N/A
statement_profile_s	aggregate	SEQUENTIAL_STATEMENT

Use the statement_profile_s attribute to dictate the order of statements. For example, to force synchronous processes to appear before asynchronous processes, you could write the following rule:

```
limit statement_profile_s in process_statement to
       first {VAR_ASS_TPL} then IF_STM, others {}
       severity error
```

where VAR_ASS_TPL and IF_STM are two valid sequential statement templates. The {} after the others keyword means that any statement is accepted. You can only use this notation for the last set of templates.

Use the latch and flipflop attributes to control the different types of memory inferred. For example, to forbid the use of latches in a process, you could write the following rule:

```
no latch in process_statement
```

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

```
force fsm in process_statement
```

Use the combinatorial attribute to force or forbid the presence of memory elements in a process. Note that a combinatorial process is defined as one that has no clock.

Use the clock_expression_count attribute to control the number of different clock expressions in a subprogram (procedure) body.

Use clock_signal attribute to control the number of different clock signals in a subprogram (procedure) body.

If you force the complete_sensitivity attribute to be present, Leda ensures that every object in the sensitivity list is read in the process and that every object read is in the sensitivity list, including objects that are also modified.

The initialize_variables and initialize_signals attributes cause Leda to generate an error if, in a combinatorial process, a variable or signal is read before being written on any flow of control, or if it is read without being written.

If you force the fully_assign_signals attribute, Leda ensures that every signal assigned in the process is fully assigned on every possible flow of control through the process.

If you force the maximum_variable_usage attribute, Leda ensures that wherever possible, variables are used instead of signals.

Use the cased_label attribute just like the label attribute, but in cases where the label is case-sensitive. Consider the following templates:

```
template T1 is process_statement
       limit label to "^[a-z][a-z0-9_]*$"
end
template T2 is process_statement
       limit cased_label to "^[a-z][a-z0-9_]*$"
end
```

The lab1 label matches both templates, whereas the LAB1 label only matches the first template. The label attribute is not case-sensitive because VHDL is not case-sensitive.

Range Template

The LRM (§3.1) defines the grammar for range constraint as follows:

```
range_constraint ::= range range
range ::= range_attribute_name | simple_expression direction
  simple_expression
direction ::=  to | downto
```

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

Table 81: range Template

Attribute	Kind	Limit_Kind
left_expression	template	EXPRESSION
right_expression	template	EXPRESSION
range_attribute	local	N/A
range_index	local	N/A
null_range	local	N/A
to	local	N/A
downto	local	N/A
evaluation_time	set	N/A
high_bound	max/min	N/A
low_bound	max/min	N/A

For example, the following rule:

```
no range_attribute in range severity error
```

means that any time a range_attribute appears in the VHDL code, Leda flags an error.

Whereas, this next rule:

```
no null_range in range severity error
```

means that ranges of the form 3 to 0 or 0 downto 5 cause Leda to flag an error.

The following rule:

```
no downto in range severity error
```

means that the presence of the downto VHDL keyword causes Leda to flag an error. The same is true for the to attribute and the presence of the to VHDL keyword in a range expression.

For the following rule:

```
set evaluation_time in range to locally_static_evaluation
       severity error
```

both left and right range expressions must be locally static; otherwise, Leda flags an error.

Use the high_bound and low_bound attributes to specify fixed, static ranges without defining literal templates. For example, the following rules:

```
max    low_bound         in range is 0 severity error
no     downto            in range severity error
```

mean that all ranges must be ascending and start at 0.

To build more complex limitations for either range expression, use the left_expression and right_expression attributes for the left-hand and right-hand expressions, respectively. To limit these attributes, point them to templates of type expression.

Record Type Definition Template

The LRM (§3.2.2) defines the grammar for record type definition as follows:

```
record_type_definition ::=
       record
              element_declaration
            { element_declaration }
       end record [ record_type_simple_name ]

element_declaration ::=
       identifier_list : element_subtype_definition ;
identifier_list ::=  identifier { , identifier }
element_subtype_definition ::=  subtype_indication
```

The record_type_definition template is a primary template belonging to no classes. It contains the attributes shown in **Table 82.**

Table 82: record_type_definition Template

Attribute	Kind	Limit_Kind
element_declaration	template	element_declaration
type_mark	template	TYPE_MARK

Report Statement Template

The LRM (§8.3) defines the grammar for report statement as follows:

```
report_statement ::= [ label : ] report expression [ severity expression ]
```

The report_statement template is a primary template belonging to the SEQUENTIAL_STATEMENT class. It contains the attributes shown in **Table 83**.

Table 83: report_statement Template

Attribute	Kind	Limit_Kind
cased_label	template	ID
label	template	ID
report_expression	template	EXPRESSION
severity_expression	template	EXPRESSION
declarative_region	template	REGION_PART

Use the cased_label attribute just like the label attribute, but in cases where the label is case-sensitive. Consider the following templates:

```
template T1 is report_statement
       limit label to "^[a-z][a-z0-9_]*$"
end
template T2 is report_statement
       limit cased_label to "^[a-z][a-z0-9_]*$"
end
```

The lab1 label matches both templates, whereas the LAB1 label only matches the first template. The label attribute is not case-sensitive because VHDL is not case-sensitive.

Return Statement Template

The LRM (§8.12) defines the grammar for return statement as follows:

```
return_statement ::= [ label : ] return expression ;
```

The return_statement template is a primary template belonging to the SEQUENTIAL_STATEMENT class. It contains the attributes shown in **Table 84**.

Table 84: return_statement Template

Attribute	Kind	Limit_Kind
cased_label	template	ID
label	template	ID
expression	template	EXPRESSION
declarative_region	template	REGION_PART

Use the cased_label attribute just like the label attribute, but in cases where the label is case-sensitive. Consider the following templates:

```
template T1 is return_statement
       limit label to "^[a-z][a-z0-9_]*$"
end
template T2 is return_statement
       limit cased_label to "^[a-z][a-z0-9_]*$"
end
```

The lab1 label matches both templates, whereas the LAB1 label only matches the first template. The label attribute is not case-sensitive because VHDL is not case-sensitive.

Selected Name Template

The LRM (§6.3) defines the grammar for selected name as follows:

```
selected_name ::= prefix . suffix
suffix ::=
        simple_name
       | character_literal
       | operator_symbol
       | all
```

The selected_name template is a secondary template belonging to the EXPRESSION, NAME, and TARGET classes. It contains the attributes shown in **Table 85**.

Table 85: selected_name Template

Attribute	Kind	Limit_Kind
cased_identifier	template	ID
identifier	template	ID
object_definition	template	OBJECT_ITEM

Use the cased_identifier attribute just like the identifier attribute, but in cases where the identifier is case-sensitive. Consider the following templates:

```
template T1 is selected_name
       limit identifier to "^[a-z][a-z0-9_]*$"
end
template T2 is selected_name
       limit cased_identifier to "^[a-z][a-z0-9_]*$"
end
```

The id1 identifier will match both templates, whereas the ID1 identifier only matches the first template. The identifier attribute is not case-sensitive because VHDL is not case-sensitive.

Selected Signal Assignment Template

The LRM (§9.5.2) defines the grammar for selected signal assignment as follows:

```
selected_signal_assignment ::=
       with expression select
             target <= options selected_waveforms ;
```

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

Table 86: selected_signal_assignment Template

Attribute	Kind	Limit_Kind
cased_label	template	ID
label	template	ID
target	template	TARGET
reject_expression	template	EXPRESSION
expression	template	EXPRESSION
operand_size_mismatch	local	N/A
postponed	local	N/A
range_overflow	local	N/A
statement_format	template	statement_format
transport	local	N/A
inertial	local	N/A
guarded	local	N/A
read_write	local	N/A
waveform_count	max/min	N/A
selected_waveforms_s	aggregate	selected_waveforms

Use the transport, inertial, and guarded attributes to force or forbid the presence of the corresponding VHDL keywords (transport, inertial, and guarded).

Use the waveform_count attribute to set the maximum number of waveforms allowed.

Use the cased_label attribute just like the label attribute, but in cases where the label is case-sensitive. Consider the following templates:

```
template T1 is selected_signal_assignment
       limit label to "^[a-z][a-z0-9_]*$"
end
template T2 is selected_signal_assignment
       limit cased_label to "^[a-z][a-z0-9_]*$"
end
```

The lab1 label matches both templates, whereas the LAB1 label only matches the first template. The label attribute is not case-sensitive because VHDL is not case-sensitive.

Selected Waveforms Template

The LRM (§9.5.2) defines the grammar for selected waveforms as follows:

```
selected_waveforms ::=
       { waveform when choices , }
         waveform when choices
```

The selected_waveforms template is a secondary template belonging to no classes. It contains the attributes shown in **Table 87** and is used in the Selected Signal Assignment Template.

Table 87: selected_waveforms Template

Attribute	Kind	Limit_Kind
waveform	template	waveform
choice	template	EXPRESSION
others	local	N/A

Use the others attribute to force or forbid the presence of the others VHDL keyword.

Shared Variable Declaration Template

The shared_variable_declaration template is a primary template belonging to the OBJECT_ITEM class. It contains the attributes shown in **Table 88**.

Table 88: shared_variable_declaration Template

Attribute	Kind	Limit_Kind
cased_identifier	template	ID
identifier	template	ID
subtype_indication	template	subtype_indication
default	template	EXPRESSION
declarative_region	template	REGION_PART
one_declaration_per_line	local	N/A
comment	local	N/A

Use the cased_identifier attribute just like the identifier attribute, but in cases where the the identifier is case-sensitive. Consider the following templates:

```
template T1 is variable_declaration
       limit identifier to "^[a-z][a-z0-9_]*$"
end
template T2 is variable_declaration
       limit cased_identifier to "^[a-z][a-z0-9_]*$"
end
```

The id1 identifier matches both templates, whereas the ID1 identifier only matches the first template. The identifier attribute is not case-sensitive because VHDL is not case-sensitive.

Use the one_declaration_per_line attribute to force a single line for each declaration.

Use the comment attribute to force a comment on the same line as the declaration.

Signal Assignment Statement Template

The LRM (§8.4) defines the grammar for signal assignment statement as follows:

```
signal_assignment_statement ::=
       [ label : ] target <=  [ delay_mechanism ]  waveform ;
target ::=
        name
       | aggregate
delay_mechanism ::=
        transport
       | [ reject time_expression ] inertial
```

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

Table 89: signal_assignment_statement Template

Attribute	Kind	Limit_Kind
cased_label	template	ID
label	template	ID
target	template	TARGET
waveform	template	waveform
operand_size_mismatch	local	N/A
reject_expression	template	EXPRESSION
declarative_region	template	REGION_PART
range_overflow	local	N/A
transport	local	N/A
inertial	local	N/A
read_write	local	N/A
waveform_count	max/min	N/A

Use the transport and inertial keywords to force or forbid the presence of the corresponding transport and inertial VHDL keywords.

Use the cased_label attribute just like the label attribute, but in cases where the label is case-sensitive. Consider the following templates:

```
template T1 is signal_assignment_statement
       limit label to "^[a-z][a-z0-9_]*$"
end
template T2 is signal_assignment_statement
       limit cased_label to "^[a-z][a-z0-9_]*$"
end
```

The lab1 label matches both templates, whereas the LAB1 label only matches the first template. The label attribute is not case-sensitive because VHDL is not case-sensitive.

Signal Declaration Template

The LRM (§4.3.1.2) defines the grammar for signal declaration as follows:

```
signal_declaration ::=
   signal identifier_list:subtype_indication [ signal_kind ]
   [:= expression] ;
signal_kind ::=  register  |  bus
```

The signal_declaration template is a primary template belonging to the OBJECT_ITEM class. It contains the attributes shown in Table 90.

Table 90: signal_declaration Template

Attribute	Kind	Limit_Kind
cased_identifier	template	ID
identifier	template	ID
subtype_indication	template	subtype_indication
default	template	EXPRESSION
declarative_region	template	REGION_PART
outputs_driven	max/min	N/A
register	local	N/A
bus	local	N/A
signals_driven	template	DRIVEN_OBJECT
driving_expression	template	EXPRESSION
flipflop	template	flipflop
latch	template	latch

Table 90: signal_declaration Template (Continued)

Attribute	Kind	Limit_Kind
one_declaration_per_line	local	N/A
comment	local	N/A
tristate	local	N/A
is_clock	local	ID
is_reset	local	ID

Use the declarative_region attribute to limit where this item is allowed. You can point the declarative_region attribute at templates of the REGION_PART class.

Use the register and bus attributes to control the presence of the corresponding register and bus VHDL keywords.

There are some application-specific attributes that you can use with this template to make it easier to write commonly-used rules. For example:

- Use the signals_driven attribute to control the signals driven by the current signal.
- Use the driving_expression attribute to control the expressions allowed as drivers for the current signal.
- Use the flipflop attribute to control whether or not the signal infers a flipflop.
- Use the latch attribute to control whether or not the signal infers a latch.
- Use the tristate attribute to control whether or not the signal infers a latch.
- Use the outputs_driven attribute to control the number of directly driven outputs. For example, in the following code the number of outputs driven by d1 is 1, whereas the number driven by d2 is 0.

```
q1<=d1;         --d1 drives port q1 directly
if cond then    --infers mux
    q2<=d2;     --d2 is input to mux driving port q2
else
    q2<='1';
end if;
```

Use the cased_identifier attribute just like the identifier attribute, but in cases where the identifier is case-sensitive. Consider the following templates:

```
template T1 is signal_declaration
       limit identifier to "^[a-z][a-z0-9_]*$"
end
template T2 is signal_declaration
       limit cased_identifier to "^[a-z][a-z0-9_]*$"
end
```

The id1 identifier matches both templates, whereas the ID1 identifier only matches the first template. The identifier attribute is not case-sensitive because VHDL is not case-sensitive.

Use the one_declaration_per_line attribute to force a single line for each declaration.

Use the comment attribute to force a comment on the same line as the declaration.

Simple Name Template

The simple_name template is a primary template belonging to the EXPRESSION, NAME, and TARGET classes. It contains the attributes shown in **Table 91.**

Table 91: simple_name Template

Attribute	Kind	Limit_Kind
cased_identifier	template	ID
identifier	template	ID
object_definition	template	OBJECT_ITEM
flipflop	template	flipflop
latch	template	latch

Use the object_definition attribute to control the type of definition associated with the simple_name. For example, to focus just on ports in a given expression, you could write the following template:

```
template CLK_NAME is simple_name
       limit object_definition to port_declaration
end
```

There are some application-specific attributes associated with this template that make it easier to write commonly-used rules. For example:

- Use the flipflop attribute to control whether or not the object infers a flipflop.
- Use the latch attribute to control whether or not the object infers a latch.

Use the cased_identifier attribute just like the identifier attribute, but in cases where the identifier is case-sensitive. Consider the following templates:

```
template T1 is simple_name
       limit identifier to "^[a-z][a-z0-9_]*$"
end
template T2 is simple_name
       limit cased_identifier to "^[a-z][a-z0-9_]*$"
end
```

The id1 identifier matches both templates, whereas the ID1 identifier only matches the first template. The identifier attribute is not case-sensitive because VHDL is not case-sensitive.

Slice Name Template

The LRM (§6.5) defines the grammar for slice name as follows:

```
slice_name ::= prefix ( discrete_range )
```

The slice_name template is a secondary template belonging to the EXPRESSION, NAME, and TARGET classes. It contains the attributes shown in **Table 92**.

Table 92: slice_name Template

Attribute	Kind	Limit_Kind
cased_identifier	template	ID
identifier	template	ID
object_definition	template	OBJECT_ITEM
range	template	range
range_overflow	local	N/A
flipflop	template	flipflop
latch	template	latch

Use the object_definition attribute to control the type of definition associated with the simple_name. For example, to focus just on ports in a given expression, you could write the following template:

```
template CLK_NAME is simple_name
       limit object_definition to port_declaration
end
```

There are some application-specific associated with this template that make it easier to write commonly-used rules. For example:

- Use the flipflop attribute to control whether or not the object infers a flipflop.
- Use the latch attribute to control whether or not the object infers a latch.

Use the cased_identifier attribute just like the identifier attribute, but in cases where the identifier is case-sensitive. Consider the following templates:

```
template T1 is slice_name
       limit identifier to "^[a-z][a-z0-9_]*$"
end
template T2 is slice_name
       limit cased_identifier to "^[a-z][a-z0-9_]*$"
end
```

The id1 identifier matches both templates, whereas the ID1 identifier only matches the first template. The identifier attribute is not case-sensitive because VHDL is not case-sensitive.

Statement Format Template

Use the statement_format template to control issues relating to compound statements; that is, statements that contain other statements.

The statement_format template is a secondary template belonging to no classes. It contains the attributes shown in **Table 93**.

Table 93: statement_format Template

Attribute	Kind	Limit_Kind
end_label	local	N/A
one_statement_per_line	local	N/A
indentation_depth	max/min	N/A
line_count	max/min	N/A

Use the end_label attribute to force the presence of an ending label. Consider the following rule:

```
template FEL is statement_format
       force end_label
end
limit statement_format in subprogram_body to FEL
```

With this rule activated, the VHDL code must contain the optional subprogram_kind clause at the end of the subprogram statement. This means that:

```
procedure P is
begin
end procedure P;
```

is acceptable, whereas:

```
procedure P is
begin
end;
```

generates an error.

Use the indentation_depth attribute to control the maximum or minimum number of indentation levels within a compound statement.

Use the line_count attribute to control the maximum or minimum number of lines of code within a compound statement.

Use the one_statement_per_line attribute to force a single line for each declaration.

Subprogram Body Template

The LRM (§2.2) defines the grammar for subprogram body as follows:

```
subprogram_body ::=
       subprogram_specification is
             subprogram_declarative_part
       begin
             subprogram_statement_part
       end [ subprogram_kind ] [ designator ] ;

subprogram_declarative_part ::=
       { subprogram_declarative_item }

subprogram_declarative_item ::=
          subprogram_declaration
       | subprogram_body
       | type_declaration
       | subtype_declaration
       | constant_declaration
       | variable_declaration
       | file_declaration
       | alias_declaration
       | attribute_declaration
       | attribute_specification
       | use_clause
       | group_template_declaration
```

```
        | group declaration

subprogram_statement_part ::=
        { sequential_statement }
        { subprogram_sequential_statement }

subprogram_kind ::=  procedure | function
```

The subprogram_body template is a primary template belonging to the REGION_PART class. It contains the attributes shown in Table 94.

Table 94: subprogram_body Template

Attribute	Kind	Limit_Kind
subprogram_declaration	template	subprogram_declaration
subprogram_body	template	subprogram_body
type_declaration	template	type_declaration
subtype_declaration	template	subtype_declaration
constant_declaration	template	constant_declaration
variable_declaration	template	variable_declaration
file_declaration	template	file_declaration
fully_assign_variables	local	N/A
alias_declaration	template	alias_declaration
attribute_declaration	template	attribute_declaration
attribute_specification	template	attribute_specification
use_clause	template	use_clause
group_template_declaration	template	group_template_declaration
group_declaration	template	group_declaration
assertion_statement	template	assertion_statement
report_statement	template	report_statement
procedure_call_statement	template	procedure_call_statement
if_statement	template	if_statement
signal_assignment_statement	template	signal_assignment_statement
variable_assignment_statement	template	variable_assignment_statement

Table 94: subprogram_body Template (Continued)

Attribute	Kind	Limit_Kind
case_statement	template	case_statement
loop_statement	template	loop_statement
for_loop_statement	template	for_loop_statement
while_loop_statement	template	while_loop_statement
next_statement	template	next_statement
exit_statement	template	exit_statement
return_statement	template	return_statement
wait_statement	template	wait_statement
null_statement	local	N/A
header_comment	template	header_comment
statement_format	template	statement_format
out_params_fully_assigned	local	N/A
dead_code	local	N/A
return_last	local	N/A
recursion_type	set	N/A
clock_expression_count	max/min	N/A
clock_signal	max/min	N/A
statement_profile_s	aggregate	N/A
return_fully_assigned	local	N/A

There are some application-specific attributes associated with this template that make it easier to write commonly-used rules. For example:

- Use the header_comment attribute to control keywords in comments at the start of units or subprograms.
- Use the statement_format attribute to handle organization issues within compound statements.
- Use the clock_expression_count attribute to control the number of different clock expressions allowed in a subprogram (procedure) body.

- Use the clock_signal attribute to control the number of different clock signals allowed in a subprogram (procedure) body.
- Use the out_params_fully_assigned attribute to ensure that every parameter of mode “out” is assigned on every flow of execution within the subprogram. For example:

```
force out_params_fully_assigned in subprogram_body
```

- Use the return_last attribute to force the last statement of a function to be a return statement. For example, if you want functions to contain only one return statement as the last statement in the function, you could write the following template for a rule:

```
template  ONE-RETURN is subprogram_body
    force return_last
    max   return_statement is 1
end
```

- Use the recursion_type attribute to specify different types of recursion that are allowed in subprograms. This attribute points to an element of the primitive type rsp_Recursion_Type. For example:

```
set recursion_type in subprogram_body to static_recursion
severity ERROR
```

  This rule forces the Leda Checker to test each subprogram. If a recursive call is found enclosed within a loop having non-static bounds, Leda flags an error.

- Use the fully_assign_variables attribute to ensure that all variables (inside a subprogram or a sequential block) are assigned on every possible flow of control through the sequential block. For example:

```
force fully_assign_variables in subprogram_body

--Fail case
function test --FAIL
(D_IN : in std_logic;
 RST  : in std_logic)
return std_logic is variable O : std_logic;
begin
    if (RST = '1') then
          O := D_IN;
    end if;
    return O;
end test;

force fully_assign_variables in subprogram_body
message "Assignment to variables missing inside subprogram"
severity WARNING
```

- Use the out_params_fully_assigned attribute to ensure that all subprogram parameters of mode out or inout (VHDL) are assigned on every possible flow of control through the subprogram. For example:

```
force out_params_fully_assigned in subprogram_body

--Fail case
procedure test --FAIL
(D_IN  : in std_logic;
 RST   : in std_logic;
 D_OUT : out std_logic) is
begin
    if (RST = '1') then
          D_OUT := D_IN;
    end if;
end test;

force out_params_fully_assigned in subprogram_body
message "Assignment to out or inout parameters missing inside
subprogram"
severity WARNING
```

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

```
force return_fully_assigned in subprogram_body

function dlatch --FAIL
(D  : in std_logic_vector(3 downto 0);
 G  : in std_logic;
 RST   : in std_logic)
return std_logic_vector is
variable ret : std_logic_vector(3 downto 0);
begin
    if (RST = '1') then
          return "0000";
    elsif (G = '1') then
          return D;
    end if;
end dlatch;

force return_fully_assigned in subprogram_body
message "No value returned for a particular flow through the
function"
severity WARNING
```

Subprogram Declaration Template

The LRM (§2.1) defines the grammar for subprogram declaration as follows:

```
subprogram_declaration ::=
       subprogram_specification ;
subprogram_specification ::=
            procedure designator [ ( formal_parameter_list ) ]
       | [ pure | impure ]  function designator [ (
formal_parameter_list ) ]
                    return type_mark
designator ::=  identifier  |  operator_symbol
formal_parameter_list ::=  parameter_interface_list
```

The block_specification template is a primary template belonging to the REGION_PART and OBJECT_ITEM classes. It contains the attributes shown in Table 95.

Table 95: subprogram_declaration Template

Attribute	Kind	Limit_Kind
cased_identifier	template	ID
identifier	template	ID
formal_parameter	template	formal_parameter
type_mark	template	TYPE_MARK
subprogram_body	template	subprogram_body
declarative_region	template	REGION_PART
operator_symbol	local	N/A
procedure	local	N/A
function	local	N/A
impure	local	N/A
pure	local	N/A
header_comment	template	ID
parameter_count	max/min	N/A

Use the declarative_region attribute to point to a template from the class REGION_PART that constrains the enclosing region of the current template. For example:

```
limit declarative_part in subprogram_declaration to
       package_declaration severity error
```

means that subprogram declarations can only appear in package declaration regions.

Use the operator_symbol attribute to force or forbid the use of operator symbols as subprogram names. For example:

```
no operator_symbol in subprogram_declaration severity error
```

Use the procedure and function attributes to force or forbid the presence of procedure or function subprograms. For example, the following command:

```
no procedure in subprogram_declaration severity error
```

means that procedure declarations are not allowed.

There are some application-specific attributes associated with this template that make it easier to write commonly-used rules. For example:

- Use the parameter_count attribute to set the maximum or minimum number of parameters allowed.
- Use the header_comment attribute to control the keywords in comments at the start of units or subprograms.

Use the cased_identifier attribute just like the identifier attribute, but in cases where the identifier is case-sensitive. Consider the following templates:

```
template T1 is subprogram_declaration
       limit identifier to "^[a-z][a-z0-9_]*$"
end
template T2 is subprogram_declaration
       limit cased_identifier to "^[a-z][a-z0-9_]*$"
end
```

The id1 identifier matches both templates, whereas the ID1 identifier only matches the first template. The identifier attribute is not case-sensitive because VHDL is not case-sensitive.

Subtype Declaration Template

The LRM (§4.2) defines the grammar for subtype declaration as follows:

```
subtype_declaration ::= subtype identifier is subtype_indication ;
```

The subtype_declaration template is a primary template belonging to the TYPE_MARK class. It contains the attributes shown in **Table 96**.

Table 96: subtype_declaration Template

Attribute	Kind	Limit_Kind
cased_identifier	template	ID
identifier	template	ID
subtype_indication	template	subtype_indication
declarative_region	template	REGION_PART

Use the cased_identifier attribute just like the identifier attribute, but in cases where the identifier is case-sensitive. Consider the following templates:

```
template T1 is subtype_declaration
       limit identifier to "^[a-z][a-z0-9_]*$"
end
template T2 is subtype_declaration
       limit cased_identifier to "^[a-z][a-z0-9_]*$"
end
```

The id1 identifier matches both templates, whereas the ID1 identifier only matches the first template. The identifier attribute is not case-sensitive because VHDL is not case-sensitive.

Subtype Indication Template

The LRM (§4.2) defines the grammar for subtype indication as follows:

```
subtype_indication ::=
       [ resolution_function_name ] type_mark [ constraint ]
type_mark ::=
          type_name
       | subtype_name
constraint ::=
          range_constraint
       | index_constraint
```

The subtype_indication template is a primary template belonging to the DISCRETE_RANGE class. It contains the attributes shown in **Table 97**.

Table 97: subtype_indication Template

Attribute	Kind	Limit_Kind
type_mark	template	TYPE_MARK
subtype_indication	template	subtype_indication
range	template	range
resolution_function	local	N/A
evaluation_time	set	N/A

Use the subtype_indication and range attributes to constrain the subtype.

Use the resolution_function attribute to force or forbid the presence of resolution function names in subtype indications.

Synchronous Initialization Template

The synchronous_initialization template is a primary template belonging to no classes. It contains the attributes shown in **Table 98**.

Table 98: synchronous_initialization Template

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

Leda recognizes all RTL reset expressions. Use the expression attribute to limit the expression type.

Use the object_definition attribute to limit the type of the reset signal.

Use the cased_identifier attribute just like the identifier attribute, but in cases where the identifier is case-sensitive. Consider the following templates:

```
template T1 is synchronous_initialization
       limit identifier to "^[a-z][a-z0-9_]*$"
end
template T2 is synchronous_initialization
       limit cased_identifier to "^[a-z][a-z0-9_]*$"
end
```

The id1 identifier matches both templates, whereas the ID1 identifier only matches the first template. The identifier attribute is not case-sensitive because VHDL is not case-sensitive.

Note

When Leda encounters a complex expression (containing more than one object) used as a clock or reset, naming convention rules are automatically deactivated. This is because Leda assumes that the clock or reset is the output of a logic gate inferred by the complex expression and not simply one of the objects in the expression.

Type Declaration Template

The LRM (§4.1) defines the grammar for type declaration as follows:

```
type_declaration ::=
          full_type_declaration
       | incomplete_type_declaration
full_type_declaration ::=
       type identifier is type_definition ;
type_definition ::=
          scalar_type_definition
       | composite_type_definition
       | access_type_definition
       | file_type_definition
```

The type_declaration template is a primary template belonging to the TYPE_MARK class. It contains the attributes shown in **Table 99**.

Table 99: type_declaration Template

Attribute	Kind	Limit_Kind
cased_identifier	template	ID
identifier	template	ID
type_definition	template	TYPE_DEFINITION
declarative_region	template	REGION_PART
incomplete_type_declaration	local	N/A

Use the incomplete_type_declaration attribute to force or forbid the presence of incomplete type declarations.

Use the cased_identifier attribute just like the identifier attribute, but in cases where the identifier is case-sensitive. Consider the following templates:

```
template T1 is type_declaration
       limit identifier to "^[a-z][a-z0-9_]*$"
end
template T2 is type_declaration
       limit cased_identifier to "^[a-z][a-z0-9_]*$"
end
```

The id1 identifier matches both templates, whereas the ID1 identifier only matches the first template. The identifier attribute is not case-sensitive because VHDL is not case-sensitive.

Unconstrained Array Definition Template

The LRM (§3.2.1) defines the grammar for unconstrained array definition as follows:

```
unconstrained_array_definition ::=
       array ( index_subtype_definition { , index_subtype_definition } )
             of element_subtype_indication
```

The unconstrained_array_definition template is a primary template belonging to no classes. It contains the attributes shown in **Table 100**.

Table 100: unconstrained_array_definition Template

Attribute	Kind	Limit_Kind
subtype_indication	template	subtype_indication
dimension_count	max/min	N/A
subtype_definition_s	aggregate	subtype_indication

Use the dimension_count attribute to set the maximum or minimum number of dimensions allowed. For example, the command:

```
max dimension_count in constrained_array_definition is 1
```

means that only 1-dimensional arrays are allowed.

To allow more than one dimension, put different constraints on the subtype definitions for each dimension. Each aggregate must point to a set of index_subtype_definition templates.

Use Clause Template

The LRM (§10.4) defines the grammar for use clause as follows:

```
use_clause ::= use selected_name { , selected_name } ;
```

The use_clause template is a primary template belonging to no classes. It contains the attribute shown in **Table 101**.

Table 101: use_clause Template

Attribute	Kind	Limit_Kind
selected_name	template	selected_name

Variable Assignment Statement Template

The LRM (§8.5) defines the grammar for variable assignment statement as follows:

```
variable_assignment_statement ::= [label : ] target :=  expression ;
```

The variable_assignment_statement template is a primary template belonging to the SEQUENTIAL_STATEMENT class. It contains the attributes shown in **Table 102**.

Table 102: variable_assignment_statement Template

Attribute	Kind	Limit_Kind
cased_label	template	ID
label	template	ID
operand_size_mismatch	local	N/A
range_overflow	local	N/A
target	template	TARGET
expression	template	EXPRESSION
declarative_region	template	REGION_PART
read_write	local	N/A

Use the cased_label attribute just like the label attribute, but in cases where the label is case-sensitive. Consider the following templates:

```
template T1 is variable_assignment_statement
       limit label to "^[a-z][a-z0-9_]*$"
end
template T2 is variable_assignment_statement
       limit cased_label to "^[a-z][a-z0-9_]*$"
end
```

The lab1 label matches both templates, whereas the LAB1 label only matches the first template. The label attribute is not case-sensitive because VHDL is not case-sensitive.

Variable Declaration Template

The LRM (§4.3.1.3) defines the grammar for variable declaration as follows:

```
variable_declaration ::=
   [ shared ] variable identifier_list:subtype_indication [:= expression ]
```

The variable_declaration template is a primary template belonging to the OBJECT_ITEM class. It contains the attributes shown in **Table 103**.

Table 103: variable_declaration Template

Attribute	Kind	Limit_Kind
cased_identifier	template	ID
identifier	template	ID
subtype_indication	template	subtype_indication
default	template	EXPRESSION
declarative_region	template	REGION_PART
one_declaration_per_line	local	N/A
comment	local	N/A

Use cased_identifier attribute just like the identifier attribute, but in cases where the identifier is case-sensitive. Consider the following templates:

```
template T1 is variable_declaration
       limit identifier to "^[a-z][a-z0-9_]*$"
end
template T2 is variable_declaration
       limit cased_identifier to "^[a-z][a-z0-9_]*$"
end
```

The id1 identifier matches both templates, whereas the ID1 identifier only matches the first template. The identifier attribute is not case-sensitive because VHDL is not case-sensitive.

Use the one_declaration_per_line attribute to force a single line for each declaration.

Use the comment attribute to force a comment on the same line as the declaration.

Wait Statement Template

The LRM (§8.1) defines the grammar for wait statement as follows:

```
wait_statement ::=
[ label : ] wait [ sensitivity_clause ] [ condition_clause ]
       [timeout_clause ] ;
sensitivity_clause ::=  on sensitivity_list
sensitivity_list ::=  signal_name  { , signal_name  }
condition_clause ::=  until condition
condition ::=  boolean_expression
timeout_clause ::=  for time_expression
```

The wait_statement template is a primary template belonging to the SEQUENTIAL_STATEMENT class. It contains the attributes shown in **Table 104**.

Table 104: wait_statement Template

Attribute	Kind	Limit_Kind
cased_label	template	ID
label	template	ID
sensitivity	template	EXPRESSION
condition	template	EXPRESSION
timeout	template	EXPRESSION
declarative_region	template	REGION_PART
sensitivity_count	max/min	N/A

Use the cased_label attribute just like the label attribute, but in cases where the label is case-sensitive. Consider the following templates:

```
template T1 is wait_statement
       limit label to "^[a-z][a-z0-9_]*$"
end
template T2 is wait_statement
       limit cased_label to "^[a-z][a-z0-9_]*$"
end
```

The lab1 label matches both templates, whereas the LAB1 label only matches the first template. The label attribute is not case-sensitive because VHDL is not case-sensitive.

Waveform Element Template

The LRM (§8.4.1) defines the grammar for waveform element as follows:

```
waveform_element ::=
        value_expression [ after time_expression ]
       | null [ after time_expression ]
```

The waveform_element template is a secondary template belonging to no classes. It contains the attributes shown in **Table 105**. This template is used in the Conditional Waveforms Template, Selected Waveforms Template, and Waveform Template**.**

Table 105: waveform_element Template

Attribute	Kind	Limit_Kind
waveform_expression	template	EXPRESSION
after_expression	template	EXPRESSION
null	local	N/A

Waveform Template

The LRM (§8.4) defines the grammar for waveform as follows:

```
waveform ::= waveform_element { , waveform_element } | unaffected
```

The waveform template is a secondary template belonging to no classes. It contains the attributes shown in **Table 106**. This template is used in the Signal Assignment Statement Template.

Table 106: waveform Template

Attribute	Kind	Limit_Kind
unaffected	local	N/A
waveform_element_s	aggregate	waveform_element
waveform_element_count	max/min	N/A

Use the unaffected attribute to force or forbid the presence of the unaffected VHDL keyword.

While Loop Statement Template

The LRM (§8.9) defines the grammar for loop statement as follows:

```
loop_statement ::=
        [ loop_label : ]
              [ iteration_scheme ] loop
                    sequence_of_statements
              end loop [ loop_label ] ;
iteration_scheme ::=
           while condition
        | for loop_parameter_specification
parameter_specification ::=
        identifier in discrete_range
```

The while_loop_statement template is a primary template belonging to the SEQUENTIAL_STATEMENT class. It contains the attributes shown in **Table 107**.

Table 107: while_loop_statement Template

Attribute	Kind	Limit_Kind
cased_label	template	ID
label	template	ID
condition	template	EXPRESSION

Table 107: while_loop_statement Template (Continued)

Attribute	Kind	Limit_Kind
assertion_statement	template	assertion_statement
report_statement	template	report_statement
procedure_call_statement	template	procedure_call_statement
if_statement	template	if_statement
signal_assignment_statement	template	signal_assignment_statement
variable_assignment_statement	template	variable_assignment_statement
case_statement	template	case_statement
loop_statement	template	loop_statement
for_loop_statement	template	for_loop_statement
while_loop_statement	template	while_loop_statement
next_statement	template	next_statement
exit_statement	template	exit_statement
return_statement	template	return_statement
wait_statement	template	wait_statement
null_statement	local	N/A
declarative_region	template	REGION_PART
statement_format	template	statement_format
statement_profile_s	aggregate	SEQUENTIAL_STATEMENT
evaluation_time	set	N/A

Use the cased_label attribute just like the label attribute, but in cases where the label is case-sensitive. Consider the following templates:

```
template T1 is while_loop_statement
       limit label to "^[a-z][a-z0-9_]*$"
end
template T2 is while_loop_statement
       limit cased_label to "^[a-z][a-z0-9_]*$"
end
```

The lab1 label matches both templates, whereas the LAB1 label only matches the first template. The label attribute is not case-sensitive because VHDL is not case-sensitive.

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 VRSL 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 VRSL 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 reference information about that attribute in this manual. A traditional index is also included at the end of the manual.

TEMPLATE x Attribute

A

Alias_declaration template

Allocator template

Architecture_body template

Assertion_statement template

Association_element template

Association_list template

Asynchronous_initialization template

Attribute_declaration template

Attribute_name template

Attribute_specification template

B

Binary_operation template

Binding_indication template

Block_configuration template

Block_specification template

Block_statement template

C

Case_statement template

Clock template

Comment template

Component_configuration template

Component_declaration template

Component_instantiation_statement template

Component_specification template

Concurrent_assertion_statement template

Concurrent_procedure_call_statement template

Conditional_signal_assignment template

Conditional_waveform template

Configuration_declaration template

Configuration_specification template

Connectivity_path template

Constant_declaration template

Constrained_array_definition template

D

Data_signal template

Design template

Design_unit template

Disconnection_specification template

E

Element_association template

Element_declaration template

Entity_declaration template

Enumeration_type_definition template

Exit_statement template

Expression template

F

File_declaration template

File_layout template

File_type_definition template

Flipflop template

Floating_type_definition template

For_loop_statement template

Formal_parameter template

G

Generic_declaration template

H

Header_comment template

I

Identifier template

If_statement template

Indexed_name template

Integer_type_definition template

Interface_file_declaration template

Interface_variable_declaration template

L

Latch template

Literal template

Logic_cost template

Loop_statement template

N

Next_statement template

P

Package_body template

Package_declaration template

Physical_type_definition template

Port_declaration template

Procedure_call_statement template

Process_statement template

R

Range template

Record_type_definition template

Report_statement template

Return_statement template

S

Selected_name template

Selected_signal _assignment template

Selected_waveforms template

Shared_variable_declaration template

Signal_assignment_statement template

Signal_declaration template

Simple_name template

Slice_name template

Statement_format template

Subprogram_body template

Subprogram_declaration template

Subtype_declaration template

Subtype_indication template

T

U

V

W

Waveform template

Waveform_element template

While_loop_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 VRSL 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 VRSL 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 reference information about that attribute in this manual. A traditional index is also included at the end of the manual.

ATTRIBUTE x Template

A

and_cost attribute

architecture attribute

architecture_name attribute

assertion_statement attribute

association_element_s attribute

asynchronous_feedback attribute

asynchronous_initialization attribute

asynchronous_logic attribute
Design template 58

asynchronous_reset attribute
Process_statement template 157

asynchronous_reset_signal attribute
Architecture_body template 75

attribute_declaration attribute
Architecture_body template 74
Block_statement template 90
Entity_declaration template 112
Generate_statement template 128
Package_declaration template 149
Process_statement template 156
Subprogram_body template 174

attribute_designator attribute
Attribute_name template 82
Attribute_specification template 83

attribute_specification attribute
Architecture_body template 74
Block_statement template 90
Configuration_declaration template 104
Entity_declaration template 112
Generate_statement template 129
Package_declaration template 149
Process_statement template 156
Subprogram_body template 174

B

C

cased_label attribute

character_count attribute

character_literal attribute

characters_per_line attribute
File_layout template 120

choice attribute
Aggregate template 70
Case_statement template 91
Element_association template 109
Selected_waveforms template 165

clock attribute
Design template 58
Flipflop template 61, 122
Latch template 62, 140

clock_count attribute
Design template 59

clock_expression_count attribute
Process_statement template 157
Subprogram_body template 175

clock_signal attribute
Architecture_body template 75
Process_statement template 157
Subprogram_body template 175

comb_cost attribute
Design template 59

combinatorial attribute
Process_statement template 157

comment attribute

comment_line attribute

comparator_cost attribute

complete_sensitivity attribute

component attribute

component_configuration attribute

component_declaration attribute

conditional_signal_assignment attribute

conditional_waveforms_s attribute

configuration attribute

configuration_specification attribute

connections attribute

connectivity_path attribute

consistent_port_order attribute

constant attribute

D

declaration_profile_s attribute

Block_statement template 90
Entity_declaration template 113
Package declaration template 149
Package_declaration template 149

declarative_region attribute

Alias_declaration template 71
Assertion_statement template 77
Attribute_declaration template 81
Block_statement template 90
Case_statement template 92
Component_declaration template 96
Component_instantiation_statement template 97
Concurrent_assertion _statement template 100
Concurrent_procedure_call_statement template 100
Constant_declaration template 107
Exit_statement template 116
File_declaration template 119
For_loop_statement template 124
Generate_statement template 129
Generic_declaration template 130
If_statement template 135
Interface_file_declaration template 138
Interface_variable_declaration template 139
Loop_statement template 144
Next_statement template 145
Port_declaration template 151
Procedure_call_statement template 154
Process_statement template 157
Report_statement template 161
Return_statement template 162

decoder_cost attribute

default attribute

deferred attribute

design_unit attribute

E

entity_aspect_name attribute
Binding_indication template 86

entity_declaration attribute
Architecture_body template 75

entity_designator attribute
Attribute_specification template 83

error_id attribute
Identifier template 133

evaluation_time attribute
Binary_operation template 85
Expression template 117
For_loop_statement template 124
Literal template 141
Range template 159
While_loop_statement template 190

exit_statement attribute
Case_statement template 92
For_loop_statement template 123
If_statement template 134
Loop_statement template 143
Process_statement template 156
Subprogram_body template 175
While_loop_statement template 190

expression attribute
Allocator template 72
Asynchronous_initialization template 80

F

file attribute

file_declaration attribute

file_layout attribute

file_length attribute

file_length_count attribute

file_name attribute

fixed_value attribute

flipflop attribute

G

H

I

identifier attribute

if_statement attribute

impure attribute

in attribute

incomplete_type_declaration attribute

indentation_depth attribute

L

latch attribute

latch_as_source attribute

M

N

O

one_declaration_per_line attribute

one_statement_per_line attribute

open attribute

open_kind attribute

operand attribute

operand_size_match attribute

operand_size_mismatch attribute

operator_symbol attribute

or_cost attribute

others attribute

P

port_declaration attribute

port_map_aspect attribute

port_order attribute

positional_association attribute

postponed attribute

power_cost attribute

Q

R

remainder_cost attribute
Logic_cost template 63

report_expression attribute
Assertion_statement template 77
Concurrent_assertion_statement template 99
Report_statement template 161

report_statement attribute
Case_statement template 91
For_loop_statement template 123
If_statement template 134
Loop_statement template 143
Process_statement template 156
Subprogram_body template 174
While_loop_statement template 190

reset_at_hierarchical_boundary attribute
Logic_cost template 63

reset_count attribute
Design template 59

resolution_function attribute
Subtype_indication template 181

return_last attribute
Subprogram_body template 175

return_statement attribute
Case_statement template 92
For_loop_statement template 123

right_expression attribute

S

selected_name attribute

selected_signal_assignment attribute

selected_waveform_s attribute

sensitivity attribute

sensitivity_count attribute

set_count attribute

severity_expression attribute

shared_variable_declaration attribute

shift_cost attribute

signal attribute

signal_assignment_statement attribute

signal_declaration attribute

signals_driven attribute

starting_unit attribute

state_count attribute

state_variable attribute

statement_format attribute

statement_profile_s attribute

subprogram_body attribute

subprogram_declaration attribute

subtype attribute

subtype_declaration attribute

subtype_definition attribute

subtype_indication attribute

sync_ff_count attribute

synchronous_initialization attribute

synchronous_reset attribute

synchronous_reset_signal attribute

T

target attribute

text attribute

U

V

W

waveform attribute

waveform_count attribute

waveform_element_count attribute

waveform_element_s attribute

waveform_expression attribute

while_loop_statement attribute

within_same_clkdomain attribute

X

xnor_cost attribute

xor_cost attribute

Index

E

F

G

H

I

L

M

N

O

P

R

S

T

U

V

W