SYNOPSYS®

ProVHDL

Rule Specifier

LEDA 3.0

Tutorial

Copyright © 2001 by Synopsys, Inc. All rights reserved.

SYNOPSYS, Inc
700 East Middlefield Road
Mountain View, CA 94043, USA

E-mail:leda-support@synopsys.com
Web: http://www.synopsys.com

This software and manual are furnished under a license agreement and may not be used or copied except in accordance with the terms of the agreement.
No part of this publication may be reproduced, transmitted, or translated, in any form or by any means, electronic, mechanical, manual, optical, or otherwise, without prior written consent of Synopsys.

The information in this manual is subject to change without notice and does not represent a commitment on the part of Synopsys.

Even though Synopsys has taken every effort in the preparation of this manual and the test of the software, Synopsys makes no warranty of any kind, either express or implied, with regards to this software and documentation, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose.

Synopsys acknowledge trademarks or registered trademarks of other organizations for their respective products and services.

Contents

Preface

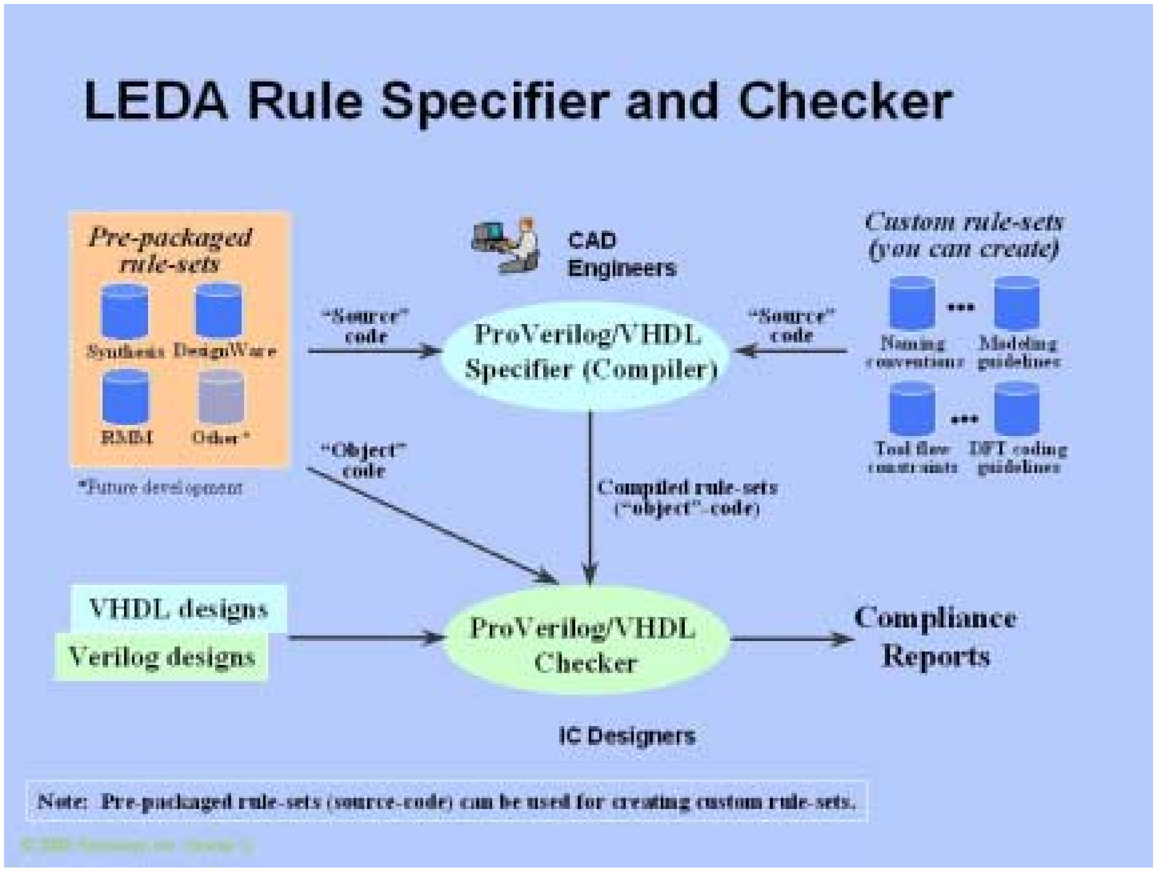
Welcome to the LEDA ProVHDL Rule Specifier Tutorial for the VHDL language. This tutorial is an example-based introduction to writing rules for checking VHDL code. The rules are written in the VHDL Rule Specification Language (VRSL).

If you have used VHDL before, the concepts in this tutorial will be familiar to you. Even if you are not familiar with VHDL, this tutorial is designed so you can get a basic understanding of VRSL's capabilities.

1. Introduction

The LEDA checkers come prepackaged with different rulesets and policies that check VHDL code for synthesizability, simulatability and portability, along with additional policies that check code for optimum performance.

Since ProVHDL is programmable, proprietary coding standards can also be readily created and/or modified to accommodate the unique requirements of design flows containing both commercially available and internally developed design tools.



ProVHDL consists of two tools: a rule specification tool and a rule checking tool. The specification tool uses a simple command set for creating and compiling custom rules. The Rule Specifier also features a user-friendly GUI for managing rulesets and coding standards.

The Rule Checker accepts VHDL code along with compiled rulesets and policies and outputs error messages, indicating which lines in the code violate different rules. The compiled rulesets either come pre-installed or are created using the Rule Specifer.

This tutorial will teach you to use the Rule Specifier to create and compile custom rules.

© Synopsys, Inc. 2001

2. Overview

ProVHDL works by enabling the designer to define "templates" and "rules" that jointly describe exactly what the input VHDL code should look like. Templates define "snapshots" of how the VHDL code should appear (e.g., what VHDL constructs should or should not be present, in what order, and so forth).

Rules constrain different VHDL constructs by ensuring that they correspond to acceptable values, ranges, or templates. The designer is thus able to define a syntactic/semantic "subset" of the overall VHDL code that is uniquely targeted to the design flow and methodology.

Rules can also be written to control the "hardware semantics" of VHDL. This means certain VHDL constructs infer specific hardware features if the description is synthesized (e.g., ck='1' and ck'event represent a clock active on the rising edge). These hardware features can be constrained through VRSL.

To implement checks for different coding style rules, the designer uses a simple yet powerful set of commands. Sequences of these commands (i.e., source code) are programmed using the ProVHDL Rule Specifier. This Rule Specifier then compiles the source code into object files used to configure one or multiple ProVHDL rule checkers.

Finally, the Rule Checker compares the input VHDL code to the templates and rules contained within the object files, and outputs error messages depending on whether the comparisons passed or failed.

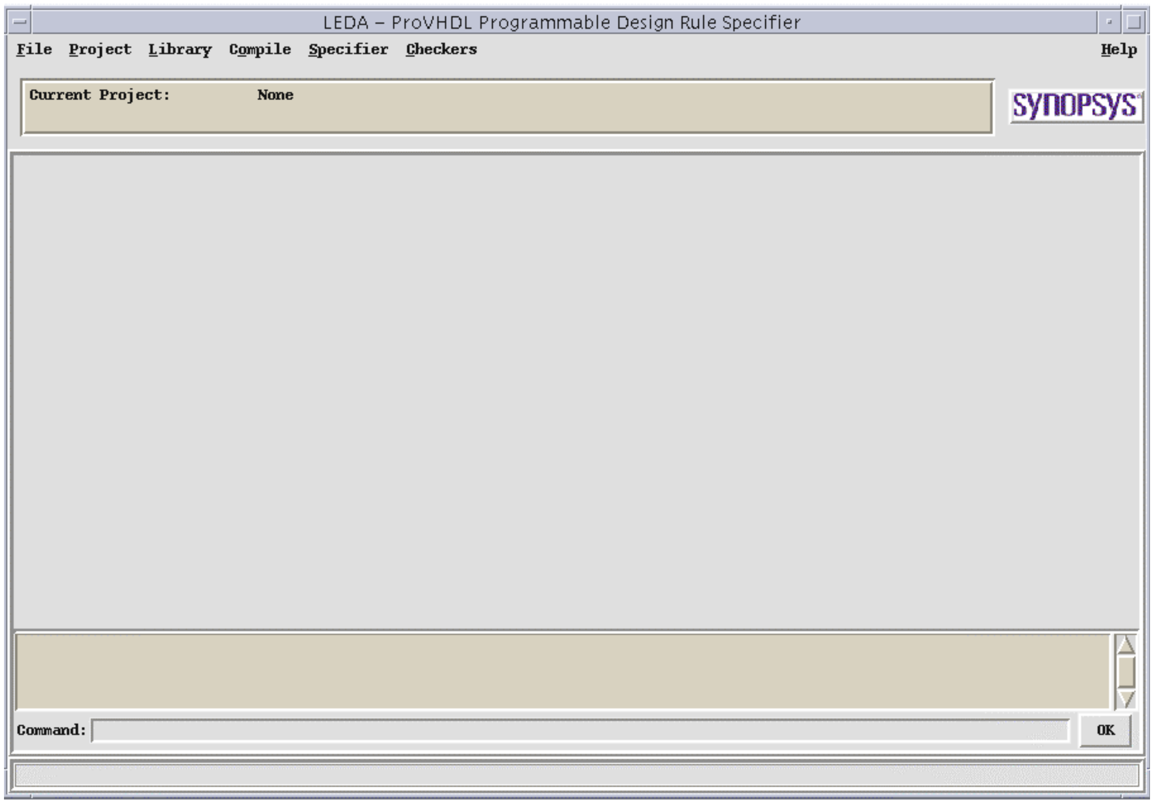
This tutorial will lead you through a series of examples where you will learn how to use VRSL to program custom coding guidelines. After completing all the examples, you will have a basic understanding of all the VRSL commands, keywords, and attributes. This will then enable you to independently develop and implement your own coding style rules.

3. First Exercise

The purpose of this first exercise is to familiarize you with the basic functions of the Rule Specifier and to create a rule in the VHDL Rule Specification Language (VRSL). The rule will then be used to check some simple VHDL code. You should be running the LEDA ProVHDL Rule Specifier which is invoked from a terminal window using the following UNIX command:

```
> provhdl_spec
```

The following screen should be showing:



In this exercise, we are first going to write a rule in VRSL. We will then write some VHDL code which the rule will check. Finally, we will use the LEDA tool to check the rule.

Before starting this tutorial, however, it is important to understand the distinction between VRSL and the VHDL language. VRSL is a macro-based language that is used to write rules that constrain the VHDL language. The VHDL language is one of two standard electronics industry hardware description languages. The other standard language is Verilog. This tutorial will teach you how to write rules in VRSL to constrain VHDL.

3.1 Creating a VRSL Ruleset file and Policy

Before we create the rule, some basic terminology will be helpful. The terms policy, ruleset, and rule are often used interchangeably. A policy can contain any number of rulesets. A ruleset can contain any number of rules.

To begin our exercise, we will create a policy with one ruleset file that will contain a single rule. This rule will constrain bidirectional ports in the VHDL code.

1 Using a text editor, type in the following text exactly as it is shown. You can use the text editor in the Specifier by choosing **File > New**. (Note that two dashes "--" at the beginning of a line designates a comment.)

```
  ruleset RULESET_1 is

  ------------------------------------------------------------

  ---------------------------------------------------
  -- Template Section
  ---------------------------------------------------

  template PORT_DEC is port_declaration
  no inout
  end

  ---------------------------------------------------
  -- Command section
  ---------------------------------------------------
  RULE1:
  limit port_declaration in entity_declaration to PORT_DEC
  message "Bi-directional ports are not allowed in the design"
  severity WARNING

end ruleset
```

2 Save this file as "ruleset.rl" making sure the file is in a convenient folder. Note that RULE1 (a single rule) is a subset of RULESET_1. Also note the ".rl" file extension which is the standard naming convention for VRSL ruleset files.

3 From the Specifier main window choose **Specifier > Specify Policies**.

4 Click in the **Set of Policies** pane then click on **New**. Type in a name for the new policy (e.g., "my_policy") and press **Enter**.

5 When the new policy appears in the left pane, click on it then click in the **Set of Rules** pane. Click on **Add**.

6 Click on the **File Name** ("ruleset.rl") for the ruleset you just created. Click on **Add** then close the open windows.

You have now created a policy ("my_policy") containing one ruleset. The ruleset contains one rule which will be used to check the VHDL code. You should see a *** Compile suc-

© *Synopsys, Inc. 2001*

cessfully *** message in the Result Frame of the main window. This message indicates that the rule is valid according to ProVHDL conventions.

3.2 Creating a Verilog file

To test this rule, we need to create some VHDL code to check it against. Since we just created a rule stating that bidirectional ports are not allowed in the design, we are going to purposely add bidirectional ports in our VHDL code (bolded below). When we check the code with our rule, we should see an error.

Using a text editor, type in the following text. Save it as "test.vhd" making sure the file is in a convenient folder. (Note the ".vhd" file extension for VHDL files and the two dash "--" comment line designation.)

```
-- VHDL test code

entity MUX is
 port (a,b,sel: in bit;
       z:       out bit := '1';
       k:       inout bit);
end MUX;

architecture describe of MUX is begin
  process begin
     if sel = '1' then
        z <= a;
     else
        z <=b;
     end if;
  end process;
end describe;
```

3.3 Creating a Project file

Before we can use the LEDA tool, we must first create a project file. A project file organizes the VHDL file(s) into easily managed units.

1 From the Specifier choose **Project > New > From Source Files**.

2 Enter the Project Name (e.g., “my_project”). A **Create Project from source files** window will appear.

3 Click on **Specify Source Files**. A **Specify Directories and/or Files** window will appear.

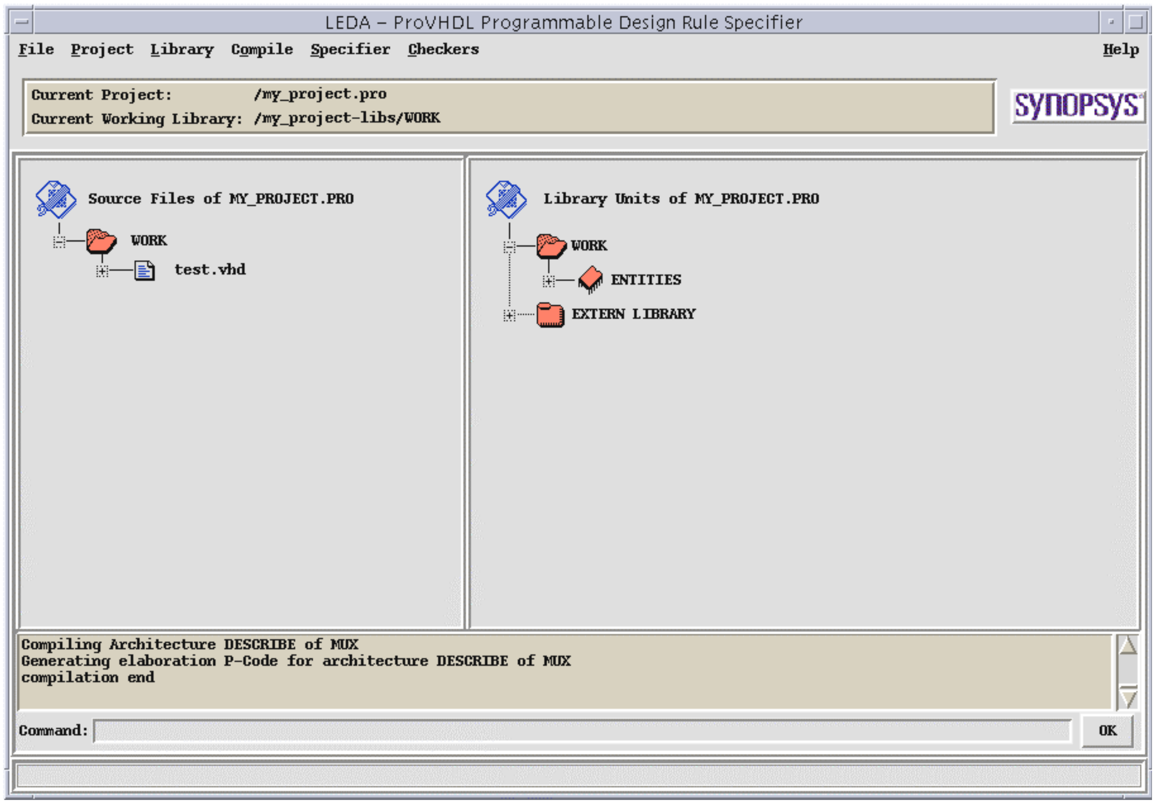
4 Click on **Add**.

5 Change the filter to *.

6 Select the “test.vhd” file. Click on **Add**.

7 Click on **Close**. The window will close.

8 Click on **Build and Compile**. You should see a “Building the Project” message and then another message ending with `*** compilation end ***`. This indicates the project is built. You should have the following screen.


LEDA – ProVHDL Programmable Design Rule Specifier
File
Project
Library
Compile
Specifier
Checkers
Help
SYNOPSYS
Source Files of MY_PROJECT.PRO
WORK
test.vhd
Library Units of MY_PROJECT.PRO
WORK
ENTITIES
EXTERN LIBRARY
Command:
OK

```
Current Project:          /my_project.pro
Current Working Library:  /my_project-libs/WORK
```

```
Compiling Architecture DESCRIBE of MUX
Generating elaboration P-Code for architecture DESCRIBE of MUX
compilation end
```

3.4 Running the Checker

With the project built, we will now set up and run the Checker which will check the VHDL code against the rule in our policy.

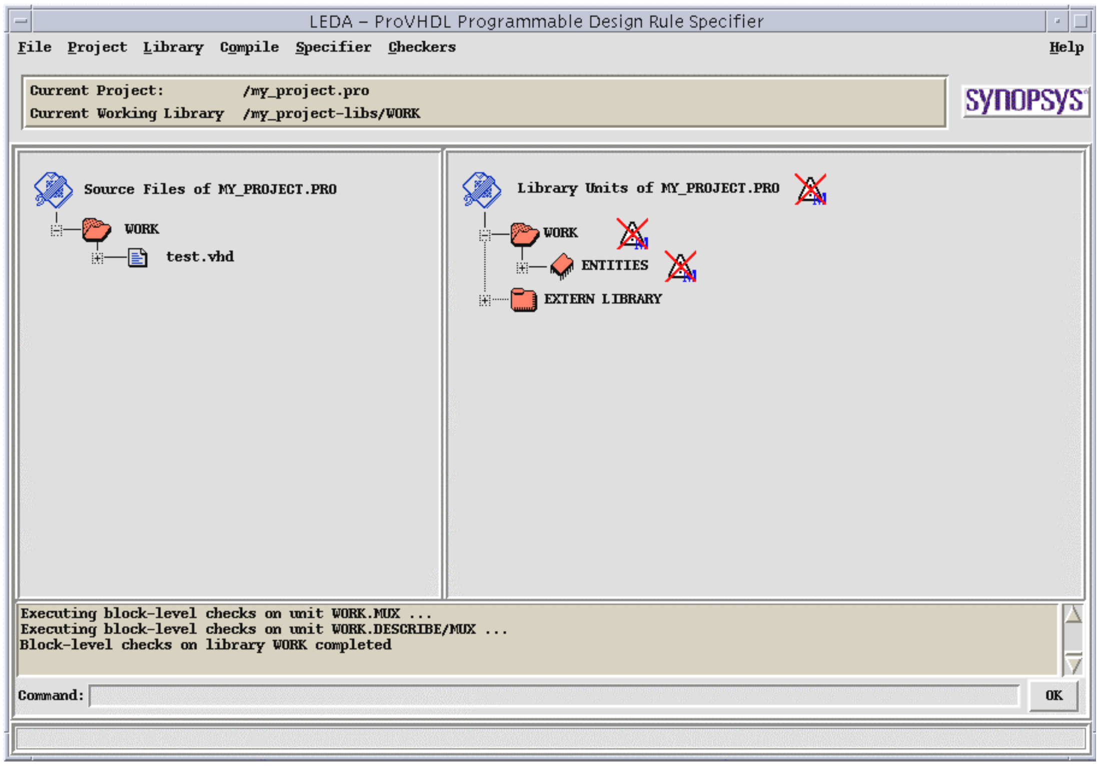
1 From the Specifier choose **Checkers > Select rules**.

2 Deselect all policies except “my_policy” by clicking any green boxes to gray. All boxes associated with “my_policy” should be green.

3 Click on **OK**.

4 From the Specifier choose **Checkers > Execute checkers**. You should see an hourglass icon on screen, indicating the files are being processed.

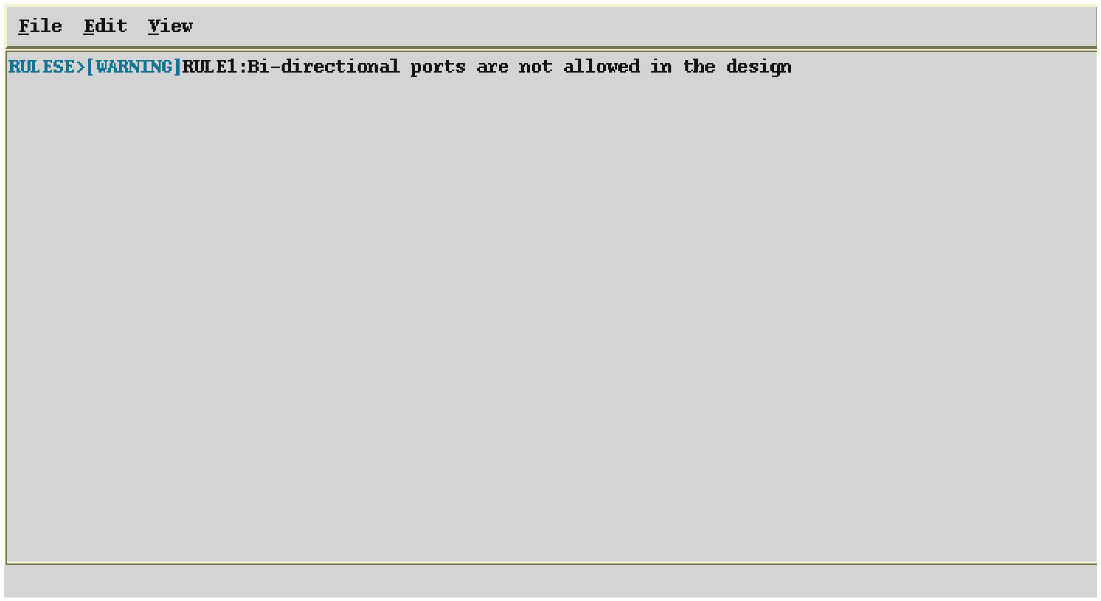
After execution, you should now see crossed-out (False) icons in the right pane beside WORK and MODULES which indicate the rule (no inout_declaration in module_declaration)was violated.


LEDA – ProVHDL Programmable Design Rule Specifier
File Project Library Compile Specifier Checkers Help
SYNOPSYS
Source Files of MY_PROJECT.PRO
WORK
test.vhd
Library Units of MY_PROJECT.PRO
WORK
ENTITIES
EXTERN LIBRARY
Command:
OK

```
Current Project:          /my_project.pro
Current Working Library   /my_project-libs/WORK
```

```
Executing block-level checks on unit WORK.MUX ...
Executing block-level checks on unit WORK.DESCRIBE/MUX ...
Block-level checks on library WORK completed
```

5 With the mouse pointer over one of the crossed-out icons, double-click to view the error results (image below).



```
RULESE>[WARNING]RULE1:Bi-directional ports are not allowed in the design
```

6 Left click on the **blue text** to view or edit the VHDL file ("test.vhd"). Right click on the **blue text** to view or edit the ruleset file ("ruleset.rl"). You can also use choose **View > Show HDL information** to see the errors listed with VHDL information. Left and right clicking the mouse over the **question marks** will give the same results as in the other view mode (Show only checker errors).To successfully compile the ruleset, you can delete the offending line of code in the VHDL file:

k: inout bit);

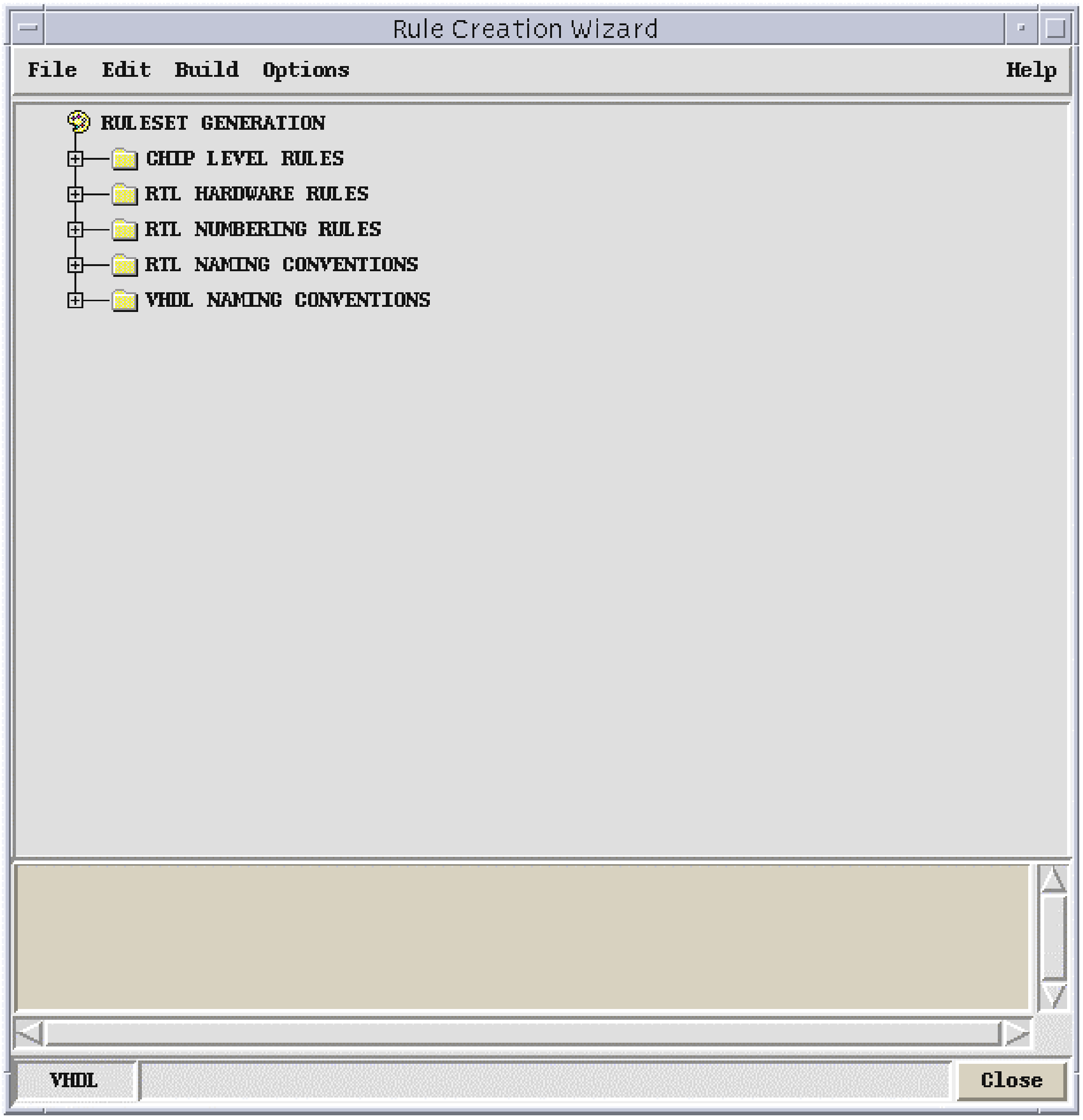
Exercise: *Try deleting the line above and recompiling. Experiment with different modifications to both the VRSL and VHDL code and look at the results after compilation.*

4. Rule Creation Wizard

We have just learned how to create a simple rule using a text editor. Some rules, however, are used frequently even though they may be configured differently depending upon company coding standards.

Rather than obliging rule-writers to write VRSL code to implement these common rules, a Rule Creation Wizard has been provided to allow these rules to be configured through a GUI. The code representing the rules is then automatically generated.

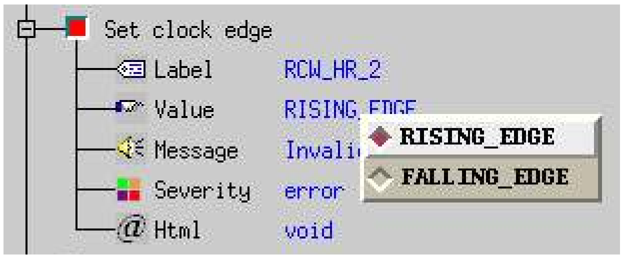
To activate the Rule Creation Wizard, choose **Specifier** > **Specify Policies**. The Policy Manager window will open. Click on **Wizard**. The following window will appear:



4.1 Using the Wizard

By clicking each of the yellow folders under the RULESET GENERATION tree, you can open up a ruleset and make appropriate selections for your policy. As an exercise, we will create a policy with a Set clock edge rule.

1. From the Wizard menu bar, choose **File > New ruleset**.
2. In the appropriate fields, enter the Policy name (e.g., "clock_policy") and the Ruleset name (e.g., "clock_rules"). The Ruleset filename will automatically be filled in with the Ruleset name prefix and the .rl extension. Click on OK.
3. Double-click on the **RTL HARDWARE RULES** folder .
4. Click on the gray icon beside Set clock edge. The icon should turn to red indicating the ruleset has been activated. Gray indicates the rule is deactivated.
5. Click on the + beside the Set clock edge icon to view the arguments for this rule.



Rules can have the following arguments:

Label: represents the label of the rule. To edit this label, click on the blue text.

Message: Message associated with the rule. To edit the message, click on blue text.

Severity: Severity of the rule (note, warning, error, and fatal). To change the severity, press right button on blue text to make severity menu appear and change the severity.

Html: Html address of html document associated to the rule. To edit the address, click on blue text.

Value: Some rules have may have a fixed value. To edit the value, click on blue text or press right button, depending on the rule.

6. For our example, we are going to change Severity to Warning (amber). Right-click over the **blue text** beside Severity, move the cursor down, and change error to warning. The Set clock edge icon should change from red to amber.
7. From the menu bar, select **Build > Build and Compile**. After a few moments, you should see the following message in the Result Frame of the main window:
   ```
   *** 1 VHDL rule created.
   ```
8. The Policy Manager window will appear and the new policy will be present in the Set of Policies pane.

Exercise: *Try adding more rules to your policy. Edit the rules with the text editor and with the mouse-click feature then recompile to see the results.*

5. Basics of Rule Creation

VRSL can be thought of as a meta or macro language with only six commands: FORCE, NO, LIMIT, SET, MAX, and MIN. Each command has a precise syntax with allowed keywords.

Two terms have specific definitions in VRSL: template and attribute. A template defines a "snapshot" of how the VHDL code should appear. An attribute represents the VHDL construct that is to be constrained.

5.1 The Six LEDA Commands (ProVHDL)

The following table lists the syntax for the LEDA commands in abbreviated form. There are exceptions to this syntax that will be discussed throughout the tutorial.

Command	Syntax	Description
FORCE	force A' [in A'']	Reports error if VHDL construct is absent.
`force default in constant_declaration`		
NO *(two versions)*	no A' [in A''] no A''' in A [of A]	Reports error if VHDL construct is present.
`no process_statement in entity_declaration`		
LIMIT	limit A [in A''] to L	Reports error if VHDL construct does not match one of a given set.
`limit clock to CLOCK_WITH_ID`		
SET	set X [in A''] to Y	Reports error if VHDL construct does not match fixed value.
`set edge in clock to rising`		
MIN	min M [in A] is N	Constrains the number of occurrences of VHDL construct to a minimum value.
`min low_bound is -2147483647`		
MAX	max M [in A] is N	Constrains the number of occurrences of VHDL construct to a maximum value.
`max dimension_count in unconstrained_array_definition is 1`		

where

A = primary template or secondary template | A' = A, local attribute, or aggregate attribute
A'' = A or all | A''' = list of templates
L = limit list | X = set_attribute| Y = STRING, number, or Enumerated type value
M = max_min_attribute | N = a number | [ ] denotes optional parameters

For an explanation of these terms, see the Appendix.

5.2 Commands and General Rule Writing Guidelines

The remainder of this tutorial is divided into the following command sections:

FORCE Command
NO Command
LIMIT Command
SET Command
MAX/MIN Command

Each section contains examples designed to give you an introduction to the basics of rule writing. The examples are chosen to give you a feel for VRSL's capabilities (what it can do) and the syntax of writing rules (how to do it). Examples denoted by a VHDL(e.g., Example 12: VHDL) include VHDL code to further demonstrate the use of the rule.

All terminology used for writing rules comes from either the IEEE Standard 1076-1993 VHDL Language Reference Manual (LRM) or the LEDA ProVHDL Rule Specificier User Manual (User Manual). The LRM is the basis upon which the User Manual was created. The User Manual also contains a double cross-referenced index of templates and attributes called the SpecDex which is designed to be used as an "online" reference (PDF format).

While a few of the terms used by LEDA are unique to LEDA, nearly all of the terms can be found in the LRM. If you have a User Manual or LRM available, try to look up these terms and become more familiar with them. This will help in your understanding of VRSL. The appendix in this tutorial has examples of some frequently-used templates and attributes taken from the User Manual.

Note: *Note that carriage returns have been inserted in the tutorial example messages to facilitate reading. When writing code, do not use carriage returns for messages; all message code lines must be continuous (i.e., no carriage returns inserted before the end).*

<u>Code as it is shown in this tutorial:</u>

```
message "Only Flip-flops with synchronous resets are allowed
          in the design according to current specs"
```

<u>Code as it should be written in VRSL:</u>

```
      message "Only Flip-flops with synchronous resets are allowed in the
design according to current specs"
```

5.3 FORCE Command

```
force A′ [in A′′]
```

where

A = primary template or secondary template
A’ = A, local attribute, or aggregate attribute
A” = A or all

The FORCE command can be thought of as a requirement that at least one occurrence of a particular construct be **present** in the VHDL code.

Example 1: FORCE command (rule basics)

As our first example of the FORCE command, we are going to create a rule that makes deferred constant declarations illegal. The following code shows how we can write this rule.

```
Example_1:
  force default in constant_declaration
  message "Deferred constant declarations are illegal"
  severity ERROR
```

In this example, `default` is an attribute within the context of `constant_declaration`. The attribute represents the VHDL clause that is to be constrained. The context indicates the circumstances under which the command (constraint) is to be applied.

Notice the message and severity lines in the code. The message line contains the text that will be displayed when the rule is violated. The severity line indicates the level of the violation (note, warning, error, and fatal). If these lines of code are not present, the rule will not be flagged and you will have no way of knowing that the rule has been violated. Therefore, omit the message and severity lines only under special circumstances.

Regarding terminology, the LRM defines constant declaration (LRM §4.3.1.1) as follows:

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

If you look in the User Manual, you will see the following definition for `constant_declaration:`

```
Primary template belonging to classes: OBJECT_ITEM

Attribute                            Kind        Limit_Kind

identifier                           template    ID
```

`subtype_indication`	`template`	`subtype_indication`
`default`	**`template`**	**`EXPRESSION`**
`declarative_region`	`template`	`REGION`
`deferred`	`local`	

Notice the occurrence of `default` (in bold) in the definition. Also notice that `constant_declaration` is a primary template in the User Manual.

Note: *Reference information will be provided for the first three examples in this tutorial. It is left as an exercise for you to use the LRM and the User Manual to find information for the remaining examples. The User Manual has a double cross-referenced index of templates and attributes called the **SpecDex**. Many novice users find this index, particularly the Attribute x Template index, helpful in understanding VRSL.*

Example 2: HTML error reporting

For our second example, we are going to write a rule requiring that process sensitivity lists be complete. Additionally, we will include some HTML code that will provide additional information about errors. The following code shows how we can write this rule.

```
Example_2:
  force complete_sensitivity in process_statement
  message "Missing or redundant signals in sensitivity list"
  severity ERROR
```

Sometimes we may need more explanation of a rule violation than the single-line error message that pops up on screen. This can be accomplished by inserting an HTML reference document below the message line.

```
force complete_sensitivity in process_statement
  message "Missing or redundant signals in sensitivity list"
  html_document "rmm_rtl_coding_guidelines.html#G_5_5_5_1"
  severity ERROR
```

When the rule is violated, the user is directed to the HTML document where more information is available. Notice that the format includes an optional link within the HTML document (#G_5_5_1_1) to the specific location of the reference.

© Synopsys, Inc. 2001

Exercise: *Using the very first exercise in this tutorial, add an html_document reference line of code and recompile. For the reference document, either create a simple HTML document or just reference a known HTML address.*

The terms `complete_sensivity` and `process_statement` can be found in the User Manual as a local attribute and a primary template, respectively. Although `process_statement` is defined in the LRM, `complete_sensivity` is not found in the LRM.

Example 3: hardware rule

For our third example, we are going to write a hardware rule requiring that synchronous resets be present in all flip-flops. A hardware rule controls the “hardware semantics” of VHDL. This means certain VHDL constructs infer specific hardware features if the description is synthesized. The following code shows how we can write this rule.

```
Example_3:
  force synchronous_reset in flipflop
  message "Flip-flops with synchronous resets only"
  severity ERROR
```

Both `synchronous_reset` and `flipflop` can be found in the User Manual as primary templates, however neither term is found in the LRM. Be aware that some terms may be found in the User Manual and not in the LRM. Conversely, some terms may be found in the LRM and not in the User Manual.

Example 4: VHDL template basics

We are now going to write a rule that will formally introduce the concept of templates. Understanding templates is one of the keys to learning VRSL.

5.3.1 Templates

A template defines a snapshot of how the VHDL code should appear. Templates can also be thought of as basic elements of VRSL code that are used to build a rule or even another template. Templates can be either pre-packaged (primary template or secondary template) or user-defined. The template always precedes the code that calls it.

For our example, we want to write a rule requiring that signal declarations in packages have default values. For this rule, we will first create a template named `PKG_SIG_DECL`. Note that the name of the template is our choice, however upper-case lettering is a convention that helps identify templates in the VRSL code.

```
  template PKG_SIG_DECL is signal_declaration
   force default
  end
```

VHDL Code

The following VHDL code demonstrates the use of this rule to flag an error.

```
PACKAGE example4_pkg IS

    -- Program memory:
    CONSTANT sg1  : NATURAL ;      -- Will fire, because there is no
                                      default value
    CONSTANT sg2  : NATURAL := 16; -- OK, default value is 16
    CONSTANT sg3  : NATURAL ;      -- Will fire, because there is no
                                      default value
    CONSTANT sg4  : NATURAL := 2* sg2; -- OK, default value is 2*sg2
END example4_pkg
```

Regarding the VRSL code we just wrote, note the similarity of the template syntax to that of the six basic LEDA commands:

```
template C is A
```

where

C = the name of the template
A = primary template or secondary template

All templates have this general form with the template description (code) falling between the `tem-plate` and `end` statements. The template description can contain multiple lines of code.

Getting back to the function of our code, the `force default` line requires (forces) this template to be default with regard to signal declarations. In effect, this template is only looking for VHDL code that deals with default signal declarations.

The next step in writing our rule is to implement a LIMIT command that will actually be our rule. The next section of the code looks like this:

```
  limit signal_declaration in package_declaration to PKG_SIG_DECL
  message "Signal declarations in packages must have default value"
  severity ERROR
```

When we assemble the two parts, the code looks like this:

```
  template PKG_SIG_DECL is signal_declaration
   force default
  end

Example_4:
  limit signal_declaration in package_declaration to PKG_SIG_DECL
  message "Signal declarations in packages must have default value"
  severity ERROR
```

This example introduces you to the basics of templates. As a short exercise, look up the `signal_declaration` in both the LRM and the User Manual. Try and understand the similarities and differences between the terminology used in each manual. Also study the syntax and structure of the basic template used in this example. Remember that understanding templates is one of the keys to learning VRSL.

5.4 NO Command

Two versions:

```
no A′ [in A′′]
no A′′′ in A [of A]
```

where

A = primary template or secondary template
A’ = A, local attribute, or aggregate attribute
A” = A or all
A”’ = list of templates

The NO command can be thought of as a requirement that a particular construct be **absent** in the VHDL code.

Example 5: NO command without context

In this first example of the NO command, we will write a simple rule without context. The rule requires that alias declarations be ignored. The following code shows how we can write this rule.

```
Example_5:
  no alias_declaration
  message "Alias declarations are ignored"
  severity WARNING
```

Example 6: VHDL NO command with context

In our next example of the NO command, we will add context to the command. This first rule we are writing requires that process statements be ignored in (the context of) entity declarations. The following code shows how we can write this rule.

```
Example_6a:
  no process_statement in entity_declaration
  message "Process statements are ignored in entities"
  severity WARNING
```

<u>VHDL Code</u>

The following VHDL code demonstrates the use of this rule to flag an error.

```
----------------------------------------------------------------------
-- ENTITY DECLARATION ------------------------------------------------
----------------------------------------------------------------------

-- LIBRARY DEFINITIONS
LIBRARY IEEE;
USE IEEE.STD_LOGIC_1164.ALL;

ENTITY example_6a_en IS
   GENERIC (
      width : NATURAL := 16
   );
   PORT (
      clk     : IN  STD_LOGIC;                              -- clock
      reset_n : IN  STD_LOGIC;                             -- reset, active low
      en      : IN  STD_LOGIC;                             -- enable, active high
      d       : IN  STD_LOGIC_VECTOR(width -1 DOWNTO 0); -- data in
      q       : OUT STD_LOGIC_VECTOR(width -1 DOWNTO 0)  -- data out
   );
   reg: PROCESS (clk)  <===== Will fire

   BEGIN

      IF RISING_EDGE(clk) THEN
         IF reset_n = '0' THEN
            q <= (OTHERS => '0');
         ELSE
            IF en = '1' THEN
               q <= d;
            END IF;
         END IF;
      EN IF;

   END PROCESS reg;

END example_6a_en;
```

The second rule for this example is left as an exercise for you. See if you can complete the rule in the following exercise.

Exercise: *Write a rule that requires that port default values be ignored. Hint: The context for this rule is a primary template that can be found in the User Manual and also in the very first exercise in this tutorial. Fill in your answer below.*

no default in ____________________
message "Port default values are ignored"
severity WARNING

```
Example_6b:
  no default in port_declaration
  message "Port default values are ignored"
  severity warning
```

Example 7: NO inferring a hardware rule

The use of latches is generally considered poor coding practice. We will use this example to write a rule that globally prohibits latches in VHDL code. Since a latch is hardware, this is another example of a hardware inference rule similar to the one we saw in Example3.

```
Example_7:
  no latch
  message "Avoid using latches in design"
  html_document "rmm_rtl_coding_guidelines.html#G_5_5_2_1"
  severity ERROR
```

Example 8: NO command used with a template

In this example, we introduce the use of a NO command with a template. The rule we are writing prohibits the use of an XNOR binary operator.

```
 template NO_XNOR is binary_operation
    limit operator_symbol to "STD.STANDARD.XNOR"
 end

Example_8:
 no NO_XNOR in binary_operation
 message "STD.STANDARD.XNOR operator not allowed"
 severity ERROR
```

Pay particular attention to how the NO command calls the template in this example.

Exercise: *Try to write a rule that prohibits the use of the XOR binary operator.*

Example 9: expressions in attribute names

In this example, we want write a rule to prohibit the use of expressions in attribute names. First, we create a template using a NO command without context in which expressions are prohibited in attribute names. Next, we use the LIMIT command to constrain the VHDL code according to the template.

```
  template ATTR_NAME is attribute_name
    no expression
  end

Example_9:
  limit attribute_name to ATTR_NAME
  message "Expressions in attribute names are illegal"
  severity ERROR
```

5.5 LIMIT Command

```
limit A [in A’’] to L*
```

where

A = primary template or secondary template
A” = A or all
L = limit list

Whereas the FORCE and NO commands represent two extremes, the LIMIT command allows VHDL constructs to fall within a prescribed legal set.

Example 10: clock naming (basic LIMIT command)

Since we have already seen the LIMIT command used with other commands, it will be helpful to look at a standalone LIMIT command. Suppose we want to limit all clocks to one name (e.g., “clk”). We can do this with the following code.

```
template CLOCK_WITH_ID is clock
  limit identifier to "^clk"
end

Example_10:
  limit clock to CLOCK_WITH_ID
  message "A clock signal should be called 'clk'"
  severity ERROR
```

Pay particular attention to the overall structure of the code. We first create the CLOCK_WITH_ID template to describe the clock by limiting the clock name to “clk.” We then write the rule that limits all clocks to the name (“clk”) as described in the template.

Example 11: VHDL LIMIT command and combinatorial logic

In this example, we will use the LIMIT command to write a rule that prohibits combinatorial processes from inferring registers or latches. The following code shows how we can write this rule.

```
  template PSS_COMBINATIONAL is process_statement
   force combinatorial
  end

Example_11:
  limit process_statement to PSS_COMBINATIONAL
  message "Only combinatorial process statements allowed"
  severity WARNING
```

The template section of code constrains for combinatorial (combinational) logic while the LIMIT command enforces the rule by limiting process statements to combinational logic only.

VHDL Code

The following VHDL code demonstrates the use of this rule to flag an error.

```
library IEEE;
use     IEEE.std_logic_1164.all;

entity example_11_en is

  port (clk      : in  std_logic;
        data_in  : in  std_logic_vector(63 downto 0);
      data_out : out std_logic_vector(63 downto 0)
       );

end example_11_en;

architecture RTL of example_11_arch is

  signal ff_in  : std_logic;
  signal ff_out : std_logic;
  signal comb   : std_logic;

begin

  Data_input: process  -- <==== Will fire here
              begin
              wait until clk'event and clk = '1';
            ff_in <= data_in(55);
              end process Data_input;

  comb <= not ff_in;

  Data_output : process  -- <==== Will fire here
                begin
              wait until clk'event and clk = '0';
              ff_out <= comb;
                end process Data_output;

  data_out(63) <= ff_out;
  data_out(62 downto 0) <= (others => '0');
```

```
end RTL;
```

Example 12: VHDL full LIMIT command

This example demonstrates the use of the full LIMIT command. The rule requires entities be named according to a particular format.

```
Example_12:
  limit file_name in entity_declaration to "<entity>.vhd"
  message "Entities should be limited to files named <entity_name>.vhd"
  severity WARNING
```

VHDL Code

The following VHDL code demonstrates the use of this rule to flag an error.

```
library IEEE;
use     IEEE.std_logic_1164.all;

entity example_18_en is

  port (clk      : in  std_logic;
        clk1     : in  std_logic;
        data_in  : in  std_logic_vector(63 downto 0);
        data_out : out std_logic_vector(63 downto 0)
       );

end example_18_en;
```

Example 13: LIMIT command with multiple templates

This example demonstrates the use of the LIMIT command with multiple templates. It also introduces the use of the SET command. We are writing a rule that prohibits the use of literals in signal assignment statements.

```
  template LOGIC_1 is literal
    limit value      to "1"
    set   value_type to enumerated_literal_type
  end
  template LOGIC_0 is literal
    limit value      to "0"
    set   value_type to enumerated_literal_type
  end

  template LOGIC_Z is literal
    limit value      to "Z"
    set   value_type to enumerated_literal_type
  end

  template INTEGER_0 is literal
    limit value      to 0
    set   value_type to integer_literal_type
  end

  template INTEGER_1 is literal
    limit value      to 1
    set   value_type to integer_literal_type
  end

Example_13:
  limit literal in signal_assignment_statement to LOGIC_1,
                                                 LOGIC_0,
                                                 LOGIC_Z,
                                                 INTEGER_0,
                                                 INTEGER_1
  message "No literals in signal assign. statements-- use constants"
  html_document "rmm_rtl_coding_guidelines.html#G_5_3_2_1"
  severity WARNING
```

Note how all the templates are called from a single LIMIT command (bolded).

Example 14: LIMIT command and conditionals

In this example, we formally introduce the use of conditionals (if... then) with LIMIT commands. It is important to understand that LIMIT commands are the only LEDA commands that use conditional statements.

The rule we are writing constrains the prefix of active low and high resets. To write this rule, we first write four templates that do the following: 1) define high asynchronous reset edges, 2) define high asynchronous names, 3) define low asynchronous reset edges, and 4) define low asynchronous names. We then test for the conditions we have defined in the templates and if met, the rule is implemented with the appropriate LIMIT command. Before we write the rule, however, let’s look at an important concept.

© *Synopsys, Inc. 2001*

5.5.1 *BETTER Rule Writing Paradigm*

At this point, you should notice a pattern emerging when writing rules in VRSL: the basic elements of the rule description become templates which then become rules. We now introduce the concept of adding conditional logic statements in our code to translate these template elements into complex rules. This BETTER rule writing paradigm is summarized as follows:

Basic rule descriptive **E**lements become
Templates which using logic **T**ranslat**E** into **R**ules.

Using the *BETTER* paradigm, you should be able to sketch out the basic structure of complex rules even though you may not be able to fill in all the keywords. Let’s try to apply *BETTER* to our rule for this example (constraining the prefix of active low and high resets).

According to the first part of ***BE****TT****ER***, we know the **B**asic rule **E**lements will become **T**emplates. The second part of *BET**TER*** tells us we will then use these templates with conditional logic (LIMIT commands) to **T**ranslat**E** into our **R**ule.

First, we will set up the templates. We know we need at least two templates for this rule since we are dealing with high and low resets. Recalling our template syntax (`template C is A`), we can easily sketch in the basic template structure as follows:

```
template (high asynch reset) is A
    LEDA command
  end

  template (low asynch reset) is A
    LEDA command
  end
```

Recalling the second paragraph at the beginning of this example, we will actually need four templates to get the job done. Try the following exercise before you see how the template code is written.

Exercise: *Since four templates are needed, we know that each reset (high or low) will have two templates. For each reset, we will need to define the edge type and naming convention. See if you can sketch out the four templates using the User Manual. Hint: Substitute asynchronous_reset for A in the code above.*

Hopefully, you got something similar to the following for the templates:

```
template HIGH_ASYNCH_RESET is asynchronous_reset
    set edge to High_Level
  end

  template HIGH_ASYNCH_RESET_NAME is asynchronous_reset
    limit identifier to "^rst","^rst$"
  end

  template LOW_ASYNCH_RESET is asynchronous_reset
    set edge to Low_Level
  end

  template LOW_ASYNCH_RESET_NAME is asynchronous_reset
    limit identifier to "^rst_n$"
  end
```

Now, let’s finish up the rule using the last part of the *BET**TER*** paradigm (i.e., use conditional logic to **T**ranslat**E** into our **R**ule). This means we need conditional LIMIT commands (if… then) to test for conditions and implement rules.

The overall structure of the code should look like the following:

(Note that the templates are symbolic for instructional purposes)

```
template xyz-high1 is A
template xyz-high2 is A
template xyz-low1 is A
template xyz-low2 is A

limit A to (xyz-high1, xyz-low1) severity NOTE

if (xyz-high1) then
  limit A to (xyz-high2)
  message "message text goes here"
  severity ERROR
end if

if (xyz-low1) then
  limit A to (xyz-low2)
  message "message text goes here"
  severity ERROR
end if
```

The difficulty now is determining A. If you did the previous exercise, you would know that **A** for the templates and the LIMIT commands is `asynchronous_reset`. Filling in the LEDA commands are now all we have left to complete our code.

Exercise: *Try to fill in the rest of the code for this rule using the BETTER paradigm. Remember to use the reference manual.*

When we put everything together, the complete code looks like the following:

```
template HIGH_ASYNCH_RESET is asynchronous_reset
  set edge to High_Level
end

template HIGH_ASYNCH_RESET_NAME is asynchronous_reset
  limit identifier to "^rst","^rst$"
end

template LOW_ASYNCH_RESET is asynchronous_reset
  set edge to Low_Level
end
```

```
  template LOW_ASYNCH_RESET_NAME is asynchronous_reset
    limit identifier to "^rst_n$"
  end


-- Use ^rst for active high reset signals, rst_n for active low

  limit asynchronous_reset to HIGH_ASYNCH_RESET,LOW_ASYNCH_RESET severity NOTE

    if HIGH_ASYNCH_RESET then
      Example_14:
      limit asynchronous_reset to HIGH_ASYNCH_RESET_NAME
      message "Active high resets should be prefixed with rst"
      html_document "rmm_rtl_coding_guidelines.html#G_5_2_1_9"
      severity WARNING
    end if

    if LOW_ASYNCH_RESET then
      Example_14:
      limit asynchronous_reset to LOW_ASYNCH_RESET_NAME
      message "Active low resets should be called rst_n"
      html_document "rmm_rtl_coding_guidelines.html#G_5_2_1_9"
      severity WARNING
    end if
```

You should review this example, paying particular attention to the application of the *BETTER* paradigm and the overall structure of the code. Conditional LIMIT commands are one of the most powerful tools in VRSL.

5.6 SET Command

```
set X [in A’’] to Y
```

where

A = primary template or secondary template
A” = A or all
X = set_attribute
Y = STRING, number, or Enumerated type value

The SET command sets an attribute to a certain value.

Example 15: full SET command

We have already seen a SET command used in a previous example with a LIMIT command.

```
  set value_type to enumerated_literal_type
```

It should be obvious that the SET command is setting an attribute to a value. In our first example of the SET command, we will use the full SET command to write a rule to make sure that for loops have globally static bounds. The following code shows how we can write this rule.

```
Example_15:
  set evaluation_time in for_loop_statement to Globally_Static_Evaluation
  message "For loops must have globally static bounds"
  severity ERROR
```

Example 16: SET command with template

In this example, we demonstrate the use of a template with a SET command. The rule we are writing prohibits real literals.

```
  template BAD_LIT is literal
    set value_type to real_literal_type
  end

Example_16:
  no BAD_LIT in literal
```

© *Synopsys, Inc. 2001*

```
message "Real literals are not allowed"
severity ERROR
```

5.7 MAX/MIN Command

```
max M [in A] is N
min M [in A] is N
```

where

A = primary template or secondary template
M = max_min_attribute
N = a number

The MAX/MIN command is used to establish maximum/minimum values for certain attributes. Note that you can only set maximum and minimum values for certain templates.

Example 17: VHDL clock expressions

In this example of a MAX command, we establish a maximum of one clock expression per process.

```
Example_17:
  max clock_expression_count in process_statement is 1
  message "Only one clock expression per process is allowed"
  severity ERROR
```

VHDL Code

The following VHDL code demonstrates the use of this rule to flag an error.

```
architecture RTL of example_18_en is

  signal ff_in  : std_logic;
  signal ff_out : std_logic;
  signal comb   : std_logic;

begin

  Data_input: process (clk)  -- <==== Will NOT fire here
              begin
                if (clk'event) and clk = '1' then
                ff_in <= data_in(55);
          end if;
              end process Data_input;

  comb <= not ff_in;
  Data_output : process (clk, clk1)  -- <==== Will fire here
                begin
                  if (clk'event) and clk = '0'then
                  ff_out <= comb;
     end if;
  if (clk1'event) and clk1 = '0' then
               ff_out <= comb;
        end if;
                end process Data_output;

  data_out(63) <= ff_out;
  data_out(62 downto 0) <= (others => '0');

end RTL;
```

Example 18: duplicate rule labels and messages

This example shows the use of a common label and message with multiple rules. Notice the duplicate rule labels and messages. Since the rule we are writing prohibits multi-dimensional arrays, we must establish the maximum dimension (1) for both unconstrained and constrained arrays. Thus, we are able to use the same rule label and error message.

```
Example_18:
  max dimension_count in unconstrained_array_definition is 1
  message "Multi-dimension arrays are illegal"
  severity ERROR

Example_18:
  max dimension_count in constrained_array_definition is 1
  message "Multi-dimension arrays are illegal"
  severity ERROR

At first glance the MAX commands look identical, but if you look closer you
will see that their context (bolded) is different.
```

Example 19: template inheritance

In this example, we use a MAX command to show how one template can inherit another. The rule is limiting the entity name to 20 characters.

```
  template SHORT_ENTITY_ID is identifier
    max character_count is 20
  end

  template SHORT_NAMED_ENTITY is entity_declaration
    limit identifier to SHORT_ENTITY_ID
  end

Example_19:
  limit entity_declaration to  SHORT_NAMED_ENTITY
    message "Name of entity is too long - Max 20 characters"
    html_document "rmm_rtl_coding_guidelines.html#G_5_2_1_4"
    severity ERROR
```

Notice how the second template inherits from the first template. This inheritance process is valid for any of the LEDA commands.

Example 20: multiple MAX/MIN commands

This example shows how MAX and MIN commands can be combined in a single template. The rule sets the range for integer values.

```
  template MAX_INTEGER_RANGE is range
    max high_bound is 2147483647
    min low_bound  is -2147483647
  end

Example_20:
  limit range in integer_type_definition to MAX_INTEGER_RANGE
  message "Integer value  must be in range -(2**31-1) to (2**31-1)"
  severity ERROR
  end
```

6. Appendix

6.1 Definition of Command Terminology

Following is a partial listing of keywords, templates, and attributes used in VRSL. This information comes from the ProVHDL Rule Specificier User Manual. The User Manual also contains a double cross-referenced index of templates and attributes called the SpecDex which is designed to be used as an “online” reference (PDF format).

6.1.1 Commonly-used Primary Templates (primary_template)

Access_type_definition	aggregate	alias_declaration
Architecture_body	assertion_statement	Association_list
Asynchronous_reset	attribute_declaration	attribute_name
Attribute_specification	binary_operation	binding_indication
Block_configuration	block_specification	block_statement
Case_statement	clock	Component_configuration
Component_declaration	component_instantiation_statement	Component_specification
Concurrent_procedure_call_statement	conditional_signal_assignment	Configuration_declaration
configuration_specification	constant_declaration	Constrained_array_definition
disconnection_specification	entity_declaration	Enumeration_type_definition
exit_statement	file_declaration	file_layout
file_type_definition	for_loop_statement	formal_parameter
Flipflop	floating_type_definition	function_call
generate_statement	group_declaration	group_template_declaration
Identifier	if_statement	integer_type_definition
interface_constant_declaration	interface_file_declaration	interface_signal_declaration
interface_variable_declaration	latch	Literal
loop_statement	next_statement	package_body
package_declaration	physical_type_definition	procedure_call_statement
process_statement	range	record_type_definition
report_statement	return_statement	selected_signal_assignment
shared_variable_declaration	signal_assignment_statement	signal_declaration
subprogram_body	subprogram_declaration	subtype_declaration
subtype_indication	synchronous_reset	type_declaration
unconstrained_array_definition	use_clause	Unconstrained_array_definition
variable_assignment_statement	variable_declaration	wait_statement
|while_loop_statement		

6.1.2 Commonly-used Secondary Templates (secondary_template)

allocator	association_element	conditional_waveforms
element_association	header_comment	indexed_name
selected_name	selected_waveforms	simple_name
slice_name	statement_format	Waveform
waveform_element		

6.1.3 Commonly-used Local Attributes (local_attribute)

actual_designator	actual_function	actual_parameter_part
actual_type_mark	after_expression	Architecture
architecture_name	attribute_designator	block_statement_label
buffer	bus	character_literal
choice	combinatorial	complete_sensitivity
component	condition	Configuration
constant	declarative_region	Default
deferred	design_library	discrete_range
downto	else	Entity
entity_designator	error_id	Expression
file	file_name	formal_function
formal_type_mark	function	generate_statement_label
generic_map_aspect	group	guard_expression
guarded	impure	In
incomplete_type_declaration	index_specification	Inertial
initialize_signals	initialize_variables	Inout
input	label	left_expression
library_clause	limit_id	linkage
logical_name	maximum_variable_usage	multiple_choices
name	name_prefix	named_association
null	null_range	null_statement
object_definition	open	open_kind
operator_symbol	others	out
output	out_params_fully_assigned	package
port_map_aspect	positional_association	postponed
procedure	pure	qualified
range_attribute	read_write	record_aggregate
register	reject_expression	report_expression
resolution_function	return	return_last

right_expression	sensitivity	severity_expression
signal	subtype	target
timeout	to	top_architecture
top_configuration	top_entity	transport
tristate	type	type_conversion
type_mark	unaffected	units
use_exponent	value	variable
waveform_expression		

Commonly-used Aggregate Attributes (aggregate_attribute)

association_element_s	condition_ s	conditional_waveforms
declaration_profile_s	element_association_s	index_constraint_s
selected_waveforms	statement_profile_s	subtype_definition_s
waveform_element		

limit_list

one_of_limit_list, allof_id_list, allof_template_list

set_attribute

base, base_specifier, evaluation_time, value_type

max_min_attribute

asynchronous_reset_signal	character_count	clock_expression_count
clock_signal	connections	dimension_count
element_count	file_length	high_bound
line_count	low_bound	object_count
parameter_count	unit_count	waveform_count

Index

defined 20