Leda
Rule Specifier Tutorial

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

Figures

Preface

About the Manual

This tutorial is an example-based introduction to using Leda to check your VHDL or Verilog code for errors or anomalies that may cause problems downstream in your design and verification flow.

This tutorial is intended for use by design and quality assurance engineers who are already familiar with VHDL or Verilog.

Related Documents

This manual is part of the Leda documentation set. For a complete list, see the *Leda Document Navigator*.

Manual Overview

This manual contains the following chapters:

Preface	Describes the manual and explains how to get technical assistance.
Chapter 1 Writing Rules for VHDL	An example-based tutorial for users who want to learn how to use Leda to specify coding rules and verify that their VHDL designs comply with those rules. Includes a hands-on introduction to the VRSL language that you use to specify coding rules for VHDL designs.
Chapter 2 Writing Rules for Verilog	An example-based tutorial for users who want to learn how to use Leda to specify coding rules and verify that their Verilog designs comply with those rules. Includes a hands-on introduction to the VeRSL language that you use to specify coding rules for Verilog designs.

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 Writing Rules for VHDL

Introduction

Welcome to the *Leda Rule Specifier Tutorial*, an example-based introduction to learning how to write coding rules for use with Leda. You use these coding rules to check your VHDL designs for errors or anomalies that may cause problems for downstream tools in the design and verification flow. For general information about Leda, see the *Leda User Guide*.

You need an optional Specifier license in order to perform the exercises in this tutorial. The tutorial is divided into two parts. Part one takes you through the process of writing some sample rules and organizing them so that you can check some test VHDL code using Leda, and later fix the problems right from the tool. This part of the tutorial is organized in the following major sections:

Part two explores the syntax and semantics of VRSL, the VHDL rule-writing language. Hands-on examples are provided there to give you a feel for all of the VRSL commands and capabilities. This part of the tutorial is organized in the following sections:

Writing Rules

In this exercise we'll create four new rules in the VHDL rule specification language (VRSL) and organize them into two different rulesets that reside in one ruleset.rl file. Then we'll write some VHDL code, which the rules will check. Finally, we'll use the rules we created to check our sample VHDL code. Note that for this first exercise, we'll create some relatively simple rules so that you can get a feel for rule creation, compilation, checking, and debugging. After you master this flow, you can proceed to the second part of this tutorial to learn how to use all of the VRSL commands to develop more complicated or sophisticated rules.

If you haven't already done so, install the Leda software and configure your environment as described in the *Leda Installation Guide*. Then, begin by invoking the Specifier tool as follows:

```
% $LEDA_PATH/bin/leda -specifier &
```

The Specifier main window (Figure 1) opens:

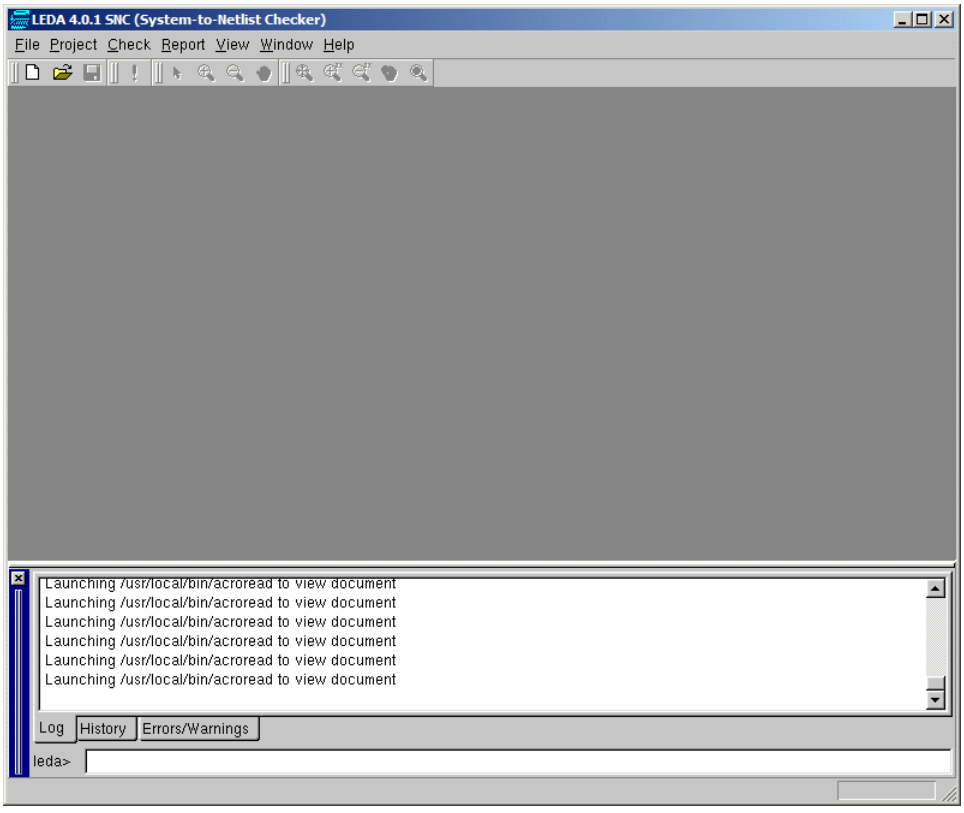

Figure 1: Specifier Main Window

Creating Ruleset Files

Suppose we need four new rules that:

- Make sure latches are not used in the design
- Catch missing or redundant signals in sensitivity lists
- Make sure that clocks are named “clock” or end in “_clk”
- Catch cases where clocks are mixed with different clock edges

For the sake of this exercise, let’s assume that we were unable to find rules to handle these needs in any of the prepackaged rules that come built-in with the Leda Checker. (In fact, there are rules similar to these that you can copy and modify.) To begin, follow these steps:

1. Using a text editor, type in the following VRSL source code exactly as shown. You can use the text editor in the Specifier by choosing **File > New**. (Note that two dashes “--” at the beginning of a line designate a comment.) Create the file as “ruleset.rl”. Note that “.rl” is the standard extension for VHDL ruleset files.

Hint

For your convenience, you can find the following example VRSL code in the $LEDA_PATH/doc/tutorial_specifier/rsl/ruleset.rl file. If you are viewing this document online, you can also cut-and-paste the text right from this PDF file.

```
------------------------------------------------------------
-- VHDL rules
------------------------------------------------------------
-- In this file we create 2 rulesets, with 2 rules in each:
-- TUTOR_RS
--   TUT_1  Avoid using latches in design
--   TUT_2  Missing or redundant signals in sensitivity list
-- TUTOR_CLOCK
--   TUT_3  Clock name must end with _clk or be clock
--   TUT_4  Avoid mixing clock with different clock edge
--
-- Rules were copy/pasted and adapted from existing Leda rules found
-- in the RMM coding guidelines policy.
-- The source of this policy for VHDL is in $LEDA_PATH/rules/rmm/
-- RMM.rl
-- For each rule, you must copy the "command" part of the rule
-- as well as the "template" part.
------------------------------------------------------------
```

```
ruleset TUTOR_RS is

------------------------------------------------------------
--   Command section
------------------------------------------------------------
-- Rule TUT_1
-- Avoid using latches in design
-- Copied from rule R_552_1 in $LEDA_PATH/rules/rmm/RMM.rl
-- Here, we just changed the label from R_552_1 to TUT_1
-- and commented out the link to an html document
---------------------------
TUT_1:
no latch
  message "Avoid using latches in design"
--  html_document "pol_rmm.html#R_552_1"
  severity ERROR

------------------------------------------------------------
-- Rule TUT_2
-- Missing or redundant signals in sensitivity list
-- Copied from rule R_554_1 in $LEDA_PATH/rules/rmm/RMM.rl
---------------------------
TUT_2:
force complete_sensitivity in process_statement
  message "Missing or redundant signals in sensitivity list"
--  html_document "pol_rmm.html#R_554_1"
  severity ERROR

end ruleset

------------ end of ruleset  TUTOR_RS.  ------

ruleset TUTOR_CLOCK is

------------------------------------------------------------
--   Template section
------------------------------------------------------------
-- Template for use with rule TUT_3 below
-- Copied from the template used by G_521_6
-- Here we changed the regular expressions to
-- reflect our own new naming convention
template CLOCK_NAME is clock
limit cased_identifier to "_clk$", "clock"
end
```

```
------------------------------------------------------------
--   Command section
------------------------------------------------------------
-- Rule TUT_3
-- Clock name must end with _clk or be clock
-- Adapted from rule G_521_6 in $LEDA_PATH/rules/rmm/RMM.rl
-- We changed the message, and the regular expressions
-- used in the template CLOCK_NAME
---------------------------
TUT_3:
limit clock to CLOCK_NAME
  message "Clock name must end with _clk or be clock"
--  html_document "pol_rmm.html#G_521_6"
  severity WARNING

------------------------------------------------------------
-- Rule TUT_4
-- Avoid mixing clock with different clock edge
-- Adapted from rule G_541_1 in $LEDA_PATH/rules/rmm/RMM.rl
-- We changed the message
---------------------------
TUT_4:
no mixed_clock in design
  message "Avoid mixing clock with different clock edge"
-- html_document "pol_rmm.html#G_541_1"
  severity WARNING

end ruleset
```

Creating Policies

Now that we have a ruleset.rl file that contains our new rule source code, we need to store the new rules in a policy. To do that, follow these steps:

1. From the Specifier main window, first open the Rule Wizard (**Check > Configure)**, and from there open the Policy Manager window **(Tool > Policy Manager**).
2. Click the VHDL tab, and then click the New button on the right side of the display. Type in a name for the new policy (for example, “MY_TUTOR”) and click OK.
3. When your new policy name appears in the Policies pane, click it to highlight the name and then click in the Rulesets pane. Click the Add button. This opens the “Please choose a rule file” window.
4. Navigate to the location of the ruleset.rl file you just created and click on the file name. Then click the Add button. This causes the tool to compile your new rulesets. You should see messages in the Log tab at the bottom of the Specifier main window similar to the following:

   ```
   Compiling ruleset TUTOR_RS
   Compiling ruleset TUTOR_CLOCK
   Compilation done (block level).
   Compilation done (chip level).
   ```

   The Rulesets pane in the Policy Manager window now shows the two rulesets we created (TUTOR_CLOCK and TUTOR_RS), and the Templatesets pane shows the templatesets we used.
5. Close the Policy Manager window.

   You have now created a policy (“MY_TUTOR”) containing two rulesets that are both in the same ruleset.rl file. The rulesets contain a total of four rules (two rules each), which we’ll use to check some sample VHDL code.

Using Tcl Shell Mode to Create New Policies

You can also create new policies and rulesets using the rule_manage_policy command in Leda’s Tcl shell mode. For example, to create the “MY_TUTOR” policy, use the following command at the Tcl shell prompt:

```
leda> rule_manage_policy -policy MY_TUTOR create
```

Then, to compile the ruleset.rl file you created for this tutorial, use the following command at the Tcl shell prompt:

```
leda> rule_manage_policy -policy MY_TUTOR compile ruleset.rl
```

For more information on using Tcl shell mode, see the *Leda User Guide*.

Creating a VHDL Test File

To test these rules, we need to create some VHDL code to check them against. We are going to purposely violate all four rules we created in our VHDL code so that we can see how the tool works. When we check the code with our new rules, we should see errors for all of them. Follow these steps:

1. Using a text editor, type in the following text, and save the file as test.vhd.

```
-  Simple testcase for the rules in the VHDL policy built with
--     ruleset.rl file
--  Will fire the rules:
--   TUT_1  Avoid using latches in design
--   TUT_2  Missing or redundant signals in sensitivity list
--   TUT_3  Clock name must end with _clk or be clock
--   TUT_4  Avoid mixing clock with different clock edge

library IEEE;
use IEEE.std_logic_1164.all;
entity test_ent is
  port(data, clk, main_clk, clock, rst, load_clk :in std_logic;
       q, q1, q2, q3: out std_logic);
end;

architecture rtl of test_ent is
begin

P_1: process(clock)
       begin
         if (rst = '1') then
-- TUT_2 fires: rst is not in sensitivity list
             q <= '0';
         elsif (clock'event and clock = '1') then
             q <= data;
         end if;
       end process;

P_2: process(clk)
       begin
         if (clk'event and clk = '1') then
-- TUT_3 fires: bad naming for a clock
             if (rst = '1') then
               q1 <= '0';
             else
               q1 <= data;
             end if;
         end if;
       end process;
```

```
P_3: process(main_clk)
-- TUT_4 fires: we mixed clock with different edges
        begin
          if (main_clk'event and main_clk = '0') then
              if (rst = '1') then
                 q2 <= '0';
              else
                 q2 <= data;
              end if;
          end if;
        end process;

P_4: process(load_clk, data)
        begin
          if (load_clk = '1') then
              q3 <= data; -- TUT_1 fires: we infer a latch here
          end if;
        end process;
end rtl;
```

Creating a Project File

Before we can use Leda to test our new rule against the sample VHDL file we created, we must first create a project file. A project file organizes our VHDL files into easily managed units. Follow these steps:

1. From the Specifier main window, choose **Project > New**. This opens the Project Creation Wizard window.
2. Click the Specify Project Name button. This opens the Specify Project Name window. Use the Browse button to navigate to the location where you want your project file to reside (for example, “WORK”) and enter the project name (for example “my_project”). Then click Save.
3. Now click the Next button at the bottom right of the window. This takes you to the Specify Compiler Options part of the Wizard, which has tabs for VHDL and Verilog. Click the Verilog tab.
4. In the Severity Level pane, click the radio button for the lowest severity level for which error messages from the VHDL analyzer compiler will be printed. Analyzer messages with a severity below the specified value are not printed. (This severity level is only used for VHDL syntax analysis, not for checking.) The default is Warning.
5. In the Version pane, click the 87 or 93 or radio button, depending on the version of VHDL you are using. The default is VHDL 93.
6. Click Next. This takes you to the Specify Libraries part of the Wizard, which has tabs for VHDL and Verilog. Click the VHDL tab.
7. In the Working Libraries pane, specify the logical names of working libraries where the VHDL analyzer will store binary results of the VHDL analysis. For this exercise, we are not using any specific working libraries, so we'll leave this pane empty and let the tool put our analyzed code into the default location.
8. In the Library Directories pane, specify the path to any directories to be searched for included files in your design. For this test, we don’t have any include files, so leave this pane empty.
9. In the Library Files pane, specify the logical names and mappings to the physical locations of additional existing compiled resource libraries. These are golden libraries that can be shared by multiple projects and users and usually contain common packages. (By default, the standard IEEE, STD, and Synopsys libraries are available.) For this exercise, we can use the default resource libraries, so just click Next to accept the defaults.

10. This takes you to the Specify Source Files part of the Wizard, which has tabs for VHDL, Verilog, and All. (The All tab is for mixed-language designs.) Source files, in this case, means VHDL source files, the ones we want to check against our new rules. Click the VHDL tab.
11. In the Directories/Files pane, click the Add button. This opens the Add VHDL Source Directory window. Navigate to the directory that contains the test.vhd file you just created to test the new rules. Then click Next.
12. This takes you to the Confirm & Create part of the Wizard. Leave the Build with Check checkbox selected and click Finish. If the tool displays a small Get Top Module/Design Entity window, note that this information is needed for checking chip-level rules. For this exercise, leave these settings at their default values and click the OK button. Leda compiles the VHDL file and executes the Checker. You should see something like the following screen (Figure 2).

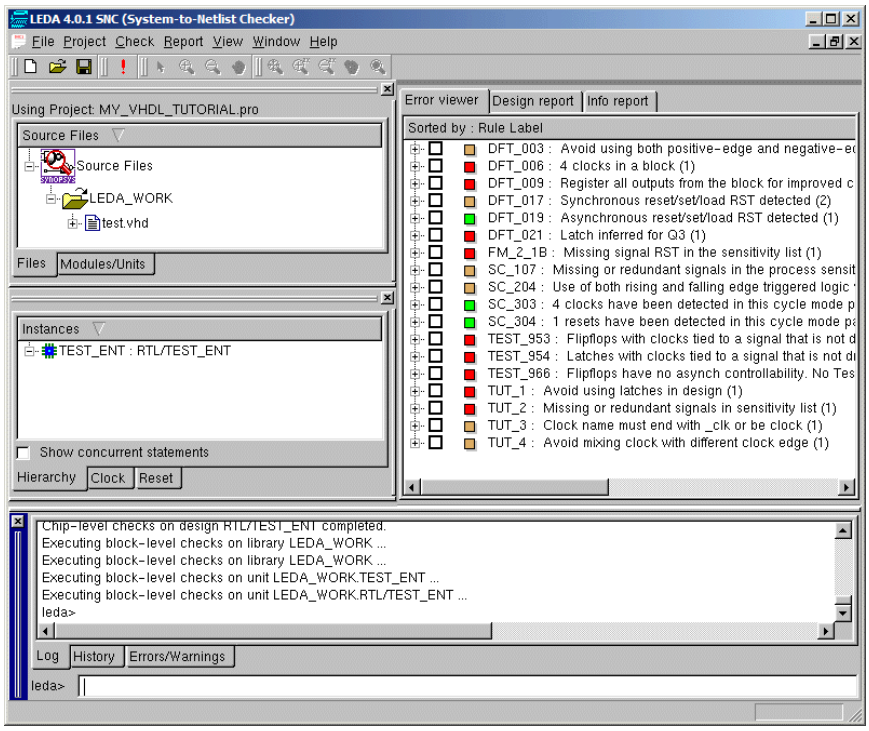

Figure 2: Specifier Project After Build

Note the test.vhd file in the Source Files pane on the left-hand side of the main window. This is the test file that we wrote earlier to check against the rules we created. But look at all the errors and warnings listed in the Error Viewer pane on the right-hand side of the main window. How many rules can you violate with one simple test file? As it turns

out, quite a few, but we can narrow the display to focus the results to just the new rules we created for this tutorial, as explained in the next section, Running the Checker & Fixing the Errors.

Running the Checker & Fixing the Errors

With the project built, we will now set up and run the Checker, which will check our sample VHDL code against just the new coding rules we created. Follow these steps:

1. From the Specifier main menu, choose **Check > Configure**. This opens the Leda Rule Wizard window, which lists all the prepackaged policies that come with the tool on the left side of the window, in addition to the new policy we just created for this exercise (MY_TUTOR). Some of the prepackaged policies are activated by default. That's why we got so many error and warning messages from the one simple test.vhd file that we wrote for this tutorial. As you learn how to use Leda to check your HDL code, don't let the number of warning and error messages you receive throw you off. In many cases, changing one line of code eliminates lots of error messages all at once. And you can easily turn off rules that you don't consider to be significant for your design (see the section on Deactivating Rules in the *Leda User Guide).*
2. For now, deactivate all policies except "MY_TUTOR" by clicking the icons next to each policy name until the boxes appear empty. When you are done, only the MY_TUTOR box icon should be filled in and colored light blue to indicate only the new rules that we wrote are now selected for checking.
3. Open the MY_TUTOR display by clicking the (+) icon so that we can get a look at the rules we created and how they are organized. Click on the colored box icons to the left of the TUTOR_CLOCK and TUTOR_RS rulesets one at a time. Note how

the upper-right side of the window changes to display the rule labels and messages for each of those rulesets. The display should look similar to the following (see Figure 3).

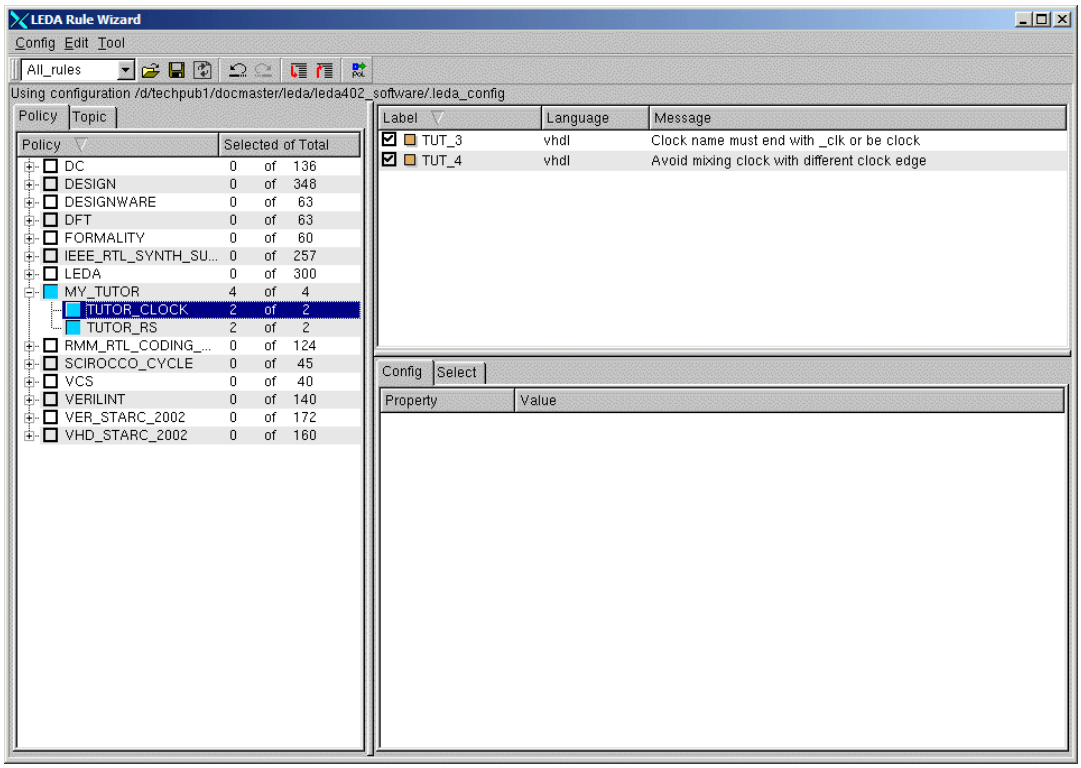

Figure 3: Rule Wizard with Custom Rules Displayed

4. From the Rule Wizard menu, choose **Config > Save**. This activates your new rule configuration for the Checker. Then choose **Config > Close** to dismiss the Rule Wizard.

5. From the Specifier main menu, choose **Check > Execute**. This time we see our same test file (test.vhd) listed in the Files tab on the left, but in the Error Viewer we see just four messages. They are the messages generated because the test file we wrote violates all four rules that we specified in MY_TUTOR custom policy (see Figure 4).

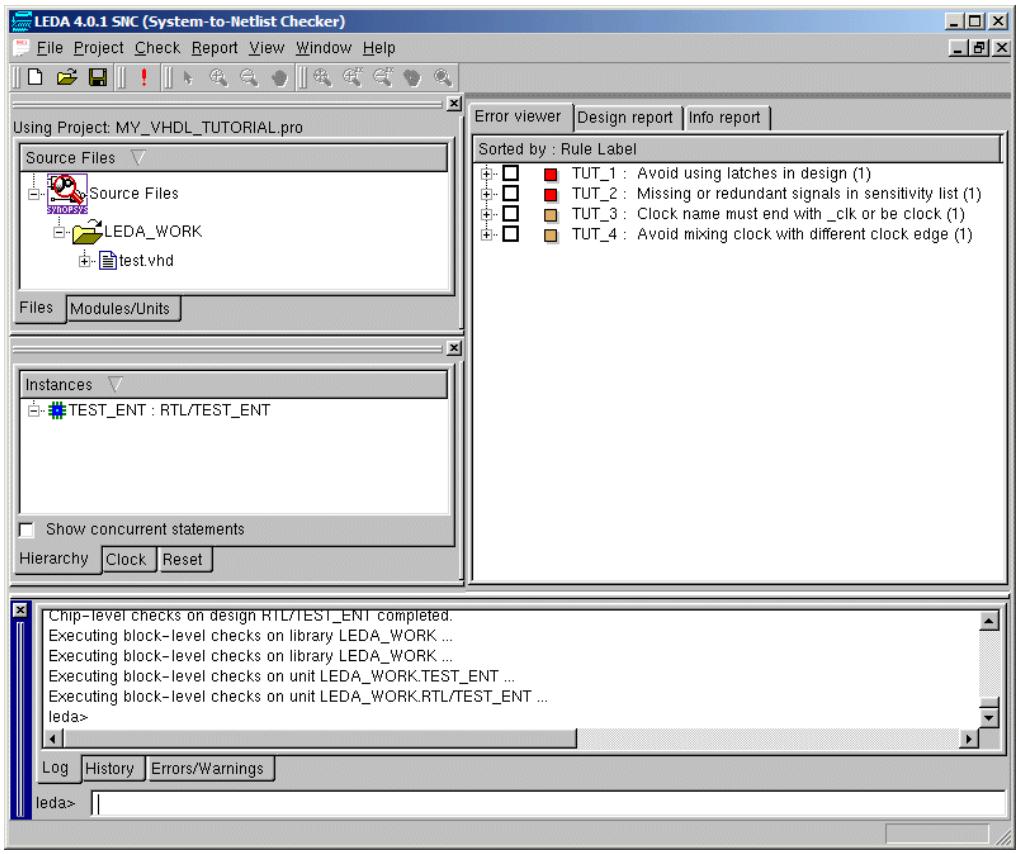

Figure 4: Checker Results for Custom Rules

6. For each warning or error message in the Error Viewer, click the (+) box icon to the left of the message. This expands the display to show the name of the test.vhd file that we tested. Click the next (+) box icon in the hierarchy. The display opens a window right on the VHDL file that was tested, with the offending line of code indicated by a green arrow pointer.

7. For each warning or error message, double click in the source code display in the Error Viewer. This opens a text editor on the file. The suspect code is already highlighted in the file. Correct the problems as shown in the following VHDL code, which is commented to show where the problems are and how you can fix them.

Hint

For your convenience, you can find a copy of this corrected test code at $LEDA_PATH/doc/tutorial_specifier/hdl/test_fixed.vhd.

```
------------------------------------------------------------------------
---
--  Simple testcase for the rules in the VHDL policy built with
--     ruleset.rl file
--  Show the corrections (just for demo purpose)
--  to avoid the firing of the rules:
--   TUT_1  Avoid using latches in design
--   TUT_2  Missing or redundant signals in sensitivity list
--   TUT_3  Clock name must end with _clk or be clock
--   TUT_4  Avoid mixing clock with different clock edge

library IEEE;
use IEEE.std_logic_1164.all;

entity test_ent is
  port(data, clk, main_clk, clock, rst, load_clk :in std_logic;
       q, q1, q2, q3: out std_logic);
end;

architecture rtl of test_ent is
begin

--P_1: process(clock)
P_1: process(clock, rst)
       begin
         if (rst = '1') then
-- TUT_2 fires if rst is not in sensitivity list
             q <= '0';
         elsif (clock'event and clock = '1') then
             q <= data;
         end if;
       end process;
```

```
--P_2: process(clk) -- TUT_3 fires: bad naming for a clock
P_2: process(clock)
        begin
--       if (clk'event and clk = '1') then
-- TUT_3 fires: bad naming for a clock
          if (clock'event and clock = '1') then
-- We just change to a "good" name for demonstration
              if (rst = '1') then
                q1 <= '0';
              else
                q1 <= data;
              end if;
          end if;
        end process;

P_3: process(main_clk)
        begin
 --       if (main_clk'event and main_clk = '0') then
          -- TUT_4 will fire if we mix clock edges
          if (main_clk'event and main_clk = '1') then l
          -- Will prevent TUT_4 from firing, but it may not be
          -- functionally correct...
              if (rst = '1') then
                 q2 <= '0';
              else
                 q2 <= data;
              end if;
          end if;
        end process;

P_4: process(load_clk, data)
        begin
          if (load_clk = '1') then
              q3 <= data;
-- Without the else clause TUT_1 fires: we infer a latch here
          else
              q3 <= '0';
-- The else clause prevents the inference of a latch
          end if;
        end process;

end rtl;
```

8. When you are done fixing each problem, choose **File > Save** from the editor's menu to save your changes.
9. From the Specifier main window, choose **Project > Build**. The tool recompiles your test file.

10. From the Specifier main window, choose **Check > Run**. The tool checks your corrected VHDL code using the rules you created. This time, since we corrected the offending code, our results come up clean, with no error messages listed in the Error Viewer.

This concludes our first exercise with the Leda Specifier tool. As a further exercise, copy some of your design team's VHDL files and try running some or all of the prepackaged policies against them to see what kind of results you get. Notice how even simple changes in your VHDL coding eliminate lots of error messages. This can help avoid downstream bottlenecks in your design and verification flow.

Now that we have the mechanics down for building and checking rules using Leda, let's explore the syntax and semantics of the VRSL rule specification language itself, beginning with the first section in part two of this tutorial: What is VRSL?

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 coding rules using prebuilt templates and attributes, 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. For complete reference information on VRSL commands, templates, and attributes, see the *VRSL Reference Guide*.

All terminology used for writing rules comes from either the IEEE Standard 1076-1993 VHDL Language Reference Manual (LRM) or the *VRSL Reference Guide*. The LRM is the basis upon which the *VRSL Reference Guide* was created. While a few of the terms used in this tutorial are unique to Leda, most of them can be found in the LRM. A review of the LRM and its terminology can help you gain a better understanding of VRSL.

About Templates and Attributes

Before we get started writing more interesting custom rules, let's take a closer look at templates and attributes, because these 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.

When you write a rule with a primary template, you don't need to provide a context. Primary templates are stronger than secondary templates in this way. For example, you can write a rule with the **limit** command and a primary template, such as **module_declaration** like this:

```
limit module_declaration to your_constraint
```

On the other hand, when you write a rule with a secondary template, you need to provide a context. Secondary templates are not as strong as primary templates in this way. For example, you can write a rule with the **limit** command and a secondary template such as **identifier** like this:

```
limit identifier in module_declaration to your_constraint
```

In this example, the **module_declaration** primary template provides the necessary context for the **identifier** secondary template.

Note

Each template in the VRSL rule specification language is either primary or secondary. They are all clearly labelled in the *VRSL Reference Guide*, which provides complete reference information for all templates and attributes, including the attribute name, kind, and limit_kind.

When you write rules, first you define a template, and then you call that template to complete the rule using VRSL commands.

VRSL Rule-Writing Examples

The remainder of this tutorial provides examples of how to write rules that check for common issues of concern in HDL designs. Most of the examples are designed to give you an introduction to the basics of rule writing using the different VRSL commands. Other examples show you how to use different features of the language such as enumerated types, variables, and template inheritance, so that you can see how each one works. The examples are presented in the following sections:

Hint

For your convenience, you can find VRSL source code for all the examples in this chapter in the $LEDA_PATH/doc/tutorial_specifier/rsl directory. The ruleset files that contain these examples are named example_1.rl. and so on, to match the example names used in the tutorial. You can also find VHDL source code for the tests used in these examples in the $LEDA_PATH/doc/tutorial_specifier/hdl directory. The .vhd files are named example_1.vhd and so on to match the examples where they are used.

Requiring Constant Declarations in Packages

For our first example, let's write a rule requiring that constant declarations in packages have default values. First, create a template named PKG_SIG_DECL. Note that the name of the template is your choice, but uppercase lettering is a convention that helps identify templates in the VRSL code.

```
template PKG_SIG_DECL is constant_declaration
force default
end

Example_4a:
limit constant_declaration in package_declaration to PKG_SIG_DECL
message "Constant declarations in packages must have default value"
severity ERROR
```

In this example, we define the string PKG_SIG_DECL to be a **template** of type **constant_declaration**, which is a primary template. Then we limit the scope in our template definition to **force** the **default** attribute of the **constant_declaration** template to be present in the code.

In the next code segment, we write a rule that calls the PKG_SIG_DECL template we defined, using the **limit** command to specify that the context for application of our rule is **package_declarations**. This means that in VHDL code that matches our PKG_SIG_DECL template, **constant_declarations** in **package_declarations** must have **default** values. Otherwise, Leda flags an error.

VHDL Test Code

The following code demonstrates how to use this rule to constrain VHDL. As an additional exercise, create both a rule and VHDL code using these examples and run the Checker.

```
PACKAGE example4_pkg IS
-- Program memory:

  CONSTANT sg1  : NATURAL ; -- Rule fires because there is no default value
  CONSTANT sg2  : NATURAL := 16; -- OK, default value is 16

  CONSTANT sg3  : NATURAL ; -- Rule fires because there is no default value
  CONSTANT sg4  : NATURAL := 2* sg2; -- OK, default value is 2*sg2
END example4_pkg;
```

Making Deferred Constant Declarations Illegal

Suppose we want to make deferred constant declarations illegal. The following code shows how we can use a **force** command to 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 of the **constant_declaration** primary template**.** This is a simple rule where we did not need to provide any context, because we want this rule to apply globally to all **constant_declarations**.

Notice the **message** and **severity** lines in the code. The **message** line contains the text that is displayed when the rule is violated. The **severity** line indicates the level of the violation (note, warning, error, or fatal). If these lines of code are not present, Leda cannot flag a rule violation, so be sure to include them.

If you look in the *VRSL Reference Guide*, you will see the following definition for **constant_declaration**:

Primary template belonging to classes: OBJECT_ITEM (see Table 2).

Table 2: constant_declaration Primary Template Description

Attribute	Kind	Limit_Kind
identifier	template	ID
subtype_indication	template	subtype_indication
default	**template**	**EXPRESSION**
declarative_region	template	REGION
deferred	local	N/A

In Table 2, the **Attribute** column lists the attributes that you can use with the **constant_declaration** primary template. The **Kind** column tells you how those attributes can be used. If the attribute is a **template** kind, this means that it can be used as a template itself with its own context when writing limit, no, and force commands. For example, all statements and declarations are **template** kinds of attributes. The **template** kind of attribute is flexible and powerful in this way.

If the attribute is a **local** kind, it can only be used with no and force commands, and cannot function as a template.

The **Limit_Kind** of an attribute tells you the type of template this attribute can be constrained to. For example, the **default** attribute in our example is constrained for use only with **expressions**. In this example, we are making sure that all constant declarations have a specified default value:

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

If the **Limit_Kind** column says N/A , this means that the attribute is not constrained to a particular type of VHDL construct or template

Adding HTML Help Files for Errors

For our second example, let's write a rule requiring that process sensitivity lists be complete. We'll also include some code that specifies an HTML file that provides additional information about errors. The following code shows how we can use a **force** command to write this rule:

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

Users may sometimes need more explanation of a rule violation than the single-line error message that pops up on the screen. You can provide this information by inserting a reference to an **html_document** below the **message** line in your VRSL code, as shown in the following example:

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

Store the **doc_policy.html** file in the $LEDA_PATH/doc/html directory. For more information about HTML error reporting, see the *Leda User Guide*.

When this rule is violated, the user can hyperlink to the specified HTML document, where more information is available. Notice that the format includes an optional anchor within the HTML document (**#G_5_5_1_1**) to the specific location of the reference.

Hint

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.

Requiring Synchronous Resets in Flip-Flops

For our next example, let's 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 result in specific hardware features when the description is synthesized. The following code shows how we can use a **force** command to write this rule:

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

In this rule, we use a **force** command to make sure that all VHDL source code in our design that matches the primary template or model **flipflop** is constrained by the attribute **synchronous_reset** to make sure only synchronous resets are used. This is an example that shows the flexibility or dual nature of certain attributes. The primary template **flipflop** has an attribute called **synchronous_reset** that is a **template** kind of attribute, meaning that you can also use **synchronous_reset** as a primary template.

Look up both the **flipflp** and **synchronous_reset** primary templates in the *VRSL Reference Guide* to see how this works. Many of the attributes of the VRSL templates can also function as templates themselves. If this is the case, the attributes are identified in the tables that describe each VRSL template as being the **template** kind. Attributes that are the **local** kind can only be used with no and force commands with the primary template where they are described.

Ignoring Alias Declarations

In this example, we'll use a **no** command without context. We can do this because **alias_declaration** is a primary template that does not require a 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
```

Adding Context to Rules

In this next example, we'll add context to the command. This rule requires that **process_statements** be ignored in the context of **entity_declarations**. Note that **process_statement** is a primary template that does not require a context, but we can add one if we want. The context simply narrows the focus of the rule, in this case to **entity_declarations**. The following code shows how we can write this rule:

```
Example_6:
no process_statement in entity_declaration
message "Process statements are ignored in entities"
severity WARNING
```

VHDL Test Code

The following code demonstrates the use of this rule to constrain VHDL. As an additional exercise, create both a rule and VHDL code using these examples and run the Checker.

```
------------------------------------------------------------------------
-- 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
             );
BEGIN
       P1: PROCESS (reset_n) -- Rule fires here

       BEGIN
           assert reset_n = '0' report "Reset active";
       END PROCESS P1;
END example_6a_en;
```

Requiring that Default Port Values be Ignored

Now, write a rule that requires that default port values be ignored.

Hint

The context for this next rule is a primary template that can be found in the *VRSL Reference Guide*.

Fill in your answer below:

```
no default in _________________________
message "Port default values are ignored"
severity WARNING
```

Solution

```
Example_6a:
no default in port_declaration
message "Port default values are ignored"
severity warning
```

Note that the **port_declaration** template is aptly named. All VRSL primary and secondary templates are given meaningful names to make it easier for you to find and use the templates that you need.

Prohibiting Latches

The use of latches is generally considered poor coding practice. We'll use this next example to write a rule that globally prohibits latches in VHDL code using a **no** command. Since a latch is hardware, this is another example of a hardware inference rule similar to the one we saw in Example 3:

```
Example_7:
no latch
message "Avoid using latches in design"
severity ERROR
```

In this example, we did not need to provide a context because **latch** is a primary template and we want to globally prohibit latches in the design.

Prohibiting XNOR Binary Operators

This example uses a **no** command with a template to prohibit 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
```

In the template definition, we define the string NO_XNOR to be a template of type **binary_operation** and use a **limit** command to focus just on the attributes of type **operator_symbol** that match STD.STANDARD.XNOR.

In the next code segment, we call our NO_XNOR template, which will match any **operator_symbols** found in **binary_operations** that match STD.STANDARD.XNOR, and use a **no** command to prohibit their use in that context. As an added exercise, write a rule that prohibits the use of the XOR binary operator.

Prohibiting Expressions in Attribute Names

In this next example, we'll use a **no** command to write a rule that prohibits the use of expressions in attribute names:

```
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
```

The string ATTR_NAME is a name of our choosing that we assign to be of template type **attribute_name**, which is a primary template that requires no context. But we want to focus our template just on **expressions**, so we use a **no** command to restrict the scope of our ATTR_NAME template or model to **expression,** which is one of the attributes of the **attribute_name** primary template.

In the next code segment, we use a **limit** command to make sure that all code that matches our ATTR_NAME template or model does not contain **expressions**.

Limiting Clocks to One Name

Suppose we want to limit all clocks in our design to one name (for example, “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
```

First, we define the CLOCK_WITH_ID string be a template of type **clock**, which is a primary template that requires no context. But we want our template or model to find just **identifier**, which is one of the attributes of the **clock** template. So we use a **limit** command in the template definition to narrow the focus to **identifier**, and specify the string (“clk”) that we want to restrict clock names to.

In the next code segment, we use a **limit** command in the rule definition to make sure all code in our design that matches our CLOCK_WITH_ID template or model is limited to the name that we specified (“clk”).

Note that the template definition alone would not do the job. First you define a template using the templates and attributes that are available, and then you write the rule that uses your template definition. Templates are flexible in this way. You can use templates along with VRSL commands like **limit** to write very specific rules with the general-purpose templates or models of VHDL code.

Forcing Process Statements to be Combinatorial

In this example, we’ll use a **limit** command to write a rule that forces process statements to be combinatorial. 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
```

First, we define the string PSS_COMBINATIONAL to be a template of type **process_statement**, which is a primary template that requires no context. But we want our template or model to find just **combinatorial**, which is one of the attributes of the

process_statement template. Since we want to make sure all process statements use combinatorial processes, we use the **force** command in our template definition to provide a context and narrow the focus just to that.

The next code segment is our rule definition. We use a **limit** command to make sure that all **process_statements** in our design that match our PSS_COMBINATIONAL template have only **combinatorial** logic.

VHDL Test Code

The following code demonstrates the use of this rule to constrain VHDL. As an additional exercise, create both a rule and VHDL code using these examples and run the Checker.

```
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_en is

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

begin

Data_input: process  -- Rule fires 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  -- Rule fires 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;
```

Using Variables to Establish Naming Conventions

In this next exercise, we use a VRSL variable (<**entity**>) and a **limit** command to write a rule requiring that 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
```

Notice the use of the **<entity>** variable with this **limit** command. You can use UNIX regular expressions like this in VRSL for naming conventions, in particular. In the above example, Leda replaces the **<entity>** variable in the regular expression **<entity>.vhd** with the name of the enclosing entity. So, a file name of foo.vhd in the entity declaration would cause this rule to fire the Checker, unless the entity itself was named foo. For more information on using UNIX regular expressions and variables in VRSL rules, see the *VRSL Reference Guide*.

VHDL Test Code

The following code demonstrates the use of this rule to constrain VHDL. As an additional exercise, create both a rule and VHDL code using these examples and run the Checker.

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

entity example_12_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_12_en;
```

Using Multiple Templates

In this next exercise, we want to prohibit the use of literals in signal assignment statements. This will require five templates, as shown in the following example:

```
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,
                                                INTEGER_0,
                                                INTEGER_1
message "No literals in signal assign. statements-- use constants"
severity WARNING
```

Notice how all the templates are called from a single **limit** command.

Constraining Prefixes for Active-High and Active-Low Resets

Suppose we want to constrain the prefixes for active-high and active-low resets. Again, we'll need multiple templates (in this case four), and we'll need to use some conditional logic with **limit** commands, which are the only VRSL commands that can use conditional statements. To write this rule, we first develop four templates that define the following:

- high asynchronous reset edges
- high asynchronous names
- low asynchronous reset edges
- low asynchronous names

We then test for the conditions defined in the templates and if met, the rule is implemented with the appropriate **limit** command.

First, let's set up the templates. We know we need at least two templates for this rule, since we are dealing with active-high and active-low resets. We can outline the basic template structure as follows:

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

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

Actually, we'll need four templates to get the job done. Try the following exercise before you see how the template code is written.

Since four templates are needed, we know that each reset (high or low) will have two templates. For each reset, we need to define the edge type and naming convention. Sketch out the four templates using the *VRSL Reference Guide* as a reference.

Hint

Substitute asynchronous_reset for **A** in the code above.

If you did the exercise, 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. We need conditional **limit** commands 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 know that **A** for the templates and the **limit** commands is **asynchronous_reset**. Filling in the VRSL commands is now all we have left to complete the code.

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

template PSS_WITH_HIGH_ASYNCH_RESET is process_statement
  force asynchronous_reset
  limit asynchronous_reset to HIGH_ASYNCH_RESET
end

template PSS_WITH_LOW_ASYNCH_RESET is process_statement
```

```
  force asynchronous_reset
  limit asynchronous_reset to LOW_ASYNCH_RESET
end

-------------------------------------------------
-- Command section
-------------------------------------------------

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

limit process_statement to PSS_WITH_HIGH_ASYNCH_RESET,
PSS_WITH_LOW_ASYNCH_RESET
  severity NOTE

    if PSS_WITH_HIGH_ASYNCH_RESET then
         Example_14:
           limit asynchronous_reset to HIGH_ASYNCH_RESET_NAME
           message "Active high resets should be prefixed with rst"
           severity WARNING
    end if

    if PSS_WITH_LOW_ASYNCH_RESET then
         Example_14:
           limit asynchronous_reset to LOW_ASYNCH_RESET_NAME
           message "Active low resets should be called rst_n"
           severity WARNING
    end if

end ruleset
```

When you review this example, pay particular attention to the overall structure of the code. Conditional **limit** commands are one of the most powerful tools in VRSL.

Using Enumerated Types

To set an attribute to a value using an enumerated type, we'll need to use a **set** command. Suppose we want to make sure that "for" loops have globally static bounds. The following code shows how we can use a **set** command to 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
```

Note that some attributes have a precise meaning that is represented by an enumerated type or a string such as **Globally_Static_Evaluation**. For example, the **evaluation_time** attribute of the **for_loop_statement** primary template in this example indicates that the expression should be globally static (available to the whole design and

not changing) as opposed to locally static (available only to the enclosing process and not changing). This is done through the built-in enumerated type **Globally_Static_Evaluation**. This feature only works with **set** commands.

Prohibiting the use of Real Literals

If we want to do something like prohibiting the use of real literals, we'll again have to use a **set** command, as shown in the following example:

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

Example_16:
no BAD_LIT in literal
message "Real literals are not allowed"
severity ERROR
```

In this example, we assign the string BAD_LIT to be a **template** of type **literal**, which is a primary template. Then we focus our template just on value types using the **value_type** attribute of the **literal** template and a **set** command to set this attribute to **real_literal_type**.

In the next code segment, we use a **no** command to prohibit **literals** in VHDL code that matches our BAD_LIT template.

Limiting the Number of Clocks in Processes

Suppose we want to limit the number of clocks in processes to one. We can accomplish this using a **max** command, as shown in the following example:

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

In this example, we simply set the **clock_expression_count** attribute of the **process_statement** primary template to a maximum value (**max**) of 1. When Leda finds VHDL code that matches the **process_statement** primary template, it issues an error message if it finds more than one clock expression.

VHDL Test Code

The following code demonstrates the use of this rule to constrain VHDL. As an additional exercise, create both a rule and VHDL code using these examples and run the Checker.

```
library IEEE;
use     IEEE.std_logic_1164.all;
entity example_17_en is
port ( clk      : in std_logic;
       clk1     : in std_logic;
       data_in  : in std_logic;
       data_out : out std_logic
     );
 end;

architecture RTL of example_17_en is

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

begin

Data_input: process (clk)  -- Rule does not fire here

begin
      if (clk'event) and clk = '1' then
            ff_in <= data_in;
      end if;
end process Data_input;

comb <= not ff_in;

Data_output : process (clk, clk1)  -- Rule fires 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 <= ff_out;

end RTL;
```

Using Duplicate Rule Labels and Messages

This example shows how to use the same 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 for the **dimension_count** attributes look identical, but a closer look reveals that their contexts are different. The first rule uses the **dimension_count** attribute of the **unconstrained_array_definition** primary template and the second rule uses the **dimension_count** attribute of the **unconstrained_array_definition** primary template.

Inheriting Templates

VRSL allows you to use one template that inherits the characteristics of another template that you already defined. To show how this works, suppose we want to restrict entity names to 20 characters. We can do this with template inheritance and the **max** and **limit** commands, as shown in the following example:

```
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"
severity ERROR
```

Notice how the SHORT_NAMED_ENTITY template inherits from the first template (SHORT_ENTITY_ID). You can use this inheritance process for any of the VRSL commands.

Using Multiple Commands in One Template

The following example shows how we can combine **max** and **min** commands in a single template. This rule uses a **limit** command to set 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
```

This concludes our tutorial on learning how to use VRSL to write custom rules for constraining VHDL designs. For complete reference information on VRSL, see the *VRSL Reference Guide*.

2
Writing Rules for Verilog

Introduction

Welcome to the *Leda Rule Specifier Tutorial*, an example-based introduction to learning how to write coding rules for use with Leda. You use coding rules to check your Verilog designs for errors or anomalies that may cause problems for downstream tools in the design and verification flow. For general information about Leda, see the *Leda User Guide*.

You need an optional Specifier license in order to perform the exercises in this tutorial. The tutorial is divided into two parts. Part one takes you through the process of writing some sample rules and organizing them so that you can check some test Verilog code using Leda, and later fix the problems right from the tool. This part of the tutorial is organized in the following major sections:

- "Writing Rules" on page 54
- "Creating Ruleset Files" on page 55
- "Creating Policies" on page 58
- "Creating a Verilog Test File" on page 59
- "Creating a Project File" on page 60
- "Running the Checker & Fixing the Errors" on page 62

Part two explores the syntax and semantics of VeRSL, the Verilog rule writing language. Hands-on examples are provided there to give you a feel for all of the VeRSL commands and capabilities. This part of the tutorial is organized in the following major sections:

- "What is VeRSL?" on page 67
- "About Templates and Attributes" on page 67
- "VeRSL Rule-Writing Examples" on page 69

Writing Rules

In this exercise we'll create four new rules in the Verilog rule specification language (VeRSL) and organize them into two different rulesets that reside in one ruleset.sl file. Then we'll write some Verilog code, which the rules will check. Finally, we'll use the rules we created to check our sample Verilog code. Note that for this first exercise, we'll create some relatively simple rules so that you can get a feel for rule creation, compilation, checking, and debugging. After you master this flow, you can proceed to the second part of this tutorial to learn how to use all of the VeRSL commands to develop more complicated or sophisticated rules.

If you haven't already done so, install the Leda software and configure your environment as described in the *Leda Installation Guide*. Then, begin by invoking the Specifier tool as follows (you use the same tool for both Verilog and VHDL):

```
% $LEDA_PATH/bin/leda -specifier &
```

The Specifier main window (Figure 5) opens:

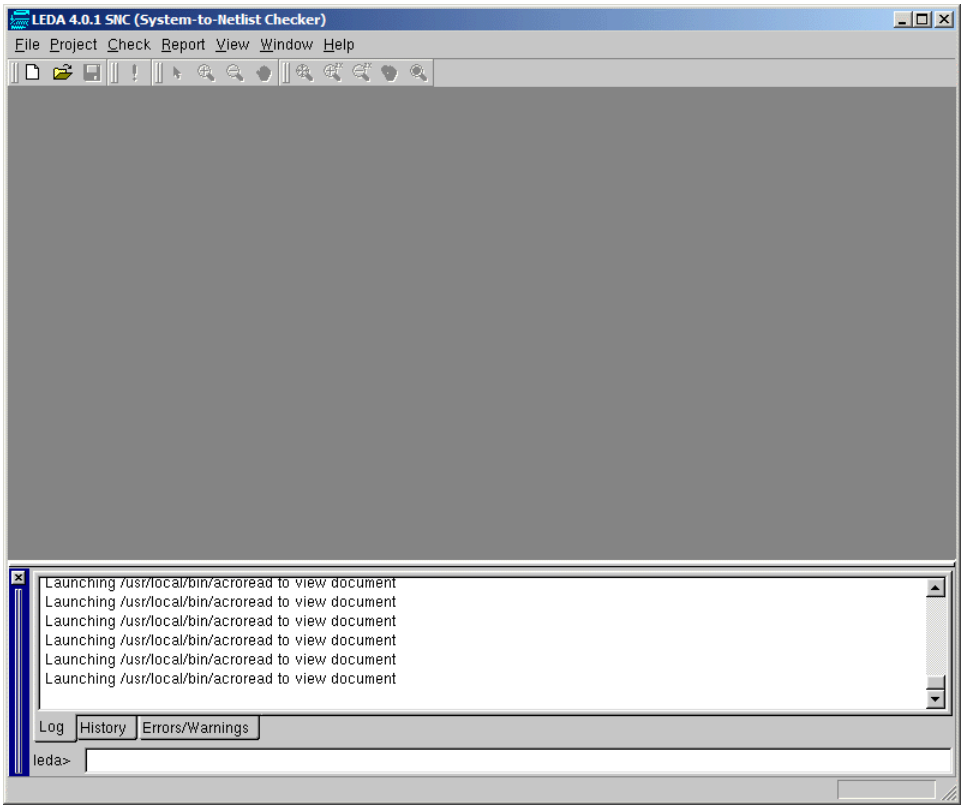

Figure 5: Specifier Main Window

Creating Ruleset Files

Suppose we need four new rules that:

- Make sure latches are not used in the design
- Catch missing or redundant signals in sensitivity lists
- Make sure that clocks are named “clock” or end in “_clk”
- Catch cases where clocks are mixed with different clock edges

For the sake of this exercise, let’s assume that we were unable to find rules to handle these needs in any of the prepackaged rules that come built-in with the Leda Checker. (In fact, there are rules similar to these that you can copy and modify.) To begin, follow these steps:

1. Using a text editor, type in the following VeRSL source code exactly as shown. You can use the text editor in the Specifier by choosing **File > New**. (Note that two dashes “--” at the beginning of a line designate a comment.) Create the file as “ruleset.sl”. Note that “.sl” is the standard extension for Verilog ruleset files.

Hint

For your convenience, you can find the following example VeRSL code in the $LEDA_PATH/doc/tutorial_specifier/rsl/ruleset.sl file. If you are viewing this document online, you can also cut-and-paste the text right from this PDF file.

```
------------------------------------------------------------
-- Verilog rules
------------------------------------------------------------
-- In this file we create 2 rulesets, with 2 rules in each:
-- TUTOR_RS
--   TUT_1  Avoid using latches in design
--   TUT_2  Missing or redundant signals in sensitivity list
-- TUTOR_CLOCK
--   TUT_3  Clock name must end with _clk or be clock
--   TUT_4  Avoid mixing clock with different clock edge
--
-- Rules were copy/pasted and adapted from existing Leda rules found
-- in the RMM coding guide policy.
-- The source of this policy for Verilog is in $LEDA_PATH/rules/rmm/
-- RMM.sl
-- For each rule, you must copy the "command" part of the rule
-- as well as the "template" part.
------------------------------------------------------------
```

```
ruleset TUTOR_RS is

------------------------------------------------------------
--   Command section
------------------------------------------------------------
-- Rule TUT_1
-- Avoid using latches in design
-- Copied from rule R_552_1 in $LEDA_PATH/rules/rmm/RMM.sl
-- Here, we just changed the label from R_552_1 to TUT_1
-- and commented out the link to a html document
---------------------------
TUT_1:
no latch in always_construct
  message "Avoid using latches in design"
--  html_document "pol_rmm.html#R_552_1"
  severity ERROR

------------------------------------------------------------
-- Rule TUT_2
-- Missing or redundant signals in sensitivity list
-- Copied from rule R_554_1 in $LEDA_PATH/rules/rmm/RMM.sl
---------------------------
TUT_2:
force complete_sensitivity in always_construct
  message "Missing or redundant signals in sensitivity list"
--  html_document "pol_rmm.html#R_554_1"
  severity ERROR

end ruleset

------------ end of ruleset  TUTOR_RS.  ------
ruleset TUTOR_CLOCK is

------------------------------------------------------------
--   Template section
------------------------------------------------------------
-- Template for use with rule TUT_3 below
-- Copied from the template used by G_521_6
-- Here, we changed the regular expressions to
-- reflect our own new naming convention

template CLOCK_ID is identifier
limit limit_id to "_clk$", "clock"
end

template CLOCK_WITH_ID is clock
limit identifier to CLOCK_ID
end
```

```
------------------------------------------------------------
--   Command section
------------------------------------------------------------
-- Rule TUT_3
-- Clock name must end with _clk or be clock
-- Adapted from rule G_521_6 in $LEDA_PATH/rules/rmm/RMM.sl
-- Here, we changed the message, and the regular expression
-- used in the template CLOCK_NAME
---------------------------
TUT_3:
limit clock to CLOCK_WITH_ID
  message "Clock name must end with _clk or be clock"
--  html_document "pol_rmm.html#G_521_6"
  severity WARNING

------------------------------------------------------------
-- Rule TUT_4
-- Avoid mixing clock with different clock edge
-- Adapted from rule G_541_1 in $LEDA_PATH/rules/rmm/RMM.sl
-- Here, we changed the message
---------------------------
TUT_4:
no mixed_clock in design
  message "Avoid mixing clock with different clock edge"
--  html_document "pol_rmm.html#G_541_1"
  severity WARNING

end ruleset
```

Creating Policies

Now that we have a ruleset.sl file that contains our new rules, we need to store the new rules in a policy. To do that, follow these steps:

1. From the Specifier main window, first open the Rule Wizard (**Check > Configure).** From there, open the Policy Manager window **(Tool > Policy Manager**).
2. Click the Verilog tab. Then click the New button on the right side of the display. Type in a name for the new policy (for example, “MY_TUTOR”) and click OK.
3. When your new policy name appears in the Policies pane, click it to highlight the name and then click in the Rulesets pane. Click the Add button. This opens the “Please choose a rule file” window.
4. Navigate to the location of the ruleset.sl file you just created and click on the file name. Then click the Add button. This causes the tool to compile your new rulesets. You should see messages in the Log tab at the bottom of the Specifier main window similar to the following:

   ```
   Compiling ruleset TUTOR_RS
   Compiling ruleset TUTOR_CLOCK
   Compilation done (block level).
   Compilation done (chip level).
   ```

   The Rulesets pane in the Policy Manager window now shows the two rulesets we created (TUTOR_CLOCK and TUTOR_RS), and the Templatesets pane shows the templatesets we used.
5. Close the Policy Manager window.

   You have now created a policy (“MY_TUTOR”) containing two rulesets that are both in the same ruleset.sl file. The rulesets contain a total of four rules (two rules each), which we’ll use to check some sample Verilog code.

Using Tcl Shell Mode to Create New Policies

You can also create new policies and rulesets using the rule_manage_policy command in Leda’s Tcl shell mode. For example, to create the “MY_TUTOR” policy, use the following command at the Tcl shell prompt:

```
leda> rule_manage_policy -policy MY_TUTOR create
```

Then, to compile the ruleset.rl file you created for this tutorial, use the following command at the Tcl shell prompt:

```
leda> rule_manage_policy -policy MY_TUTOR compile ruleset.sl
```

For more information on using Tcl shell mode, see the *Leda User Guide*.

Creating a Verilog Test File

To test these rules, we need to create some Verilog code to check them against. We are going to purposely violate all four rules we created in our Verilog code so that we can see how the tool works. When we check the code with our new rules, we should see errors for all of them. Follow these steps:

1. Using a text editor, type in the following text, and save the file as test.v.

```
// Simple testcase for the rules in the Verilog policy built with
// ruleset.sl file
// Will fire these rules:
//   TUT_1  Avoid using latches in design
//   TUT_2  Missing or redundant signals in sensitivity list
//   TUT_3  Clock name must end with _clk or be clock
//   TUT_4  Avoid mixing clock with different clock edge

module test_mod (data, clk, main_clk, clock, rst, load_clk,
                 q, q1, q2, q3 );
input data, clk, main_clk, clock, rst, load_clk;
output q, q1, q2, q3;
reg   q, q1, q2, q3;

always @(data)
       begin
         if (load_clk)
// TUT_2 fires: load_clk is not in sensitivity list
            q <= data;
         else
            q <= 1'b0;
         end

always @(posedge clk or posedge rst)
//TUT_3 fires: bad naming for a clock
        begin
          if (rst == 1'b1  )
             q1 <= 1'b0;
          else
             q1 <= data;
          end

always @(negedge main_clk)
// TUT_4 fires: we mixed clock with different edges
        begin
   if (rst == 1'b1)
               q2 <= 1'b0;
          else
               q2 <= data;
          end
```

```
always @(load_clk or data)
       begin
         if (load_clk)
            q3 <= data;   // TUT_1 fires: we infer a latch here
         end

endmodule
```

Creating a Project File

Before we can use Leda to test our new rule against the sample Verilog file we created, we must first create a project file. A project file organizes the Verilog files into easily managed units. Follow these steps:

1. From the Specifier main window, choose **Project > New**. This opens the Project Creation Wizard window.
2. Click the Specify Project Name button. This opens the Specify Project Name window. Use the Browse button to navigate to the location where you want your project file to reside (for example, “WORK”), and enter the project name (for example “my_project”). Then click Save.
3. Now click the Next button at the bottom right of the window. This takes you to the Specify Compiler Options part of the Wizard, which has tabs for VHDL and Verilog. Click the Verilog tab.
4. In the Severity Level pane, click the radio button for the lowest severity level for which error messages from the Verilog compiler will be printed. Compiler messages with a severity below the specified value are not printed. (This severity level is only used for Verilog syntax analysis, not for checking.) The default is Warning.
5. In the Version pane, click the 95, 2001, or SystemVerilog radio button, depending on the version of Verilog you are using. The default is Verilog 95.
6. Click Next. This takes you to the Specify Libraries part of the Wizard, which has tabs for VHDL and Verilog. Click the Verilog tab.
7. In the Include Directories pane, specify the path to any directories to be searched for included files in your design. For this test, we don’t have any include files, so leave this pane empty.
8. In the Library Directories and Library Files panes, click the Add button and navigate to the location of any required source code libraries or files to be searched by the Verilog compiler in order to resolve unresolved module instances. For this exercise, we’re not using any resource directories or files, so leave these panes empty.

9. Click Next. This takes you to the Specify Source Files part of the Wizard. which has tabs for VHDL, Verilog, and All. (The All tab is for mixed-language designs.) Source files, in this case, means Verilog source files, the ones we want to check against our new rules. Click the Verilog tab.
10. In the Files pane, click the Add button. This opens the Add Files window. Navigate to the location of the test.v file we created to test the new rules. Highlight the file name and then click OK to confirm your selection. The full path to our Verilog source file (test.v) is now displayed in the Files pane window.
11. Click Next. This takes you to the Confirm & Create part of the Wizard. Leave the Build with Check checkbox selected and click Finish. If the tool displays a small Get Top Module/Design/Entity window, note that this information is needed for checking chip-level rules. For this exercise, leave these settings at their default values and click the OK button. Leda compiles the Verilog file and executes the Checker. You should see something like the following screen (Figure 6).

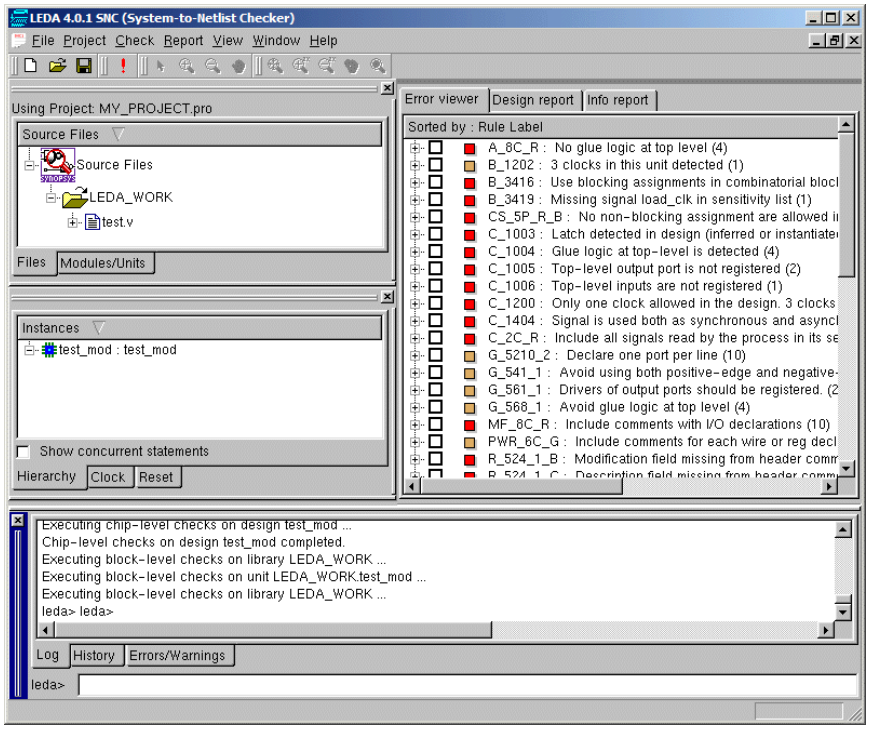

Figure 6: Specifier Project After Build

Note the test.v file in the Source Files pane on the left-hand side of the main window. This is the test file that we wrote earlier to check against the rules we created. But look at all the errors and warnings listed in the Error Viewer pane on the right-hand side of the main window. How many rules can you violate with one simple test file? As it turns

out, quite a few, but we can narrow the display to focus the results to just the new rules we created for this tutorial, as explained in the next section, Running the Checker & Fixing the Errors.

Running the Checker & Fixing the Errors

With the project built, we will now set up and run the Checker, which will check our sample Verilog code against the coding rules we created. Follow these steps:

1. From the Specifier main menu, choose **Check > Select**. This opens the Leda Rule Wizard window, which lists all the prepackaged policies that come with the tool on the left side of the window, in addition to the new policy we just created for this exercise (MY_TUTOR). Some of the prepackaged policies are activated by default. That's why we got so many error and warning messages from the one simple test.v file that we wrote for this tutorial. As you learn how to use Leda to check your HDL code, don't let the number of warning and error messages you receive throw you off. In many cases, changing one line of code eliminates lots of error messages all at once. And you can easily turn off rules that you don't consider to be significant for your design (see the section on Deactivating Rules in the *Leda User Guide).*
2. For now, deactivate all policies except MY_TUTOR by clicking the icons next to each policy name until the boxes appear empty. When you are done, only the MY_TUTOR box icon should be filled in and colored light blue to indicate that only the new rules that we wrote are now selected for checking.
3. Open the MY_TUTOR display by clicking the (+) icon so that we can get a look at the rules we created and how they are organized. Click the colored box icons to the left of the TUTOR_CLOCK and TUTOR_RS rulesets one at a time. Note how the

upper-right side of the window changes to display the rule labels and messages for each of those rulesets. The display should look similar to the following (see Figure 7).

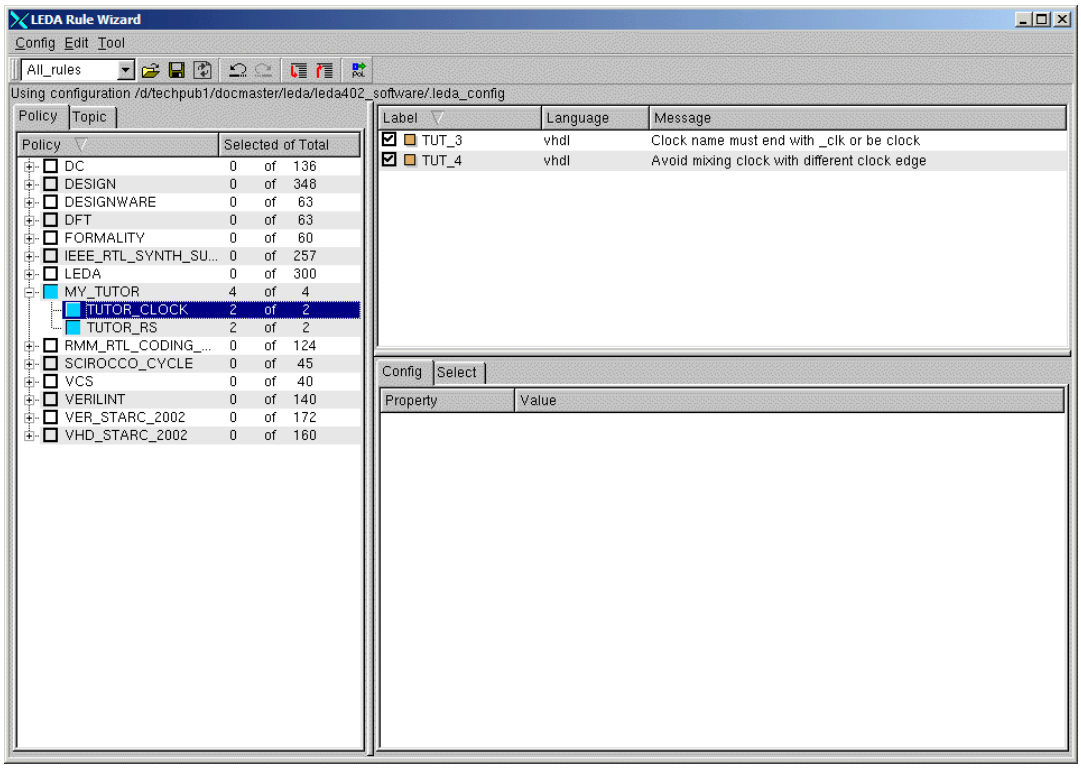

Figure 7: Rule Wizard with Custom Rules Displayed

4. Click the OK button.
5. From the Specifier main menu, choose **Check > Execute**. This time we see our same test file (test.v) listed in the Files tab on the left, but in the Error Viewer we see just four messages. They are the messages generated because the test file we wrote violates all four rules that we specified in the MY_TUTOR policy (see Figure 8).

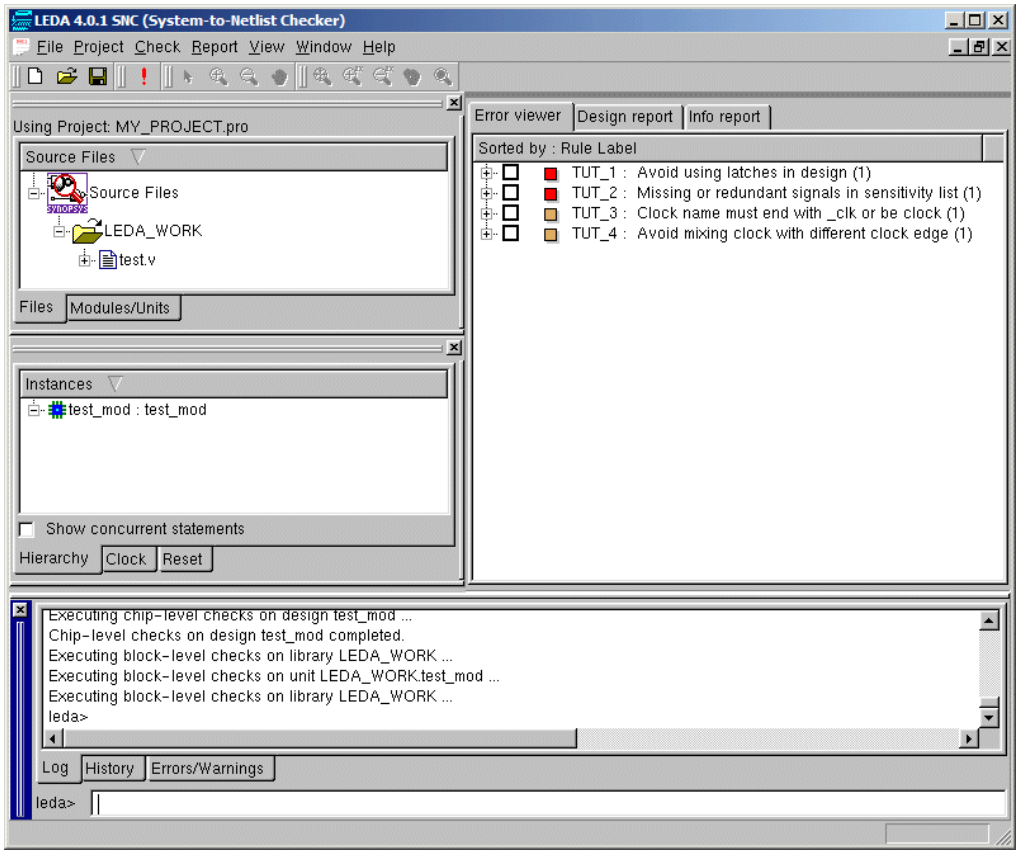

Figure 8: Checker Results for Custom Rules

6. For each warning or error message in the Error Viewer, click the (+) box icon to the left of the message. This expands the display to show the name of the test.v file that we tested. Click the next (+) box icon in the hierarchy. The display opens a window right on the Verilog file that was tested, with the offending line of code indicated by a green arrow pointer.
7. For each warning or error message, double click in the source code display in the Error Viewer. This opens a text editor on the file. The suspect code is already highlighted in the file. Correct the problems as shown in the following Verilog code, which is commented to show where the problems are and how you can fix them.

Hint

For your convenience, you can find a copy of this corrected test code at $LEDA_PATH/doc/tutorial_specifier/hdl/test_fixed.v.

```
// Simple testcase for the rules in the Verilog policy built with
//     ruleset.sl file
//  Show the corrections (just for demo purpose)
//  to avoid the firing of the rules:
//   TUT_1  Avoid using latches in design
//   TUT_2  Missing or redundant signals in sensitivity list
//   TUT_3  Clock name must end with _clk or be clock
//   TUT_4  Avoid mixing clock with different clock edge


module test_mod (data, clk, main_clk, clock, rst, load_clk,
                 q, q1, q2, q3 );
input data, clk, main_clk, clock, rst, load_clk;
output q, q1, q2, q3;
reg   q, q1, q2, q3;


//always @(data)
always @(data or load_clk)
       begin
         if (load_clk)
          // TUT_2 fires if load_clk is not in sensitivity list
            q <= data;
         else
            q <= 1'b0;
         end

//always @(posedge clk or posedge rst)
//TUT_3 fires: bad naming for a clock
always @(posedge clock or posedge rst)
// We just change to a "good" name for demonstration

        begin
          if (rst == 1'b1  )
             q1 <= 1'b0;
          else
             q1 <= data;
          end
```

```
//always @(negedge main_clk)
// TUT_4 fires: we mixed clock with different edges
always @(posedge main_clk)
// This will prevent TUT_4 from firing (just for demo...)
       begin
           if (rst == 1'b1)
               q2 <= 1'b0;
           else
               q2 <= data;
           end

always @(load_clk or data)
       begin
         if (load_clk)
            q3 <= data;
// Without the else clause, TUT_1 fires: we infer a latch here
         else
            q3 <= 1'b0;
         end

endmodule
```

8. When you are done fixing each problem, choose **File > Save** from the editor's pulldown menu to save your changes.
9. From the Specifier main window, choose **Project > Build**. The tool recompiles your test file.
10. From the Specifier main window, choose **Check > Run**. The tool checks your corrected Verilog code again using the rules you created. This time, since we corrected the offending code, our results come up clean, with no error messages listed in the Error Viewer.

This concludes our first exercise with the Leda Specifier tool. As a further exercise, copy some of your design team's Verilog files and try running some or all of the prepackaged policies against them to see what kind of results you get. Notice how even simple changes in your Verilog coding eliminate lots of error messages. This can help avoid downstream bottlenecks in your design and verification flow.

Now that we have the mechanics down for building rules using Leda, let's explore the syntax and semantics of the VeRSL rule specification language itself, beginning with the first section in part two of this tutorial, What is VeRSL?

What is VeRSL?

VeRSL is a macro-based language used for writing rules that check Verilog code for errors or anomalies that you specify using prebuilt templates and attributes, and a simple set of commands. There are six VeRSL commands: force, no, limit, set, max, and min. Each command has a precise syntax with allowed keywords. For complete reference information on VeRSL commands, templates, and attributes, see the *VeRSL Reference Guide.*

All terminology used for writing rules comes from either the IEEE Standard 1364-1995 Verilog Language Reference Manual (LRM) or the*VeRSL Reference Guide*. The LRM is the basis upon which the *VeRSL Reference Guide* was created. While a few of the terms used by Leda are unique, most of them can be found in the LRM. A review of the LRM will help you gain a better understanding of VeRSL.

About Templates and Attributes

Before we get started writing custom rules, let's take a closer look at templates and attributes, because these are the building blocks that you combine with VeRSL commands to write rules.

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

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

When you write a rule with a primary template, you don't need to provide a context. Primary templates are stronger than secondary templates in this way. For example, you can write a rule with the **limit** command and a primary template, such as **module_declaration** like this:

```
limit module_declaration to your_constraint
```

On the other hand, when you write a rule with a secondary template, you need to provide a context. Secondary templates are not as strong as primary templates in this way. For example, you can write a rule with the **limit** command and a secondary template such as **range** like this:

```
limit range in lsb_constant_expression to your_constraint
```

In this example, the **lsb_constant_expression** attribute provides the necessary context for the **range** secondary template.

Note

Each template in the VeRSL rule specification language is either primary or secondary. They are all clearly labelled in the *VeRSL Reference Guide*, which provides complete reference information for all templates and attributes, including the attribute name, kind, and limit_kind.

When you write rules, first you define a template, and then you call that template to complete the rule using VeRSL commands.

VeRSL Rule-Writing Examples

The remainder of this tutorial provides examples of how to write rules that check for common issues of concern in HDL designs. Most of the examples are designed to give you an introduction to the basics of rule writing using the different VeRSL commands. Other examples show you how to use different features of the language such as variables and template inheritance, so that you can see how they work. The examples are presented in the following sections:

Hint

For your convenience, you can find VeRSL source code for all the examples in this chapter in the $LEDA_PATH/doc/tutorial_specifier/rsl directory. The ruleset files that contain these examples are named example_1.sl. and so on, to match the example names used in the tutorial. You can also find Verilog source code for the tests used in these examples in the $LEDA_PATH/doc/tutorial_specifier/hdl directory. The .v files are named example_1.v and so on to match the examples where they are used.

Requiring that Module Ports be Named by Association

For our first example, supposed we want to ensure that all module ports in an instantiation are named by association rather than by position. The following code shows how we can use a **force** command to implement this rule:

```
Example_1:
force named_port_connection in module_instantiation
message "Map module ports in instantiation by named association, not
by position"
severity ERROR
```

In this example, we use the **named_port_connection** attribute of the **module_instantiation** primary template to **force** port connections to have named associations. Because **module_instantiation** is a primary template, we did not have to provide a context for this rule, but we wanted to limit the focus to a specific aspect of module instantiations, so we used the **named_port_connection** attribute to specify that context. If Leda finds Verilog code where module ports (in **module_instantiations**) are named by position, it flags an error.

Notice the **message** and **severity** lines in the code. The message line contains the text that is displayed when the rule is violated. The severity line indicates the level of the violation (note, warning, error, or fatal). If these lines of code are not present, Leda cannot flag a rule violation, so be sure to include them when you write rules.

Verilog Test Code

The following example shows how to use this rule to constrain Verilog. As an additional exercise, create both a rule and Verilog code using these examples and run the Checker.

```
module abc (a, b, c);
  input a, b;
  output c;
endmodule

module def (d, e, f);
```

```
  input d, e;
  output f;
endmodule
module example_1 (h, i, j, k, l, m);
  input h, i, j, k;
  output l, m;

  abc abc (h, i, l);      <<-------------- Leda points here
  def def (.d(j), .e(k), .f(m));
endmodule
```

The *VeRSL Reference Guide* defines **module_instantiation** as follows:

Primary template belonging to classes: CONCURRENT_STATEMENT (see Table 3).

Table 3: module_instantiation Primary Template Description

Attribute	Kind	Limit_Kind
instance_identifier	template	ID
module_identifier	template	ID
parameter_value_assignment	template	N/A
parameter_value_assignment	template	N/A
port_expression	template	EXPRESSION
range	template	range
port_reference_declaration	local	N/A
named_port_connection	**local**	**N/A**
complete_port_connection	local	N/A

In Table 3, the **Attribute** column lists the attributes that you can use with the **module_instantiation** primary template. The **Kind** column tells you how those attributes can be used. If the attribute is a **template** kind, this means that it can be used as a template itself with its own context when writing limit, no, and force commands. For example, all statements and declarations are **template** kinds of attributes. The template kind of attribute is flexible and powerful in this way.

If the attribute is a **local** kind, it can only be used with no and force commands, and cannot function as a template. The **Limit_Kind** of an attribute tells you the type of template or context the attribute can be constrained to. If the **Limit_Kind** column says N/A (see Table 3), this means that the attribute is not constrained to a particular type of Verilog construct or template.

For example, the **named_port_connection** attribute in our example is used as a **local** kind of attribute and is not constrained to use with a particular type of template or Verilog construct. In this case, we used the **named_port_connection** local attribute of the **module_instantiation** primary template to define the rule requiring that all ports in module instantiations be named by association:

```
Example_1:
force named_port_connection in module_instantiation
message "Map module ports in instantiation by named association, not
by position"
severity ERROR
```

Requiring Synchronous Resets in Flip-Flops

For our next example, let's write a rule requiring that synchronous resets be present in all flip-flops. We can again use a **force** command to implement this rule, as shown in the following example:

```
Example_2:
force synchronous_reset in flipflop
message "Flip flops with synchronous resets only are allowed"
severity ERROR
```

In this example, we use the **synchronous_reset** attribute of the **flipflop** primary template to focus our **force** command just on the items of interest.

Adding HTML Help Files for Errors

Sometimes users need more explanation of a rule violation than the single-line error message that pops up in the Error Viewer pane of the Checker GUI. To provide this supplementary information, you can insert an HTML reference document below the message line as follows:

```
force synchronous_reset in flipflop
message "Flip flops with synchronous resets only are allowed"
html_document "sync_reset.html#G_5_5_1_1"
severity ERROR
```

When this rule is violated, users can hyperlink to the HTML document specified by **html_document**, where more information is available. Notice that the format includes an optional anchor within the HTML document (**#G_5_5_1_1**) to the specific location of the reference.

Store the **sync_reset.html** file in the $LEDA_PATH/doc/html directory. For more information about HTML error reporting, see the *VeRSL Reference Guide*.

Hint

Using the very first exercise in this tutorial, add an **html_document** reference line to your code and recompile. For the reference document, either create a simple HTML document or just reference a known good HTML address. Then access the HTML help document right from the Error Viewer after you run a check that violates the rule.

Hardware Semantics

The terms synchronous_reset and flip-flop can be found in the *VeRSL Reference Guide* as primary templates. However, neither of these terms is present in the LRM. This is because the concept of a synchronous reset is not explicit in Verilog. It is inferred by synthesis tools from Verilog code written in a certain way. This kind of inference is an example of the hardware semantics of Verilog. VeRSL has attributes that allow these hardware semantics to be constrained in a way similar to the language semantics. Remember that some terms may be found in the *VeRSL Reference Guide* that are not in the LRM.

Requiring Port Connections

Suppose we now want to make a slight modification to the rule in Example 1 and write a rule requiring that all ports be connected. Before showing you how to write this rule, try the following exercise using the code from the first example to help you.

Write a rule requiring that all ports be connected.

Hint

Look up module_instantiation in the *VeRSL Reference Guide* and find the appropriate local attribute. Fill in your answer below. You can use Example 1 as a reference.

```
force ___________________________ in module_instantiation
message "All ports must be connected"
severity ERROR
```

The following code shows how we can use a **force** command to write this rule.

```
Example_3:
force complete_port_connection in module_instantiation
message "All ports must be connected"
severity ERROR
```

In this example, we use the **complete_port_description** attribute of the **module_instantiation** primary template and a **force** command to make sure code that matches our focused model of Verilog code has all ports connected.

Verilog Test Code

The following code shows the use of this rule to constrain Verilog. As an additional exercise, create both a rule and Verilog code using these examples and run the Checker.

```
module def (d, e, f);
  input d, e;
  output f;
endmodule
module example_3 (h, i, j, k, l, m);
  input h, i, j, k;
  output l, m;

  def def_1 (.d(j), .e(k));       <<-------------- Leda points here
  def def_2 (.d(j), .e(k), .f(m));
endmodule
```

Ensuring Complete Sensitivity Lists

In this example, we'll write a rule that uses the **always_construct** primary template, which is one of the basic VeRSL templates. The **always_construct** template comes directly from Verilog's always statement and is one of the primary constructs used in Verilog coding. Let's write a rule to make sure that all combinatorial logic in always constructs has a complete sensitivity list. The following code shows how we can use a **force** command to write this rule:

```
template COMBINATIONAL_ALWAYS is always_construct
   force combinatorial
end

limit always_construct to COMBINATIONAL_ALWAYS severity NOTE


if COMBINATIONAL_ALWAYS then

Example_4:
force complete_sensitivity in always_construct
message "Missing/redundant signals in sensitivity list of a
combinatorial block"
severity ERROR
end if
```

First, we define the string COMBINATIONAL_ALWAYS to be a template of type **always_construct**, which is a VeRSL primary template, Then we narrow the scope of our template definition to the **combinatorial** local attribute of that template using a **force** command, Thus, our template is only looking for Verilog code in always blocks that deals with combinatorial logic.

In the next section of code, we **limit** always constructs in our Verilog code to our COMBINATIONAL_ALWAYS template definition. If Leda finds an always block with clock-based sequential logic, it issues a note message, per the severity NOTE statement at the end of that line of code.

Next, we use some conditional logic to further constrain Verilog code that matches our template. If Leda finds **always_constructs** that match our COMBINATIONAL_ALWAYS template, it **forces** them to have a complete sensitivity list, using the **complete_sensitivity** attribute of the **always_construct** primary template. This rule causes Leda to flag an error if it finds combinatorial logic in always constructs that do not have complete sensitivity lists.

Verilog Test Code

The following code demonstrates the use of this rule to constrain Verilog. As an additional exercise, create both a rule and Verilog code using these examples and run the Checker.

```
module example_4 (clk, a, b, foo, c, e);
  input a, b, foo, clk;
  output c, e;
  reg c, d;

always @(a or b)    // Here, the sensitivity list is OK
   c <= a & b;

always @(a or foo) // - Leda points here--foo is redundant
  begin
   d <= a & b;      // - Leda points here--b is missing from the list
   c <= d;          // - Leda points here--d is missing from the list
  end

always @(posedge clk) // Just get a NOTE message here--it's sequential
block
  begin
   d <= b & a;
   c <= d;
  end

endmodule
```

As a further exercise, look up the always_construct in both the LRM and the *VeRSL Reference Guide*. Consider the similarities and differences in the terminology used in each manual. Also, study the syntax and structure of the basic template used in this example. Understanding templates is one of the keys to learning VeRSL.

Prohibiting Macromodules in Module Declarations

In this example, we want to constrain the Verilog code so that **macromodule** is not used in module declarations. The LRM says that either module or macromodule can be used in module declarations, but most coding conventions recommend using module. The following code shows how we can use a **no** command to implement this rule:

```
Example_5:
no macromodule in module_declaration
message "Macromodules are not supported for synthesis"
severity ERROR
```

In this example, **module_declaration** is a primary template, so it does not require a context. But we want our rule to just look for macromodules and signal an error when it finds them, so we use the **macromodule** local attribute of the **module_declaration** primary template to narrow the focus of our rule to just macromodules and the **no** command to prohibit their use.

Verilog Test Code

The following code demonstrates the use of this rule to constrain Verilog. As an additional exercise, create both a rule and Verilog code using these examples and run the Checker.

```
macromodule example_5 (in1, in2, out1);  <<-----  Leda pointa here
  input [3:0] in1, in2;
  output [4:0] out1;
  assign out1 = in1 + in2;
endmodule
```

Prohibiting Bidirectional Ports

This example shows how to use a **no** command to prohibit the use of bidirectional ports. See if you can use the previous rule as a guide to writing this rule, as described in the following exercise.

Write a rule that prohibits the use of bidirectional ports in Verilog code.

Hint

A bidirectional port corresponds to an inout_declaration in VeRSL. Fill in your answer below.

```
no ____________________________ in module_declaration
message "No bi-directional ports are allowed in the design"
severity ERROR
```

The following example shows how we can write this rule:

```
Example_6:
no inout_declaration in module_declaration
message "No bi-directional ports are allowed in the design."
severity ERROR
```

In this example, **module_declaration** is again a primary template, so it does not require a context. But we want our rule to look only for bidirectional ports and signal an error when it finds them, so we use the **inout_declaration** local attribute of the **module_declaration** to narrow the focus of our rule just to bidirectional ports and the **no** command to prohibit their use.

Verilog Test Code

The following code demonstrates the use of this rule to constrain Verilog. As an additional exercise, create both a rule and Verilog code using these examples and run the Checker.

```
module example_6 (ab, aBc, ad, aBc12, ad_3, DBC);
  input [3:0] ab;
  input aBc ;

  inout aBc12;       <<------------  Leda points here
  inout ad;           <<------------- Leda points here
  output ad_3;
  output [2:0] DBC;
endmodule
```

Prohibiting Latches

You use the **all** keyword to apply a command or rule globally throughout your design. In this next example, we'll write a rule that globally prohibits latches in Verilog code:

```
Example_7:
no latch in all
message "Avoid latch inferring in your design"
severity ERROR
```

In this example, we use the **latch** primary template, which does not require a context. In our earlier examples, we narrowed the focus of primary templates anyway using attributes of the primary templates because we wanted our templates to match only particular aspects of the Verilog constructs represented by the templates. In this case, we use a new feature of VeRSL, the **all** keyword, to instead widen the scope of our rule, and the **no** command to globally prohibit latches in the design.

Here are some other examples for how to use the **all** keyword that are similar to the previous example. The only difference is that they globally prohibit different Verilog constructs, using the **repeat_statement** and **while_statement** primary templates as models of the code that we want to constrain.

```
Example_7a:
no repeat_statement in all
message "Repeat loop statements are not supported for synthesis"
severity ERROR
```

```
Example_7b:
no while_statement in all
message "While loop statements are not supported for synthesis"
severity ERROR
```

Note

If no context is present in a command, the default is all.

Prohibiting Case Equality Operators

Suppose we want to prohibit the use of the case equality operator "===" in our design. To write this rule, we first create a template that uses a **limit** command to focus on the case equality operator and then use a **no** command in our rule to constrain the Verilog code according to the template:

```
template CASE_EQUALITY_BIN_OP is binary_operation
    limit operator_symbol to "==="
end

Example_8:
no CASE_EQUALITY_BIN_OP in binary_operation
message "The case equality operator '===' is not supported in binary
operations"
severity ERROR
```

In the template definition, we define the string CASE_EQUALITY_BIN_OP to be a template of type **binary_operation**, which is a primary template that does not require a context. But we want to narrow the focus of our template to a particular operator symbol, so we use a **limit** command and the **operator_symbol** attribute of the **binary_operations** primary template to make our CASE_EQUALITY_BIN_OP template or model match just case equality operators.

In the next code segment, we call our CASE_EQUALITY_BIN_OP template and use the **no** command to prohibit the use of case equality operators in binary operations in our Verilog code.

Verilog Test Code

The following code demonstrates the use of this rule to constrain Verilog. As an additional exercise, create both a rule and Verilog code using these examples and run the Checker.

```
module example_8 (in1, in2, out1);
  input [3:0] in1, in2;
  output [4:0] out1;
  assign out1 = in1 === in2;      <<-------- Leda pointa here
endmodule
```

As a further exercise, write a rule that prohibits the use of the "+" symbol in binary operations.

Limiting Clocks to One Name

Suppose we want to limit all clocks in our design to one name (for example, “clk”). The following code shows how we can use a **limit** command to write this rule:

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

Example_9:
limit clock to CLOCK_WITH_ID
message "A clock signal should be called 'clk'"
severity ERROR
```

In this example, we first define the string CLOCK_WITH_ID to be a template of type **clock**, which is a primary template that does not require a context. But we want to narrow the focus of our template to just identifiers, so we use a **limit** command in the template definition along with the **identifier** attribute of the **clock** primary template to make our template or model match just identifiers named “clk”.

In the next code segment, we call our CLOCK_WITH_ID template and use a **limit** command to make sure all clocks in our design match our template’s definition of clock names. When you run this rule on your Verilog code, Leda flags an error if it finds any clocks not named “clk”.

Verilog Test Code

The following code demonstrates the use of this rule to constrain Verilog. As an additional exercise, create both a rule and Verilog code using these examples and run the Checker.

```
module example_9a (D, my_clk, rst_n, Q);
input D, my_clk, rst_n;
output Q;
reg Q;

always @(posedge my_clk or posedge rst_n) // <<--- Leda pointa here
  begin
    if (rst_n)
       Q = 0;
    else
       Q = D;
  end
endmodule

module example_9b (D, clk, rst, Q);
input D, clk, rst;
output Q;
reg Q;
```

```
always @(posedge clk or posedge rst)
  begin
     if (rst)
       Q = 0;
    else
       Q = D;
  end
endmodule
```

Requiring Instance Names with “U_”

Here’s a slight variation on the previous example. In this example, we’ll write a rule requiring that instance names begin with “U_”:

```
template MOD_INSTANTIATION_ID is identifier
    limit limit_id to "^U_"
end

Example_10:
limit instance_identifier in module_instantiation to
MOD_INSTANTIATION_ID
message "Instance names should begin with 'U_'"
severity WARNING
```

In this example, we first define the string MOD_INSTANTIATION_ID to be a template of type **identifier**, which is a primary template that does not require a context. But we want to narrow the focus of our template or model to just limit IDs, so we use the **limit** command in the template definition along with the **limit_id** attribute of the **identifier** primary template to make our template or model match just limit IDs named “U_”.

In the next code segment, we call our MOD_INSTANTIATION_ID template and use a **limit** command to make sure all **instance_identifiers** in our design match our template’s definition of limit IDs. When you run this rule on your Verilog code, Leda flags an error if it finds any instance identifiers in module instatiations that do not begin with “U_”, as we specified.

Note the use of **identifier** as a primary template in this example. Compare this use to the previous example, where we used **identifier** as an attribute of the **clock** primary template. This is an example of the dual nature of some primary template attributes; they can function as primary templates or just as attributes of other primary templates, depending on your needs. VeRSL is flexible in this way. When you look up templates in the *VeRSL Reference Guide* to see what’s available, remember that when you see the word **template** in the **Kind** column for the various template definitions, this means that the attribute can also function as a template itself.

Verilog Test Code

The following code demonstrates the use of this rule to constrain Verilog. As an additional exercise, create both a rule and Verilog code using these examples and run the Checker.

```
module abc (a, b, c);
  input a, b;
  output c;
endmodule

module def (d, e, f);
  input d, e;
  output f;
endmodule
module example_10 (h, i, j, k, l, m);
  input h, i, j, k;
  output l, m;
  abc U_abc (.a(h), .b(i), .c(l));

  def def (.d(j), .e(k), .f(m)); <<------- Leda pointa here

endmodule
```

Using Multiple Commands in Templates

This next example introduces the concept of multiple lines of code in templates. Let's write a rule that forbids the use of initial constructs to initialize variables:

```
template INITIAL_WITHOUT_ASSIGNMENTS is initial_construct
    no non_blocking_assignment
    no blocking_assignment
    no procedural_continuous_assign
    no procedural_continuous_force
  end

Example_11:
limit initial_construct to INITIAL_WITHOUT_ASSIGNMENTS
message "Do not use initial constructs to initialize variables"
severity WARNING
```

In this example, we first define the string INITIAL_WITHOUT_ASSIGNMENTS to be a template of type **initial_construct**, which is a primary template that does not require a context. But we want to narrow the focus of our template to several specific Verilog constructs, so we use a series of **no** commands coupled with the **non_blocking_assignment**, **blocking_assignment**, **procedural_continuous_assign**, and **procedural_continuous_force** attributes of the **initial_construct** primary template to focus our rule just on these attributes of interest.

Note

Notice how you can widen or narrow the scope of your template definitions using the attributes defined for each template and VeRSL commands such as force, no, and limit.

In the next code segment, we call the INITIAL_WITHOUT_ASSIGNMENTS template or model that we just defined and use a **limit** command to make sure that variables in our Verilog code are not initialized with the initial constructs we specified in our template definition.

Using Variables to Ensure One Module per File

In this next example, we'll write a rule to constrain our Verilog code so there is only one module per file and the name of the module is the same as the file name. This is a common naming convention rule that uses the reserved <**module**> variable:

```
template MODULE_FILE_NAME is module_declaration
  limit file_name to "<module>"
end

Example_12:
limit module_declaration to MODULE_FILE_NAME
message "The module name should be the same as the file name"
severity ERROR
```

In this example, we first define the string MODULE_FILE_NAME to be a template of type **module_declaration**, which is a primary template that does not require a context. But we want to narrow the focus of our template or model to just file names, so we use a **limit** command and the **file_name** attribute of the **module_declaration** template to focus our template just on file names for modules, using the **<module>** variable.

In the next code segment, we call our MODULE_FILE_NAME template and use the **limit** command to apply this model of Verilog code to all **module_declarations** in our design.

Limiting Shifts to Constant Values

In this example, we'll write a rule that makes sure all shifts are done by constant (non-variable) values. It is generally accepted as good coding practice to keep variables out of shifting operations. One of the unique features of the **limit** command is that it lets you define conditional logic tests. In fact, the **limit** command is the only VeRSL command that uses conditional logic.

To write this rule (all shifts are by a constant value), we first need to create three templates, each with a **set** command, that do the following:

- Define a static variable
- Limit our code to binary operations
- Detect the shift operators

We can then test for the conditions defined in the templates, as shown in the following example:

```
template EXP is conditional_expression
  set evaluation_time to locally_static_evaluation
end

template EXP2 is conditional_expression
  set evaluation_time to globally_static_evaluation
end

template LIMIT_SHIFT_RHS is binary_operation
  limit right_expression to EXP,EXP2
end

template DETECT_SHIFT_OPERATION is binary_operation
  limit operator_symbol to "<<", ">>"
end

limit binary_operation to DETECT_SHIFT_OPERATION severity NOTE
  if DETECT_SHIFT_OPERATION then
    Example_13:
    limit binary_operation to LIMIT_SHIFT_RHS
    message "Do not shift by a non-constant value"
    severity ERROR
end if
```

In each template definition, we assign strings of our choosing to templates, as shown in earlier examples, and use the **set** command and template attributes to focus our templates or models only on specific aspects of the Verilog code. Notice how the EXP and EXP2 templates that we defined first are called by the third template, LIMIT_SHIFT_RHS. In the LIMIT_SHIFT_RHS template definition, we use a **limit**

command and the **right_expression** attribute of the **binary_operations** primary template to limit right expressions in our Verilog code to the templates or models we set up in EXP and EXP2.

In the next code segment, we call our DETECT_SHIFT_OPERATION template and use a **limit** command to restrict **binary_operations** to the **operator_symbols** that we defined in our template (<< and >>**)**. If Leda finds operator symbols in binary operations that do not match this template or model, it issues a NOTE message, as we specified.

In the next code segment, we use our DETECT_SHIFT_OPERATION template with some if ... then ... logic. If Leda finds Verilog code that matches the definition we supplied in this template, it enforces a rule using a **limit** command to make sure **binary_operations** in such code segments match the LIMIT_SHIFT_RHS template we specified.

Restricting Asynchronous Resets in Always Blocks

For our next exercise, let's build on what we've already learned and write a rule that allows only one asynchronous reset in each always block.

Hint

To start writing this rule, first create two templates that do the following: (1) look for sequential logic only, and (2) set the maximum number of asynchronous resets to a value of 1.

Recalling our template syntax, here are the first lines of each of our templates in rough form:

```
template (sequential logic only) is A
template (asynchronous reset value is 1) is A
```

where:

A = VeRSL primary or secondary template

Now, let's name our templates, remembering the uppercase convention. Let's also fill in the template structure as follows:

```
template ALWAYS_SEQUENTIAL is A
VeRSL command
end

template ONE_ASYNC_RESET_IN_ALWAYS is A
VeRSL command
end
```

At this point, you probably need another hint to fill in more information. To finish the code, we'll need conditional **limit** commands. The structure of the code should look like this:

```
template ALWAYS_SEQUENTIAL is A
VeRSL command
end

template ONE_ASYNC_RESET_IN_ALWAYS is A
VeRSL command
end

limit A to (one of our templates) severity NOTE
if (one of our templates) then

Example_BetterRuleParadigm:
limit A to (the other template)
message "Only one asynchronous reset is allowed in an always block"
severity ERROR
```

The difficulty now is determining **A**. Looking back at the rule, note the phrase "in an always block." This Verilog construct may be familiar to you as the VeRSL always_construct from Example 4. In fact, **A** for both templates and the **limit** commands is **always_construct**.

The VeRSL commands within the templates are now all that are left to complete the code. For the first template, since we only want sequential logic, we use a **no** command with the **combinatorial** local attribute (from the **always_construct** primary template) as follows:

```
template ALWAYS_SEQUENTIAL is always_construct
  no combinatorial
end
```

For the second template, since we want to limit the number of asynchronous resets to 1, we use the **max** command as follows:

```
template ONE_ASYNC_RESET_IN_ALWAYS is always_construct
  max asynchronous_reset is 1
end
```

The completed code looks like the following example:

```
template ALWAYS_SEQUENTIAL is always_construct
  no combinatorial
end

template ONE_ASYNC_RESET_IN_ALWAYS is always_construct
  max asynchronous_reset is 1
end
```

```
limit always_construct to ALWAYS_SEQUENTIAL severity NOTE
if ALWAYS_SEQUENTIAL then

Example_BetterRuleParadigm:
limit always_construct to ONE_ASYNC_RESET_IN_ALWAYS
message "Only one async. reset is allowed in an always block"
severity ERROR

end if
```

Setting the Clock Edge

Suppose we want to make sure that all clock edges in our design are set to rising. We can do this with a **set** command, which we'll use to set an attribute to a value, as shown in the following code:

```
Example_14:
set edge in clock to rising
message "Use rising edge clock"
severity ERROR
```

In this example, we **set** the **edge** attribute of the **clock** primary template to **rising**. In fact, the **edge** attribute of the **clock** primary template can only be used with **set** commands.

Verilog Test Code

The following code shows how to use this rule to constrain Verilog. As an additional exercise, create both a rule and Verilog code using these examples and run the Checker.

```
module example_14 (clk1, clk2, a , b);

input clk1, clk2, a;
output b;
reg b;

always @(posedge clk1)
begin
  b <= a;
 end

always @(negedge clk2)  <<-------------- Leda pointa here
 begin
  b <= a;
 end
endmodule
```

As a further exercise, review the *VeRSL Reference Guide* and see if you can find the "rising" keyword.

Hint

Enumerated types are primitives in VeRSL. In VeRSL, rising is one of the values listed for edge_type.

Ensuring One Clock Input in Sequential Processes

Suppose we want to create a rule to make sure that sequential always blocks have only one clock input. The following code shows how we can use a **max** command to write this rule:

```
Example_15:
max clock in always_construct is 1
message "Sequential always block must have one clock signal exactly"
severity ERROR
```

In this example, we use a **max** command along with the **clock** attribute of the **always_construct** primary template to make sure there is not more than one clock per always block in our Verilog code.

Specifying Max Characters for Input Port Names

Now let's write a rule requiring that input port names have a maximum of 15 characters. We can break this rule down into a couple of basic elements:

- Maximum character count is 15
- Scope is limited to input ports

We'll need two templates to model our code for these two needs. With the first template, we'll use a **max** command to set the maximum value on character count. With the second template, we'll **limit** the scope to input ports. Then we'll call the templates from our rule using another **limit** command, as shown in the following example:

```
template SIG_PORT_CHAR_COUNT is identifier
  max character_count is 15
end

template INPUT_CHAR_COUNT is input_declaration
  limit identifier to SIG_PORT_CHAR_COUNT
end
```

```
Example_16:
limit input_declaration to INPUT_CHAR_COUNT
message "An input port name can only have a max of 15 characters"
severity ERROR
```

As a further exercise with **max** commands, write a rule to make sure input port names have a minimum of five characters with this additional constraint: the port names must start with “PR_”.

Hint

See the earlier examples that use **limit** commands.

Advanced Rule Creation

The examples in this last section are included to demonstrate the power and flexibility of VeRSL. While you are not expected to understand all of the concepts presented here, you can benefit by studying the examples, paying particular attention to the use of templates and the overall structure of the code.

Because the following examples are long, there are running comments (*******) that explain the code segments.

Regular Expressions, Template-to-Template Calling, Rule Label Duplication

Three basic concepts are presented in this example:

- Constraining names using UNIX regular expressions
- Calling a template from another template
- Duplicating rule labels

We can use duplicate rule labels when we want to classify errors created by rule violations under broad categories:

```
***Using templates to define regular expressions***

template DW_ID is identifier
  limit limit_id to "^DW_"
end

template PRODUCT_ID is identifier
  limit limit_id to "^.._..._",
                    "^.._...._",
                    "^.._....._"
end

***Calling a template from a template***

template MODULE_ID is identifier
```

```
  limit limit_id to "<PRODUCT_ID>.$",
                    "<PRODUCT_ID>..$",
                    "<PRODUCT_ID>...$",
                    "<PRODUCT_ID>....$",
                    "<PRODUCT_ID>.....$",
                    "<PRODUCT_ID>......$",
                    "<PRODUCT_ID>.......$",
                    "<PRODUCT_ID>........$"

end

template DW_MODULE is module_declaration
  limit identifier to DW_ID
end

template MIDDLE_MODULE is module_declaration
  limit identifier to PRODUCT_ID
end

template END_MODULE is module_declaration
  limit identifier to MODULE_ID
end
```

*****Rule label duplication*****

```
MF_1C_R:
  limit module_declaration to  DW_MODULE
     message "Use 'DW_' at the beginning of module name"
     severity ERROR
MF_1C_R:
  limit module_declaration to  MIDDLE_MODULE
     message "Use 3 to 5 characters for the PRODUCT name of the form
              'DW_<PRODUCT_NAME>_"
     severity ERROR
MF_1C_R:
  limit module_declaration to  END_MODULE
      message "The module name has to have up to 8 characters of the
               form 'DW_<PRODUCT_NAME>_<MODULE>"
     severity ERROR
```

Using Multiple Templates in Commands

This example shows how to use multiple templates in commands. The **limit** command constrains the code to the sequential and combinatorial templates. Then a conditional statement enforces the rules. Note the rule label duplication, as in the previous example:

```
template ALWAYS_SEQUENTIAL is always_construct
  no combinatorial
end

template ALWAYS_COMBINATORIAL is always_construct
  force combinatorial
end

***Two templates in limit command***

limit always_construct to ALWAYS_SEQUENTIAL, ALWAYS_COMBINATORIAL
severity NOTE
if ALWAYS_SEQUENTIAL then
CS_5P_R:
  no blocking_assignment
  message "No blocking assignment are allowed in a seqential block"
  severity ERROR
end if
if ALWAYS_COMBINATORIAL then
CS_5P_R:
  no non_blocking_assignment
  message "No non-blocking assignment are allowed in a combinatorial
           block"
  severity ERROR
end if
```

No Variables in Loops

Suppose we want to constrain our Verilog code so that variables do not show up in loops (for statements). To solve this problem, we once again need to use multiple templates, as shown in the following example:

```
***Define the possible variables (local, global, parameter, literal)***

  template LOC_STAT_BINOP is binary_operation
    set evaluation_time to locally_static_evaluation
  end

  template GLOB_STAT_BINOP is binary_operation
    set evaluation_time to globally_static_evaluation
  end

  template PARAM_SIMPLE_NAME is name
    limit object_definition to parameter_declaration
  end

  template LITERAL_VALUE is literal
  end

***Right side of expression must be one of previous templates***

  template BINOP_WITH_STATIC_RIGHT_EXP is binary_operation
    limit right_expression to LOC_STAT_BINOP, GLOB_STAT_BINOP,
                              LITERAL_VALUE, PARAM_SIMPLE_NAME
  end

***Write the rule (note multiple templates in LIMIT command)***

  SYN9_26:
  limit expression in for_statement to LOC_STAT_BINOP,
                                       GLOB_STAT_BINOP,
                                       LITERAL_VALUE,
                                       PARAM_SIMPLE_NAME,
                                       BINOP_WITH_STATIC_RIGHT_EXP
    message "Expression bound in for loop statements should be
             statically computable"
    severity ERROR
```

Constraining Technology-independent Registers

In this final example, let's create a rule that constrains our Verilog code for technology-independent registers. The Verilog code we are modeling is shown in the first comments. To write this rule, we first create all the templates and then use them in the **limit** command section to constrain the Verilog code:

```
ruleset RULESET_6 is
-------------------------------------------------
-- Rule G_551_1 : Use the following templates to infer
technology-independent registers
--
--   always @(posedge clk [or posedge reset])
--     begin : LABEL
--       if (reset == 1'b1)
--         begin
--           ...
--         end
--       else
--         begin
--           ...
--         end
--     end

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

  template EVENT_TIMING_CONTROL is procedural_timing_control_statement
    force event_control
  end

  template SEQUENTIAL_FF_ALWAYS is always_construct
    force flipflop
  end

  template ALWAYS_WITH_ONE_TIMING_CONTROL is always_construct
    max procedural_timing_control_statement is 1
  end

  template ALWAYS_WITH_CLOCK_AND_ASYNCH_RESET is always_construct
    force flipflop
    force clock
    force asynchronous_reset
  end

  template ALWAYS_WITH_CLOCK_AND_SYNCH_RESET is always_construct
    force flipflop
    force clock
```

```
    force synchronous_reset
  end

  template POSEDGED_ASYNC_RESET is asynchronous_reset
    set edge to rising
  end

  template NEGEDGED_ASYNC_RESET is asynchronous_reset
    set edge to falling
  end

  template BINARY_0 is literal
    set base to 2
    limit value to "^0$"
  end

  template BINARY_1 is literal
    set base to 2
    limit value to "^1$"
  end

  template BINARY_LOW_RESET_CONDITION is binary_operation
    limit operator_symbol to "=="
    limit right_expression to BINARY_0
  end

  template BINARY_HIGH_RESET_CONDITION is binary_operation
    limit operator_symbol to "=="
    limit right_expression to BINARY_1
  end

-----------------------------------------------------------------
-- Command Section
-----------------------------------------------------------------

G_551_1:
limit statement in always_construct to EVENT_TIMING_CONTROL
message "The always keyword must be followed by an event list @(...)"
severity WARNING

limit always_construct to SEQUENTIAL_FF_ALWAYS severity NOTE
if SEQUENTIAL_FF_ALWAYS then

G_551_1:
max clock in always_construct is 1
message "There should be one clock signal exactly in the sensitivity
list of a sequential block"
severity WARNING
```

```
limit always_construct to ALWAYS_WITH_ONE_TIMING_CONTROL
severity NOTE

if ALWAYS_WITH_ONE_TIMING_CONTROL then

G_551_1:
limit expression in event_control to posedge_event, negedge_event
message "Level sensitive events are not allowed in a sequential always
block"
severity WARNING
end if

-- THIS SECTION FOR SEQUENTIAL BLOCKS WITH SYNCH/ASYNCHRONOUS RESET

limit always_construct to
ALWAYS_WITH_CLOCK_AND_ASYNCH_RESET,
ALWAYS_WITH_CLOCK_AND_SYNCH_RESET
severity NOTE

if ALWAYS_WITH_CLOCK_AND_ASYNCH_RESET then

G_551_1:
max asynchronous_reset in always_construct is 1
message "There should be exactly one asynchronous signal in the
sensitivity list of a sequential block"
severity WARNING

G_551_1:
limit asynchronous_reset to POSEDGED_ASYNC_RESET, NEGEDGED_ASYNC_RESET
message "An asynchronous set/reset signal should be preceded by the
keyword 'posedge' or 'negedge' in the sensitivity list"
severity WARNING

limit asynchronous_reset to POSEDGED_ASYNC_RESET, NEGEDGED_ASYNC_RESET
severity NOTE
if POSEDGED_ASYNC_RESET then
G_551_1:
limit expression in asynchronous_reset to BINARY_HIGH_RESET_CONDITION
message "Use 'if(<asynch_reset> == 'b1)' for rising edge asynchronous
reset"
severity WARNING
end if

if NEGEDGED_ASYNC_RESET then
G_551_1:
limit expression in asynchronous_reset to BINARY_LOW_RESET_CONDITION
message "Use 'if(<asynch_reset> == 'b0)' for falling edge asynchronous
reset"
severity WARNING
```

```
   end if
end if

if ALWAYS_WITH_CLOCK_AND_SYNCH_RESET then
G_551_1:
max synchronous_reset in always_construct is 1
message "There should be exactly one synchronous reset signal in a
synchronous block"
severity WARNING

G_551_1:
limit expression in synchronous_reset to BINARY_HIGH_RESET_CONDITION,
BINARY_LOW_RESET_CONDITION
message "Use  'if(<synch_reset> == 'b0)' or 'if(<synch_reset> == 'b1)'
for synchronous reset expressions"
severity WARNING
   end if
end if

end ruleset
```

This concludes our tutorial on learning how to use VeRSL to write custom rules for constraining Verilog designs. For complete reference information on VeRSL, see the *VeRSL Reference Guide*.

Index

H

I

K

L

M

N

O

P

R

S

T

V

W