vhdl-style-guide Documentation

Release 1.10.0

Jeremiah C Leary

Jun 04, 2020

Contents:

CHAPTER 1

Overview

VHDL Style Guide (VSG) provides coding style guide enforcement for VHDL code.

1.1 Why VSG?

VSG was created after participating in a code review in which a real issue was masked by a coding style issue. A finding was created for the style issue, while the real issue was missed. When the code was re-reviewed, the real issue was discovered. The coding style issue seemed to blind me to the real issue.

Depending on your process, style issues can take a lot of time to resolve.

1. Create finding/ticket/issue
2. Disposition finding/ticket/issue
3. Fix the problem
4. Verify the problem was fixed

Spending less time on style issues leaves more time to analyze code structure. Eliminating style issues reduces the amount of time performing code reviews. This results in a higher quality code base.

1.2 Key Benefits

- Explicitly define VHDL coding standards
- Make coding standards visible to everyone
- Improve code reviews
- Quickly bring code up to current standards

VSG allows the style of the code to be defined and enforced over portions or the entire code base.

1.3 Key Features

- Command line tool
  - Integrates into continuous integration flow tools
- Reports and fixes issues found
  - Horizontal whitespace
  - Vertical whitespace
  - Upper and lower case
  - Keyword alignments
  - etc. . .
- Fully configurable rules via JSON/YAML configuration file
  - Disable rules
  - Alter behavior of existing rules
  - Change phase of execution
- Localize rule sets
  - Create your own rules using python
  - Use existing rules as a template
  - Fully integrates into base rule set

CHAPTER 2

Gallery

The examples shown below illustrate the formatting enforced by VSG. They show a subset of the rules:

- capitalization
- indentation
- column alignments
  - comments
  - :'s
  - assignment operators (<= and =>)
- vertical spacing

2.1 Entities

```
entity GRP_DEBOUNCER is
  generic (
    N       : positive := 8;                     -- input bus width
    CNT_VAL : positive := 10000                  -- clock counts for debounce period
  );
  port (
    CLK_I   : in    std_logic := 'X';            -- system clock
    DATA_I  : in    std_logic_vector(1 downto 0)    -- noisy input data
    DATA_O  : out   std_logic_vector(1 downto 0); -- registered stable output data
    STRB_O  : out   std_logic                    -- strobe for new data available
  );
end entity GRP_DEBOUNCER;
```

2.2 Architectures

```
architecture BEHAVIORAL of PIC is

  type state_type is (
    reset_s, get_commands, jump_int_method, start_polling,
    ack_txinfo_rxd, start_priority_check, tx_int_info_priority
  );

  signal next_s               : state_type :=reset_s;
  signal int_type             : unsigned(1 downto 0):="01";
  signal int_index, count_cmd : integer := 0;

  type prior_table is array (0 to 7) of unsigned(2 downto 0);

  signal pt                   : prior_table := (others => (others => '0'));
  signal int_pt               : unsigned(2 downto 0):="000";
  signal flag,       flag1    : std_logic := '0';

begin

end architecture BEHAVIORAL;
```

2.3 Component Declarations

```
component CPU is
  port (
    CLK_I         : in    std_logic;
    SWITCH        : in    std_logic_vector(9 downto 0);

    SER_IN        : in    std_logic;
    SER_OUT       : out   std_logic;

    TEMP_SPO      : in    std_logic;
    TEMP_SPI      : out   std_logic;
    TEMP_CE       : out   std_logic;
    TEMP_SCLK     : out   std_logic;

    SEG1          : out   std_logic_vector(7 downto 0);
    SEG2          : out   std_logic_vector( 7 downto 0);
    LED           : out   std_logic_vector( 7 downto 0);

    XM_ADR        : out   std_logic_vector(15 downto 0);
    XM_RDAT       : in    std_logic_vector( 7 downto 0);
    XM_WDAT       : out   std_logic_vector( 7 downto 0);
    XM_WE         : out   std_logic;
    XM_CE         : out   std_logic
  );
end component;
```

2.4 Component Instantiations

```
INTERLEAVER_I0 : INTERLEAVER
  generic map (
    DELAY       => TREL1_LEN + TREL2_LEN + 2 + delay,
    WAY         => 0
  )
  port map (
    CLK         => clk,
    RST         => rst,
    D           => tmp0,
    Q           => tmp1
  );
```

2.5 Concurrent Assignments

```
nCounter       <= x"FFFFFF" when Counter=x"FFFFFF" and Button='1' else
                  x"000000" when Counter=x"000000" and Button='0' else
                  Counter + 1 when Button='1' else
                  Counter - 1;
nextHistory    <= '0' when Counter=x"000000" else
                  '1';
nButtonHistory <= nextHistory & ButtonHistory(1);
Dout           <= '1' when ButtonHistory="01" else
                  '0';
```

CHAPTER 3

Installation

There are two methods to install VSG.

3.1 PIP

The most recent released version is hosted on PyPI. It can be installed using **pip**.

```
pip install vsg
```

This is the preferred method for installing VSG.

3.2 Git Hub

The latest development version can be cloned from the git hub repo.

```
git clone https://github.com/jeremiah-c-leary/vhdl-style-guide.git
```

Then installed using the setup.py file.

```
python setup.py install
```

CHAPTER 4

Usage

VSG is a both a command line tool and a python package. The command line tool can be invoked with:

```
$ vsg
usage: VHDL Style Guide (VSG) [-h] [-f FILENAME [FILENAME ...]]
                              [-lr LOCAL_RULES]
                              [-c CONFIGURATION [CONFIGURATION ...]] [--fix]
                              [-fp FIX_PHASE] [-j JUNIT] [-of {vsg,syntastic}]
                              [-b] [-oc OUTPUT_CONFIGURATION] [-v]

Analyzes VHDL files for style guide violations. Reference documentation is
located at: http://vhdl-style-guide.readthedocs.io/en/latest/index.html

optional arguments:
  -h, --help            show this help message and exit
  -f FILENAME [FILENAME ...], --filename FILENAME [FILENAME ...]
                        File to analyze
  -lr LOCAL_RULES, --local_rules LOCAL_RULES
                        Path to local rules
  -c CONFIGURATION [CONFIGURATION ...], --configuration CONFIGURATION [CONFIGURATION .
↪..]
                        JSON or YAML configuration file(s)
  --fix                 Fix issues found
  -fp FIX_PHASE, --fix_phase FIX_PHASE
                        Fix issues up to and including this phase
  -j JUNIT, --junit JUNIT
                        Extract Junit file
  -of {vsg,syntastic}, --output_format {vsg,syntastic}
                        Sets the output format.
  -b, --backup          Creates copy of input file for comparison with fixed
                        version.
  -oc OUTPUT_CONFIGURATION, --output_configuration OUTPUT_CONFIGURATION
                        Output configuration file name
  -v, --version         Displays version information
```

Command Line Options

Option	Description
-f FILENAME	The VHDL file to be analyzed or fixed. Multiple files can be passed through this option.
–local_rules LOCAL_RULES	Additional rules not in the base set.
–configuration CONFIGURATION	JSON or YAML file(s) which alters the behavior of VSG. Configuration can also include a list files to analyze. Any combination of JSON and YAML files can be passed. Each will be processed in order from left to right.
–fix	Update issues found. Replaces current file with updated one.
–fix_phase	Applies for all phases up to and including this phase. Analysis will then be performed on all phases.
–junit	Filename of JUnit XML file to generate.
–output_format	**Configures the sdout output format.** vsg – standard VSG output syntastic – format compatible with the syntastic VIM module
–backup	Creates a copy of the input file before applying any fixes. This can be used to compare the fixed file against the original.
–output_configuration	Writes a JSON configuration file of the current run. It includes a file_list, local_rules (if used), and how every rule was configured. This configuration can be fed back into VSG.
–version	Displays the version of VSG.

Here is an example output running against a test file:

```
$ vsg -f PIC.vhd
File:  PIC.vhd
==============
Phase 1... Reporting
Phase 2... Reporting
Phase 3... Reporting
Phase 4... Reporting
Phase 5... Reporting
  comment_002               |         51 | Ensure proper alignment of comment with␣
↪previous line.
  comment_002               |         52 | Ensure proper alignment of comment with␣
↪previous line.
  comment_002               |         54 | Ensure proper alignment of comment with␣
↪previous line.
  comment_002               |         55 | Ensure proper alignment of comment with␣
↪previous line.
  comment_003               |     76-256 | Inconsistent alignment of comments within␣
↪process.
  sequential_005            |      87-93 | Inconsistent alignment of "<=" in group of␣
↪lines.
  sequential_005            |    102-103 | Inconsistent alignment of "<=" in group of␣
↪lines.
  sequential_005            |    105-108 | Inconsistent alignment of "<=" in group of␣
↪lines.
  sequential_005            |    110-113 | Inconsistent alignment of "<=" in group of␣
↪lines.
```

(continues on next page)

(continued from previous page)

```
  sequential_005            |    115-118 | Inconsistent alignment of "<=" in group of␣
↪lines.
  sequential_005            |    120-124 | Inconsistent alignment of "<=" in group of␣
↪lines.
  sequential_005            |    129-133 | Inconsistent alignment of "<=" in group of␣
↪lines.
  sequential_005            |    160-161 | Inconsistent alignment of "<=" in group of␣
↪lines.
  sequential_005            |    173-174 | Inconsistent alignment of "<=" in group of␣
↪lines.
  comment_002               |        183 | Ensure proper alignment of comment with␣
↪previous line.
  sequential_005            |    225-226 | Inconsistent alignment of "<=" in group of␣
↪lines.
  sequential_005            |    238-239 | Inconsistent alignment of "<=" in group of␣
↪lines.
Phase 6... Not executed
Phase 7... Not executed
==============
Total Rules Checked: 204
Total Failures:      523
```

VSG will report the rule which is violated and the line number or group of lines where the violation occured. It also gives a suggestion on how to fix the violation. The rules VSG uses are grouped together into *Phases*. These phases follow the order in which the user would take to address the violations. Each rule is detailed in the *Rules* section. The violation and the appropriate fix for each rule is shown.

The violations can be fixed manually, or use the **–fix** option to have VSG update the file.

```
$ vsg -f PIC.vhd --fix
File:  PIC.fixed.vhd
====================
Phase 1... Reporting
Phase 2... Reporting
Phase 3... Reporting
Phase 4... Reporting
Phase 5... Reporting
Phase 6... Reporting
Phase 7... Reporting
====================
Total Rules Checked: 290
Total Failures:      0
```

If rule violations can not be fixed, they will be reported after fixing everything else:

```
$ vsg -f PIC.vhd --fix
File:  PIC.vhd
==============
Phase 1... Reporting
  signal_007                |         66 | Remove default assignment.
  signal_007                |         67 | Remove default assignment.
  signal_007                |         68 | Remove default assignment.
  signal_007                |         72 | Remove default assignment.
  signal_007                |         73 | Remove default assignment.
  signal_007                |         74 | Remove default assignment.
  process_016               |         78 | Add a label for the process.
  process_018               |        259 | Add a label for the "end process".
```

(continues on next page)

(continued from previous page)

```
Phase 2... Not executed
Phase 3... Not executed
Phase 4... Not executed
Phase 5... Not executed
Phase 6... Not executed
Phase 7... Not executed
==============
Total Rules Checked: 48
Total Failures:      8
```

4.1 Multiple configuration example

More than one configuration can be passed using the **–configuration** option. This can be useful in two situations:

1) Block level configurations
2) Multilevel rule configurations

The priority of the configurations is from right to left. The last configuration has the highest priority. This is true for all configuration parameters except **file_list**.

4.1.1 Block level configurations

Many code bases are large enough to be broken into multiple sub blocks. A single configuration can be created and maintained for each subblock. This allows each subblock to be analyzed independently.

When the entire code base needs be analyzed, all the subblock configurations can be passed to VSG. This reduces the amount of external scripting required.

config_1.json

```
{
    "file_list":[
      "fifo.vhd",
      "source/spi.vhd",
      "$PATH_TO_FILE/spi_master.vhd",
      "$OTHER_PATH/src/*.vhd"
    ]
}
```

config_2.json

```
{
    "file_list":[
      "dual_port_fifo.vhd",
      "flash_interface.vhd",
      "$PATH_TO_FILE/ddr.vhd"
    ]
}
```

Both configuration files can be processed by vsg with the following command:

```
$ vsg --configuration config_1.json config_2.json
```

4.1.2 Multilevel rule configurations

Some code bases may require rule adjustments that apply to all the files along with rule adjustments against individual files. Use multiple configurations to accomplish this. One configuration can handle code base wide adjustments. A second configuration can target individual files. VSG will combine any number of configurations to provide a unique set of rules for any file.

config_1.json

```
{
    "rule":{
        "entity_004":{
            "disable":true
        },
        "entity_005":{
            "disable":true
        },
        "global":{
            "indentSize":2
        }
    }
}
```

config_2.json

```
{
    "rule":{
        "entity_004":{
            "disable":false,
            "indentSize":4
        }
    }
}
```

Both configuration files can be processed by VSG with the following command:

```
$ vsg --configuration config_1.json config_2.json -f fifo.vhd
```

VSG will combine the two configurations into this equivalent configuration...

```
{
    "rule":{
        "entity_004":{
            "disable":false,
            "indentSize":4
        },
        "entity_005":{
            "disable":true
        },
        "global":{
            "indentSize":2
        }
    }
}
```

...and run on the file **fifo.vhd**.

CHAPTER 5

Configuring

VSG can use a configuration file to alter it's behavior or include a list of files to analyze. This is accomplished by passing JSON and/or YAML file(s) through the **–configuration** command line argument. This is the basic form of a configuration file in JSON:

```
{
    "file_list":[
      "fifo.vhd",
      "source/spi.vhd",
      "$PATH_TO_FILE/spi_master.vhd",
      "$OTHER_PATH/src/*.vhd"
    ],
    "local_rules":"$DIRECTORY_PATH",
    "rule":{
        "global":{
            "attributeName":"AttributeValue"
        },
        "ruleId_ruleNumber":{
            "attributeName":"AttributeValue"
        }
    }
}
```

This is the basic form of a configuration file in YAML:

```
---
file_list:[
  - fifo.vhd
  - source/spi.vhd
  - $PATH_TO_FILE/spi_master.vhd
  - $OTHER_PATH/src/*.vhd
local_rules: $DIRECTORY_PATH
rule:
  global:
    attributeName: AttributeValue
```

(continues on next page)

(continued from previous page)

```
  ruleId_ruleNumber:
      attributeName: AttributeValue
...
```

It is not required to have **file_list**, **local_rules**, and **rule** defined in the configuration file. Any combination can be defined. The order does not matter either.

Note: All examples of configurations in this documentation use JSON. However, YAML can be used instead.

5.1 file_list

The file_list is a list of files that will be analyzed. Environment variables will expanded. File globbing is also supported. The Environment variables will be expanded before globbing occurs. This option can be useful when running VSG over multiple files.

5.2 local_rules

Local rules can be defined on the command line or in a configuration file. If they are defined in both locations, the configuration will take precedence.

5.3 rule

Any attribute of any rule can be configured. Using **global** will set the attribute for every rule. Each rule is addressable by using it's unique **ruleId** and **ruleNumber** combination. For example, whitespace_006 or port_010.

Note: If **global** and unique attributes are set at the same time, the unique attribute will take precedence.

Here are a list of attributes that can be altered for each rule:

Attribute	Values	Description
indentSize	Integer	Sets the number of spaces for each indent level.
phase	Integer	Sets the phase the rule will run in.
disable	Boolean	If set to True, the rule will not run.
fixable	Boolean	If set to False, the violation will not be fixed

Note: Some rules have additional attributes. These will be noted in the rule description.

5.3.1 Example: Disabling a rule

Below is an example of a JSON file which disables the rule **entity_004**

```
{
    "rule":{
        "entity_004":{
            "disable":true
        }
    }
}
```

Use the configuration with the **–configuration** command line argument:

```
$ vsg -f RAM.vhd --configuration entity_004_disable.json
```

5.3.2 Example: Setting the indent increment size for a single rule

The indent increment size is the number of spaces an indent level takes. It can be configured on an per rule basis...

```
{
    "rule":{
        "entity_004":{
            "indentSize":4
        }
    }
}
```

5.3.3 Example: Setting the indent increment size for all rules

Configure the indent size for all rules by setting the **global** attribute.

```
{
    "rule":{
        "global":{
            "indentSize":4
        }
    }
}
```

5.4 Configuring Uppercase and Lowercase Rules

There are several rules that enforce either uppercase or lowercase. They are noted in the documentation for each rule what the default is. The example violation and fixes are in reference to the default setting.

The default value for each of these case rules can be overridden using a configuration.

5.4.1 Overriding Default Case Enforcement

The default setting can be changed using a configuration. For example the rule constant_002 defaults to lowercase. We can use the following configuration to change the case to upper:

```
---

rule :
  constant_002 :
     case : 'upper'
```

Conversley, rule entity_008 defaults to uppercase. We can use the following configuration to change the case to lower:

```
---

rule :
  entity_008 :
     case : 'lower'
```

5.4.2 Changing Multiple Case Rules

If there are a lot of case rules you want to change, you can use the global option to reduce the size of the configuration. For example, if we want to uppercase everything except the entity name, we could write the following configuration:

```
---

rule :
  global :
    case : 'upper'
  entity_008 :
    case : 'lower'
```

5.5 Configuring Prefix and Suffix Rules

There are several rules that enforce specific prefixes/suffixes in different name identifiers. It is noted, in the documentation, what are the default prefixes/suffixes for each such rule.

All prefix/suffix rules are disabled by default. The default prefixes/suffixes for each of these rules can be overridden using a configuration.

5.5.1 Overriding Default Prefixes/Suffixes Enforcement

The default setting can be changed using a configuration. For example, the rule port_025 defaults to following suffixes: [‘_I’, ‘_O’, ‘_IO’]. We can use the following configuration to change allowed suffixes:

```
---

 rule :
     port_025:
          # Each prefix/suffix rule needs to be enabled explicitly.
          disable: false
          suffixes: ['_i', '_o']
```

The rule variable_012 defaults to following prefix: [‘v_’]. We can use the following configuration to change allowed prefix:

```
---

 rule :
     variable_012:
         # Each prefix/suffix rule needs to be enabled explicitly.
         disable: false
         prefixes: ['var_']
```

5.6 Configuring Number of Signals in Signal Declaration

VHDL allows of any number of signals to be declared within a single signal declaration. While this may be allowed, in practice there are limits impossed by the designers. Limiting the number of signals declared improves the readability of VHDL code.

The default number of signals allowed, 2, can be set by configuring rule **signal_015**.

5.6.1 Overriding Number of Signals

The default setting can be changed using a configuration. We can use the following configuration to change the number of signals allowed to 1.

```
---

rule :
  signal_015 :
     consecutive : 1
```

5.7 Configuring the Maximum Line Length

Limiting the line length of the VHDL code can improve readability. Code that exceeds the editor window is more difficult to read.

The default line length is 120, and can be set by configuring rule **length_001**.

5.7.1 Overriding Line Length

The default setting can be changed using a configuration. We can use the following configuration to change the line length to 180.

```
---

rule :
  length_001 :
     length : 180
```

CHAPTER 6

Code Tags

VSG supports inline tags embedded into code to enable or disable rules. This can be useful in fine tuning rule exceptions within a file. The code tags are embedded in comments similar to pragmas, and must be on it's own line.

6.1 Full rule exclusion

Entire portions of a file can be ignored using the **vsg_off** and **vsg_on** tags.

```
-- vsg_off
process (write, read, full) is
begin
  a <= write;
  b <= read;
end process;
-- vsg_on
```

The **vsg_off** tag disables all rule checking. The **vsg_on** tag enables all rule checking, except those disabled by a configuration.

6.2 Individual Rule Exclusions

Individual rules can be disabled by adding the rule identifier to the **vsg_off** and **vsg_on** tags. Multiple identifiers can be added.

```
-- vsg_off process_016 process_018
process (write, read, full) is
begin
  a <= write;
  b <= read;
end process;
-- vsg_on
```

The bare **vsg_on** enables all rules not disabled by a configuration.

Each rule can be independently enabled or disabled:

```
-- vsg_off process_016 process_018
process (write, read, full) is
begin
  a <= write;
  b <= read;
end process;

-- vsg_on process_016
FIFO_PROC : process (write, read, full) is
begin
  a <= write;
  b <= read;
end process;

-- vsg_on process_018
FIFO_PROC : process (write, read, full) is
begin
  a <= write;
  b <= read;
end process FIFO_PROC;
```

In the previous example, the *process_016* and *process_018* are disabled for the first process. *Process_018* is disabled for the second process. No rules are disabled for the third process.

CHAPTER 7

Editor Integration

If your editor can execute programs on the command line, you can run VSG without having to leave your editor. This brings a new level of efficiency to coding in VHDL.

7.1 VIM

Add the following macro into your .vimrc file:

```
map <F9> :setl autoread<CR>:let b:current_file = @%<CR>:w!<CR>:execute '!vsg -f ' .␣
↪b:current_file ' --fix'<CR><CR>:edit<CR>:setl noautoread<CR>
```

This macro bound to the <F9> key performs the following steps:

1. Save the current buffer
2. Execute vsg with the –fix option
3. Reload the buffer

When you are editing a file, you can hit <F9> and VSG will run on the current buffer without leaving VIM.

CHAPTER 8

Localizing

VSG supports customization to your coding style standards by allowing localized rules. These rules are stored in a directory with an __init__.py file and one or more python files. The files should follow the same structure and naming convention as the rules found in the vsg/rules directory.

The localized rules will be used when the **–local_rules** command line argument is given or using the **local_rules** option in a configuration file.

8.1 Example: Create rule to check for entity and architectures in the same file.

Let's suppose in our organization the entity and architecture should be split into separate files. This rule is not in the base rule set, but we can add it through localization. For this example, we will be setting up the localized rules in your home directory.

8.1.1 Prepare local rules directory

Create an empty directory with an empty __init__.py file

```
$ mkdir ~/local_rules
$ touch ~/local_rules/__init__.py
```

8.1.2 Create new rule file

We will create a new rule by extending the base rule class.

Note: The file name and class name must start with **rule_**. Otherwise VSG will not recognize it as a rule.

The rule will be in the **localized** group. Since this is the first rule, we will number it **001**.

```
from vsg import rule


class rule_001(rule.rule):

  def __init__(self):
      rule.rule.__init__(self, 'localized', '001')
```

Referencing the *Phases*, we decide it should be in phase 1: structural.

```
from vsg import rule


class rule_001(rule.rule):

  def __init__(self):
      rule.rule.__init__(self, 'localized', '001')
      self.phase = 1
```

Now we need to add the **analyze** method to perform the check.

```
from vsg import rule


class rule_001(rule.rule):

  def __init__(self):
      rule.rule.__init__(self, 'localized', '001')
      self.phase = 1

  def analyze(self, oFile):
```

The built in variables in the vsg.line class can be used to build rules. There are helper functions in *vsg.utilities*, *vsg.check*, and *vsg.fix* also. In this case, the vsg.vhdlFile class has two attributes (**hasEntity** and **hasArchitecture**) that are exactly what we need. We are ready to write the body of the **analyze** method:

```
from vsg import rule


class rule_001(rule.rule):

  def __init__(self):
      rule.rule.__init__(self, 'localized', '001')
      self.phase = 1

  def analyze(self, oFile):
      if oFile.hasEntity and oFile.hasArchitecture:
          self.add_violation(1)
```

The base rule class has an **add_violation** method which takes a line number as an argument. This method appends the line number to a violation list, which is processed later for reporting and fixing purposes. In this case, any line number will do so we picked 1.

We must decide if we want to give VSG the ability to fix this rule on it's own. If so, then we will need to write the **_fix_violations** method. However, for this violation we want the user to split the file. We will tell VSG the rule is not fixable.

```
from vsg import rule


class rule_001(rule.rule):

  def __init__(self):
      rule.rule.__init__(self, 'localized', '001')
      self.phase = 1
      self.fixable = False  # User must split the file

  def analyze(self, oFile):
      if oFile.hasEntity and oFile.hasArchitecture:
          self.add_violation(1)
```

We also need to provide a solution to the user so they will know how to fix the violation:

```
from vsg import rule


class rule_001(rule.rule):

  def __init__(self):
      rule.rule.__init__(self, 'localized', '001')
      self.phase = 1


      self.fixable = False  # User must split the file
      self.solution = 'Split entity and architecture into seperate files.'

  def analyze(self, oFile):
      if oFile.hasEntity and oFile.hasArchitecture:
          self.add_violation(1)
```

Finally, we need to add a code tag check so the rule can be disabled via comments in the code:

```
from vsg import rule


class rule_001(rule.rule):

  def __init__(self):
      rule.rule.__init__(self, 'localized', '001')
      self.phase = 1
      self.fixable = False  # User must split the file
      self.solution = 'Split entity and architecture into seperate files.'

  def analyze(self, oFile):
      if not self.is_vsg_off(oLine):
          if oFile.hasEntity and oFile.hasArchitecture:
              self.add_violation(1)
```

The rule is complete, so we save it as rule_localized_001.py. Performing an **ls** on our local_rules directory:

```
$ ls ~/local_rules
__init__.py  rule_localized_001.py
```

8.1.3 Use new rule to analyze

When we want to run with localized rules, use the **–local_rules** option.

```
$ vsg -f RAM.vhd --local_rules ~/local_rules
File:  RAM.vhd
==============
Phase 1... Reporting
localized_001            |            1 | Split entity and architecture into seperate␣
↪files.
Phase 2... Not executed
Phase 3... Not executed
Phase 4... Not executed
Phase 5... Not executed
Phase 6... Not executed
Phase 7... Not executed
==============
Total Rules Checked: 50
Total Failures:      1
```

Our new rule will now flag files which have both an entity and an architecture in the same file. That was a fairly simple rule. To write more complex rules, it is important to understand how the rule class works.

8.2 Understanding the Rule class

Every rule uses the base rule class. There are a few methods to the base rule class, but we are interested in only the following:

Method	Description
add_violations	Adds violations to a list.
analyze	Calls _pre_analyze and then _analyze.
_analyze	Code that performs the analysis.
fix	calls analyze and then _fix_violations.
_fix_violations	Code that fixes the violations.
_get_solution	Prints out the solution to stdout.
_pre_analyze	Code that sets up variables for _analyze.

We will look at the rule **constant_014** to illustrate how VSG uses the methods above:

```
class rule_014(rule.rule):
    '''
    Constant rule 014 checks the indent of multiline constants that are not arrays.
    '''

    def __init__(self):
        rule.rule.__init__(self)
        self.name = 'constant'
        self.identifier = '014'
        self.solution = 'Align with := keyword on constant declaration line.'
        self.phase = 5

    def _pre_analyze(self):
        self.alignmentColumn = 0
        self.fKeywordFound = False
```

(continues on next page)

(continued from previous page)

```
    def _analyze(self, oFile, oLine, iLineNumber):
        if not oLine.isConstantArray and oLine.insideConstant:
            if oLine.isConstant and ':=' in oLine.line:
                self.alignmentColumn = oLine.line.index(':=') + len(':= ')
                self.fKeywordFound = True
            elif not oLine.isConstant and self.fKeywordFound:
                sMatch = ' ' * self.alignmentColumn
                if not re.match('^' + sMatch + '\w', oLine.line):
                    self.add_violation(iLineNumber)
                    self.dFix['violations'][iLineNumber] = self.alignmentColumn
            if oLine.isConstantEnd:
                self.fKeywordFound = False

    def _fix_violations(self, oFile):
        for iLineNumber in self.violations:
            sLine = oFile.lines[iLineNumber].line
            sNewLine = ' ' * self.dFix['violations'][iLineNumber] + sLine.strip()
            oFile.lines[iLineNumber].update_line(sNewLine)
```

8.2.1 Creating Class

First we create the rule by inheriting from the base rule class. We also add a comment to describe what the rule is doing.

```
class rule_014(rule.rule):
    '''
    Constant rule 014 checks the indent of multiline constants that are not arrays.
    '''
```

8.2.2 Adding __init__

Then we add the **__init__** method. It calls the init of the base rule class, then we modify attributes for this specific rule:

```
def __init__(self):
    rule.rule.__init__(self)
    self.name = 'constant'
    self.identifier = '014'
    self.solution = 'Align with := keyword on constant declaration line.'
    self.phase = 5
```

For this rule we set it's *name*, *identifier*, *solution*, and *phase*.

8.2.3 Analyzing Considerations

The **analyze** method of the base rule class will first call **_pre_anaylze** before **_analyze**. The **_analyze** method is wrapped in a loop that increments through each line of the file. The **analyze** method also checks if the rule has been turned off for a line, via code tags. If the code tag indicates to ignore the line, then it will be skipped. If you decide to override the **analyze** method, then you should add the code tag check.

8.2.4 Adding _pre_analyze method

In this rule, we use the **_pre_analyze** method to initialize some variables. These variables must be set outside the loop that is present in the **analyze** method.

```
def _pre_analyze(self):
    self.alignmentColumn = 0
    self.fKeywordFound = False
```

8.2.5 Adding _analyze method

The **_analyze** method is called on every line of the VHDL file. Any memory needed between lines must be declared in the **_pre_analyze** method. In the following code, notice *self.alignmentColumn* and *self.fKeywordFound*.

```
def _analyze(self, oFile, oLine, iLineNumber):
    if not oLine.isConstantArray and oLine.insideConstant:
        if oLine.isConstant and ':=' in oLine.line:
            self.alignmentColumn = oLine.line.index(':=') + len(':= ')
            self.fKeywordFound = True
        elif not oLine.isConstant and self.fKeywordFound:
            sMatch = ' ' * self.alignmentColumn
            if not re.match('^' + sMatch + '\w', oLine.line):
                self.add_violation(iLineNumber)
                self.dFix['violations'][iLineNumber] = self.alignmentColumn
        if oLine.isConstantEnd:
            self.fKeywordFound = False
```

This code is searching for the characteristics of a non-array constant.

```
def _analyze(self, oFile, oLine, iLineNumber):
    if not oLine.isConstantArray and oLine.insideConstant:
```

Once the non-array constant is found, it notes the column of the *:=* keyword.

```
if oLine.isConstant and ':=' in oLine.line:
    self.alignmentColumn = oLine.line.index(':=') + len(':= ')
    self.fKeywordFound = True
```

On successive lines of the constant declaration, it checks to see if there are enough spaces from the beginning of the line to match the column number the *:=* is located at.

```
elif not oLine.isConstant and self.fKeywordFound:
```

If there are not enough spaces, then a violation is added. We also store off the required column into a predefined dictionary named *dFix*. This will be used later when the **fix** method is called.

```
sMatch = ' ' * self.alignmentColumn
if not re.match('^' + sMatch + '\w', oLine.line):
    self.add_violation(iLineNumber)
    self.dFix['violations'][iLineNumber] = self.alignmentColumn
```

When we detect the end of the constant declaration, we clear a flag and prepare for the next constant declaration.

```
if oLine.isConstantEnd:
    self.fKeywordFound = False
```

8.2.6 Fixing considerations

The **fix** method will first call the **analyze** method and then the **_fix_violations** method. Unlike the **analyze** method, it does not wrap the **_fix_violations** in a loop. This is due to some fixes needing to execute either top down or bottom up. Rules that add or delete lines need to work from the bottom up. Otherwise, the violations detected by the **analyze** method will have moved.

8.2.7 Adding the _fix_violations method

In this rule, we are going to iterate on all the violations in the *self.violations* attribute.

```
def _fix_violations(self, oFile):
    for iLineNumber in self.violations:
```

We store the current line off to make it easier to read. Then we strip the line of all leading and trailing spaces and prepend the number of spaces required to align with the *:=* keyword.

```
sLine = oFile.lines[iLineNumber].line
sNewLine = ' ' * self.dFix['violations'][iLineNumber] + sLine.strip()
```

Finally, we update the line with our modified line using the **update_line** method.

```
oFile.lines[iLineNumber].update_line(sNewLine)
```

8.3 Rule creation guidelines

Keep these points in mind when creating new rules:

1. Use an existing rule as a starting point
2. Remember that **analyze** calls **_pre_analyze** and then **_analyze**
3. Override **_get_solution** to return complex messages
4. **analyze** method can be overridden if necessary
5. If overriding **analyze**, then include a check for *vsg_off*

CHAPTER 9

Phases

Rules are grouped together and executed in phases. This simplifies rule generation for rules in later phases. If issues are found during a phase, then successive phases will not run. The phases are constructed to model the proper order of fixing issues. Each phase prepares the code for the next phase.

9.1 Phase 1 - Structural

This phase checks the structure of VHDL statements. This ensures the VHDL is structured properly for future phases.

9.2 Phase 2 - Whitespace

This phase checks whitespace rules. However, this does not include indentation.

9.3 Phase 3 - Vertical Spacing

This phase checks all vertical spacing requirements.

9.4 Phase 4 - Indentation

This phase checks all indentation rules.

9.5 Phase 5 - Alignment

This phase checks all alignment rules.

9.6 Phase 6 - Capitalization

This phase checks capitalization rules.

9.7 Phase 7 - Naming conventions

This phase checks naming conventions for signals, constants, ports, etc…

CHAPTER 10

Subphases

Each phase can have multiple subphases. There are rules which are executed within the same phase, but one is dependent on another. Utilizing a subphase allows for the proper execution of the rules.

10.1 Subphase 1

Prepare code for rules in subphase 2.

10.2 Subphase 2

Execute on code prepared in subphase 1.

CHAPTER 11

Rules

The rules are divided into catagories depending on the part of the VHDL code being operated on.

11.1 After Rules

11.1.1 after_001

This rule checks for **after x** in signal assignments in clock processes.

Note: All rules in this group are disabled by default. Use a configuration to enable them.

Violation

```
CLK_PROC : process(clock, reset) is
begin
  if (reset = '1') then
    a <= '0';
    b <= '1';
  elsif (clock'event and clock = '1') then
    a <= d;
    b <= c;
  end if;
end process CLK_PROC;
```

Fix

```
CLK_PROC : process(clock, reset) is
begin
  if (reset = '1') then
    a <= '0';
    b <= '1';
```

(continues on next page)

(continued from previous page)

```
  elsif (clock'event and clock = '1') then
    a <= d after 1 ns;
    b <= c after 1 ns;
  end if;
end process CLK_PROC;
```

Note: This rule has two configurable items:

- magnitude
- units

The **magnitude** is the number of units. Default is *1*.

The **units** is a valid time unit: ms, us, ns, ps etc... Default is *ns*.

11.1.2 after_002

This rule checks the *after* keywords are aligned in a clock process.

Note: All rules in this group are disabled by default. Use a configuration to enable them.

Violation

```
CLK_PROC : process(clock, reset) is
begin
  if (reset = '1') then
    a <= '0';
    b <= '1';
  elsif (clock'event and clock = '1') then
    a <= d     after 1 ns;
    b <= c   after 1 ns;
  end if;
end process CLK_PROC;
```

Fix

```
CLK_PROC : process(clock, reset) is
begin
  if (reset = '1') then
    a <= '0';
    b <= '1';
  elsif (clock'event and clock = '1') then
    a <= d     after 1 ns;
    b <= c     after 1 ns;
  end if;
end process CLK_PROC;
```

11.1.3 after_003

This rule checks the *after* keywords do not exist in the reset portion of a clock process.

Note: All rules in this group are disabled by default. Use a configuration to enable them.

Violation

```
CLK_PROC : process(clock, reset) is
begin
  if (reset = '1') then
    a <= '0' after 1 ns;
    b <= '1' after 1 ns;
  elsif (clock'event and clock = '1') then
    a <= d after 1 ns;
    b <= c after 1 ns;
  end if;
end process CLK_PROC;
```

Fix

```
CLK_PROC : process(clock, reset) is
begin
  if (reset = '1') then
    a <= '0';
    b <= '1';
  elsif (clock'event and clock = '1') then
    a <= d  after 1 ns;
    b <= c  after 1 ns;
  end if;
end process CLK_PROC;
```

11.2 Architecture Rules

11.2.1 architecture_001

This rule checks for blank spaces before the **architecture** keyword.

Violation

```
  architecture RTL of FIFO is
begin
```

Fix

```
architecture RTL of FIFO is
begin
```

11.2.2 architecture_002

This rule checks for a single space between **architecture**, **of**, and **is** keywords.

Violation

```
architecture  RTL  of    FIFO   is
```

Fix

```
architecture RTL of FIFO is
```

11.2.3 architecture_003

This rule check for a blank line above the **architecture** declaration.

Violation

```
library ieee;
architecture RTL of FIFO is
```

Fix

```
library ieee;

architecture RTL of FIFO is
```

11.2.4 architecture_004

This rule checks the proper case of the **architecture** keyword in the architecture declaration.

Note: The default is lowercase.

Refer to the section Configuring Uppercase and Lowercase Rules for information on changing the default case.

Violation

```
ARCHITECTURE RTL of FIFO is
```

```
architecture RTL of FIFO is
```

11.2.5 architecture_005

This rule checks the **of** keyword is on the same line as the **architecture** keyword.

Violation

```
architecture RTL
  of FIFO is
```

Fix

```
architecture RTL of FIFO is
```

11.2.6 architecture_006

This rule checks the **is** keyword is on the same line as the **architecture** keyword.

Violation

```
architecture RTL of FIFO
  is

architecture RTL of FIFO
```

Fix

```
architecture RTL of FIFO is

architecture RTL of FIFO is
```

11.2.7 architecture_007

This rule checks for spaces before the **begin** keyword.

Violation

```
architecture RTL of FIFO is
  begin
```

Fix

```
architecture RTL of FIFO is
begin
```

11.2.8 architecture_008

This rule checks for spaces before the **end architecture** keywords.

Violation

```
architecture RTL of FIFO is
begin
  end architecture
```

Fix

```
architecture RTL of FIFO is
begin
end architecture
```

11.2.9 architecture_009

This rule checks the **end** and **architecture** keywords are lower case.

Violation

```
END architecture;

end Architecture;
```

Fix

```
end architecture;

end architecture;
```

11.2.10 architecture_010

This rule checks for the keyword **architecture** in the **end architecture** statement. It is clearer to the reader to state what is ending.

Violation

```
end ARCHITECTURE_NAME;
```

Fix

```
end architecture ARCHITECTURE_NAME;
```

11.2.11 architecture_011

This rule checks the architecture name case in the **end architecture** statement.

Note: The default is uppercase.

Refer to the section Configuring Uppercase and Lowercase Rules for information on changing the default case.

Violation

```
end architecture architecture_name;
```

Fix

```
end architecture ARCHITECTURE_NAME;
```

11.2.12 architecture_012

This rule checks for a single space between **end** and **architecture** keywords.

Violation

```
end    architecture ARCHITECTURE_NAME;
```

Fix

```
end architecture ARCHITECTURE_NAME;
```

11.2.13 architecture_013

This rule checks the case of the architecture name in the architecture declaration.

Note: The default is uppercase.

Refer to the section Configuring Uppercase and Lowercase Rules for information on changing the default case.

Violation

```
architecture rtl of FIFO is
```

Fix

```
architecture RTL of FIFO is
```

11.2.14 architecture_014

This rule checks the case of the entity name in the architecture declaration.

Note: The default is uppercase.

Refer to the section Configuring Uppercase and Lowercase Rules for information on changing the default case.

Violation

```
architecture RTL of fifo is
```

Fix

```
architecture RTL of FIFO is
```

11.2.15 architecture_015

This rule check for a blank line below the architecture declaration.

Violation

```
architecture RTL of FIFO is
  signal wr_en : std_logic;
begin
```

Fix

```
architecture RTL of FIFO is

  signal wr_en : std_logic;
begin
```

11.2.16 architecture_016

This rule checks for a blank line above the **begin** keyword.

Violation

```
architecture RTL of FIFO is

  signal wr_en : std_logic;
begin
```

Fix

```
architecture RTL of FIFO is

  signal wr_en : std_logic;

begin
```

11.2.17 architecture_017

This rule checks for a blank line below the **begin** keyword.

Violation

```
begin
  wr_en <= '0';
```

Fix

```
begin

  wr_en <= '0';
```

11.2.18 architecture_018

This rule checks for a blank line above the **end architecture** declaration.

Violation

```
  rd_en <= '1';
end architecture RTL;
```

Fix

```
  rd_en <= '1';

end architecture RTL;
```

11.2.19 architecture_019

This rule checks the proper case of the **of** keyword in the architecture declaration.

Note: The default is lowercase.

Refer to the section Configuring Uppercase and Lowercase Rules for information on changing the default case.

Violation

```
architecture RTL OF FIFO is
```

Fix

```
architecture RTL of FIFO is
```

11.2.20 architecture_020

This rule checks the proper case of the **is** keyword in the architecture declaration.

Note: The default is lowercase.

Refer to the section Configuring Uppercase and Lowercase Rules for information on changing the default case.

Violation

```
architecture RTL of FIFO IS
```

Fix

```
architecture RTL of FIFO is
```

11.2.21 architecture_021

This rule checks the proper case of the **begin** keyword.

Note: The default is lowercase.

Refer to the section Configuring Uppercase and Lowercase Rules for information on changing the default case.

Violation

```
architecture RTL of FIFO is
BEGIN
```

Fix

```
architecture RTL of FIFO is
begin
```

11.2.22 architecture_022

This rule checks for a single space before the entity name in the end architecture declaration.

Violation

```
end architecture    FIFO;
```

Fix

```
end architecture FIFO;
```

11.2.23 architecture_023

This rule ensures the inline comments are aligned between the architecture declaration and the **begin** keyword.

Violation

```
architecture RTL of FIFO is

  signal wr_en : std_logic;    -- Enables writes to FIFO
  signal rd_en : std_logic;   -- Enables reads from FIFO
  signal overflow : std_logic;     -- Indicates the FIFO has overflowed when asserted

begin
```

Fix

```
architecture RTL of FIFO is

  signal wr_en : std_logic;        -- Enables writes to FIFO
  signal rd_en : std_logic;        -- Enables reads from FIFO
  signal overflow : std_logic;     -- Indicates the FIFO has overflowed when asserted

begin
```

11.2.24 architecture_024

This rule checks for the architecture name in the **end architecture** statement. It is clearer to the reader to state which architecture the end statement is closing.

Violation

```
end architecture;
```

Fix

```
end architecture ARCHITECTURE_NAME;
```

11.2.25 architecture_025

This rule checks for valid names for the architecture. Typical architecture names are: RTL, EMPTY, and BEHAVE. This rule allows the user to restrict what can be used for an architecture name.

Note: This rule is disabled by default. You can enable and configure the names using the following configuration.

```
---

rule :
  architecture_025 :
    disabled : False
    names :
```

(continues on next page)

(continued from previous page)

```
      - rtl
      - empty
      - behave
```

Violation

```
architecture SOME_INVALID_ARCH_NAME of ENTITY1 is
```

Fix

The user is required to decide which is the correct architecture name.

11.3 Assert Rules

11.3.1 assert_001

This rule checks alignment of multiline assert statements.

Violation

```
assert WIDTH > 16
     report "FIFO width is limited to 16 bits."
 severity FAILURE;
```

Fix

```
assert WIDTH > 16
  report "FIFO width is limited to 16 bits."
  severity FAILURE;
```

11.4 Attribute Rules

11.4.1 attribute_001

This rule checks the indent of **attribute** declarations.

Violation

```
architecture RTL of FIFO is

attribute ram_init_file : string;
attribute ram_init_file of ram_block :
      signal is "contents.mif";

begin
```

Fix

```
architecture RTL of FIFO is

  attribute ram_init_file : string;
```

(continues on next page)

(continued from previous page)

```
  attribute ram_init_file of ram_block :
    signal is "contents.mif";

begin
```

11.4.2 attribute_002

This rule checks the **attribute** keyword has proper case.

Note: The default is lowercase.

Refer to the section Configuring Uppercase and Lowercase Rules for information on changing the default case.

Violation

```
architecture RTL of FIFO is

  ATTRIBUTE ram_init_file : string;
  Attribute ram_init_file of ram_block :
    signal is "contents.mif";

begin
```

Fix

```
architecture RTL of FIFO is

  attribute ram_init_file : string;
  attribute ram_init_file of ram_block :
    signal is "contents.mif";

begin
```

11.4.3 attribute_003

This rule checks for a single space after the **attribute** keyword.

Violation

```
attribute   ram_init_file : string;
```

Fix

```
attribute ram_init_file : string;
```

11.5 Case Rules

11.5.1 case_001

This rule checks the indent of **case**, **when**, and **end case** keywords.

Violation

```
 case data is

     when 0 =>
 when 1 =>
          when 3 =>

end case;
```

Fix

```
case data is

  when 0 =>
  when 1 =>
  when 3 =>

end case;
```

11.5.2 case_002

This rule checks for a single space after the **case** keyword.

Violation

```
case    data is
```

Fix

```
case data is
```

11.5.3 case_003

This rule checks for a single space before the **is** keyword.

Violation

```
case data    is
```

Fix

```
case data is
```

11.5.4 case_004

This rule checks for a single space after the **when** keyword.

Violation

```
case data is

  when   3 =>
```

Fix

```
case data is

  when 3 =>
```

11.5.5 case_005

This rule checks for a single space before the **=>** operator.

Violation

```
case data is

  when 3   =>
```

Fix

```
case data is

  when 3 =>
```

11.5.6 case_006

This rule checks for a single space between the **end** and **case** keywords.

Violation

```
case data is

end    case;
```

Fix

```
case data is

end case;
```

11.5.7 case_007

This rule checks for a blank line before the **case** keyword. Comments are allowed before the **case** keyword.

Violation

```
a <= '1';
case data is


-- This is a comment
case data is
```

Fix

```
a <= '1';

case data is


-- This is a comment
case data is
```

11.5.8 case_008

This rule checks for a blank line below the **case** keyword.

Violation

```
case data is
  when 0 =>
```

Fix

```
case data is

  when 0 =>
```

11.5.9 case_009

This rule checks for a blank line above the **end case** keywords.

Violation

```
  when others =>
    null;
end case;
```

Fix

```
  when others =>
    null;

end case;
```

11.5.10 case_010

This rule checks for a blank line below the **end case** keywords.

Violation

```
end case;
a <= '1';
```

Fix

```
end case;

a <= '1';
```

11.5.11 case_011

This rule checks the alignment of multiline **when** statements.

Violation

```
case data is

  when 0 | 1 | 2 | 3
   4 | 5 | 7 =>
```

Fix

```
case data is

  when 0 | 1 | 2 | 3
       4 | 5 | 7 =>
```

11.5.12 case_012

This rule checks for code after the **=>** operator.

Violation

```
when 0 => a <= '1';
```

Fix

```
when 0 =>
  a <= '1';
```

11.5.13 case_013

This rule checks the indent of the **null** keyword.

Violation

```
  when others =>
     null;

  when others =>
null;
```

Fix

```
when others =>
  null;

when others =>
  null;
```

11.5.14 case_014

This rule checks the **case** keyword has proper case.

Note: The default is lowercase.

Refer to the section Configuring Uppercase and Lowercase Rules for information on changing the default case.

Violation

```
CASE address is

Case address is

case address is
```

Fix

```
case address is

case address is

case address is
```

11.5.15 case_015

This rule checks the **is** keyword has proper case.

Note: The default is lowercase.

Refer to the section Configuring Uppercase and Lowercase Rules for information on changing the default case.

Violation

```
case address IS

case address Is

case address iS
```

Fix

```
case address is

case address is

case address is
```

11.5.16 case_016

This rule checks the **when** has proper case.

Note: The default is lowercase.

Refer to the section Configuring Uppercase and Lowercase Rules for information on changing the default case.

Violation

```
WHEN a =>
When b =>
when c =>
```

Fix

```
when a =>
when b =>
when c =>
```

11.5.17 case_017

This rule checks the **end** keyword in the **end case** has proper case.

Note: The default is lowercase.

Refer to the section Configuring Uppercase and Lowercase Rules for information on changing the default case.

Violation

```
End case;
END case;
end case;
```

Fix

```
end case;
end case;
end case;
```

11.5.18 case_018

This rule checks the **case** keyword has proper case in the **end case**.

Note: The default is lowercase.

Refer to the section Configuring Uppercase and Lowercase Rules for information on changing the default case.

Violation

```
end CASE;
end CAse;
end case;
```

Fix

```
end case;
end case;
end case;
```

11.5.19 case_019

This rule checks for labels before the **case** keyword. The label should be removed. The preference is to have comments above the case statement.

Violation

```
CASE_LABEL : case address is
CASE_LABEL: case address is
case address is
```

Fix

```
case address is
case address is
case address is
```

11.5.20 case_020

This rule checks for labels after the **end case** keywords. The label should be removed. The preference is to have comments above the case statement.

Violation

```
end case CASE_LABEL;
end case;
```

Fix

```
end case;
end case;
```

11.6 Comment Rules

11.6.1 comment_001

This rule aligns comments above library use statements with the use statement.

Violation

```
library ieee;
-- Use standard logic library
  use ieee.std_logic_1164.all;
```

Fix

```
library ieee;
  -- Use standard logic library
  use ieee.std_logic_1164.all;
```

11.6.2 comment_003

This rule checks the alignment of in line comments between the process begin and end process lines.

Violation

```
PROC_1: process () is
begin

  a <= '1';    -- Assert
  b <= '0';       -- Deassert
  c <= '1'; -- Enable

end process PROC_1;
```

Fix

```
PROC_1: process () is
begin

  a <= '1';       -- Assert
  b <= '0';       -- Deassert
  c <= '1';       -- Enable

end process PROC_1;
```

11.6.3 comment_004

This rule checks for at least a single space before inline comments.

Violation

```
wr_en <= '1';--Write data
rd_en <= '1';    -- Read data
```

Fix

```
wr_en <= '1'; --Write data
rd_en <= '1';    -- Read data
```

11.6.4 comment_005

This rule aligns consecutive comment only lines above a **when** keyword in a case statement with the **when** keyword.

Violation

```
      -- comment 1
-- comment 2
   -- comment 3
  when wr_en =>
    rd_en <= '0';
```

Fix

```
-- comment 1
-- comment 2
-- comment 3
when wr_en =>
  rd_en <= '0';
```

11.6.5 comment_006

This rule aligns in line comments between the end of the process sensitivity list and the process **begin** keyword.

Violation

```
PROC_1 : process () is

   variable counter : integer range 0 to 31;      -- Counts the number of frames␣
↪received
   variable width   : natural range 0 to 255; -- Keeps track of the data word size

   variable size    : natural range 0 to 7; -- Keeps track of the frame size

begin
```

Fix

```
PROC_1 : process () is

   variable counter : integer range 0 to 31;      -- Counts the number of frames␣
↪received
   variable width   : natural range 0 to 255;     -- Keeps track of the data word size

   variable size    : natural range 0 to 7;       -- Keeps track of the frame size

begin
```

11.6.6 comment_008

This rule aligns consecutive comment only lines above the **elsif** keyword in if statements. These comments are used to describe what the elsif code is going to do.

Violation

```
      -- comment 1
-- comment 2
   -- comment 3
  elsif (a = '1')
    rd_en <= '0';
```

Fix

```
-- comment 1
-- comment 2
-- comment 3
elsif (a = '1')
  rd_en <= '0';
```

11.6.7 comment_009

This rule aligns consecutive comment only lines above the **else** keyword in if statements. These comments are used to describe what the elsif code is going to do.

Violation

```
      -- comment 1
-- comment 2
   -- comment 3
  else
    rd_en <= '0';
```

Fix

```
-- comment 1
-- comment 2
-- comment 3
else
  rd_en <= '0';
```

11.6.8 comment_010

This rule checks the indent lines starting with comments.

Violation

```
    -- Libraries
libary ieee;

 -- Define architecture
architecture RTL of FIFO is

-- Define signals
  signal wr_en : std_logic;
  signal rd_en : std_Logic;

begin
```

Fix

```
-- Libraries
libary ieee;

-- Define architecture
architecture RTL of FIFO is

  -- Define signals
  signal wr_en : std_logic;
  signal rd_en : std_Logic;

begin
```

11.7 Component Rules

11.7.1 component_001

This rule checks the indentation of the **component** keyword.

Violation

```
architecture RTL of FIFO is
begin

component FIFO is

     component RAM is
```

Fix

```
architecture RTL of FIFO is
begin

  component FIFO is

  component RAM is
```

11.7.2 component_002

This rule checks for a single space after the **component** keyword.

Violation

```
component    FIFO is
```

Fix

```
component FIFO is
```

11.7.3 component_003

This rule checks for a blank line above the **component** declaration.

Violation

```
end component FIFO;
component RAM is
```

Fix

```
end component FIFO;

component RAM is
```

11.7.4 component_004

This rule checks the **component** keyword has proper case.

Note: The default is lowercase.

Refer to the section Configuring Uppercase and Lowercase Rules for information on changing the default case.

Violation

```
COMPONENT FIFO is

Component FIFO is
```

Fix

```
component FIFO is

component FIFO is
```

11.7.5 component_005

This rule checks the **is** keyword is on the same line as the **component** keyword.

Violation

```
component FIFO

component FIFO
is
```

Fix

```
component FIFO is

component FIFO is
```

11.7.6 component_006

This rule checks the **is** keyword has proper case.

Note: The default is lowercase.

Refer to the section Configuring Uppercase and Lowercase Rules for information on changing the default case.

Violation

```
componet FIFO IS

componet FIFO Is
```

Fix

```
component FIFO is

component FIFO is
```

11.7.7 component_007

This rule checks for a single space before the **is** keyword.

Violation

```
component FIFO    is
```

Fix

```
component FIFO is
```

11.7.8 component_008

This rule checks the component name has proper case in the component declaration.

Note: The default is uppercase.

Refer to the section Configuring Uppercase and Lowercase Rules for information on changing the default case.

Violation

```
component fifo is
```

Fix

```
component FIFO is
```

11.7.9 component_009

This rule checks the indent of the **end component** keywords.

Violation

```
  OVERFLOW : std_logic
);
    end component FIFO;
```

Fix

```
    OVERFLOW : std_logic
  );
end component FIFO;
```

11.7.10 component_010

This rule checks the **end** keyword has proper case.

Note: The default is lowercase.

Refer to the section Configuring Uppercase and Lowercase Rules for information on changing the default case.

Violation

```
END component FIFO;
```

Fix

```
end component FIFO;
```

11.7.11 component_011

This rule checks for single space after the **end** keyword.

Violation

```
end   component FIFO;
```

Fix

```
end component FIFO;
```

11.7.12 component_012

This rule checks the proper case of the component name in the **end component** line.

Note: The default is uppercase.

Refer to the section Configuring Uppercase and Lowercase Rules for information on changing the default case.

Violation

```
end component fifo;
```

Fix

```
end component FIFO;
```

11.7.13 component_013

This rule checks for a single space after the **component** keyword in the **end component** line.

Violation

```
end component    FIFO;
```

Fix

```
end component FIFO;
```

11.7.14 component_014

This rule checks the **component** keyword in the **end component** line has proper case.

Note: The default is lowercase.

Refer to the section Configuring Uppercase and Lowercase Rules for information on changing the default case.

Violation

```
end COMPONENT FIFO;
```

Fix

```
end component FIFO;
```

11.7.15 component_015

This rule checks for the **component** keyword in the **end component** line.

Violation

```
end FIFO;

end;
```

Fix

```
end component FIFO;

end component;
```

11.7.16 component_016

This rule checks for blank lines above the **end component** line.

Violation

```
    OVERFLOW : std_logic
  );



end component FIFO;
```

Fix

```
    OVERFLOW : std_logic
  );
end component FIFO;
```

11.7.17 component_017

This rule checks the alignment of the : in port declarations.

Violation

```
RD_EN : in    std_logic;
WR_EN   : in    std_logic;
OVERFLOW : out   std_logic;
```

Fix

```
RD_EN    : in    std_logic;
WR_EN    : in    std_logic;
OVERFLOW : out   std_logic;
```

11.7.18 component_018

This rule checks for a blank line below the **end component** line.

Violation

```
end component FIFO;
signal rd_en : std_logic;
```

Fix

```
end component FIFO;

signal rd_en : std_logic;
```

11.7.19 component_019

This rule checks for comments at the end of the port and generic assignments in component declarations. These comments represent additional maintainence. They will be out of sync with the entity at some point. Refer to the entity for port types, port directions and purpose.

Violation

```
WR_EN : in    std_logic;  -- Enables write to RAM
RD_EN : out   std_logic; -- Enable reads from RAM
```

Fix

```
WR_EN : in    std_logic;
RD_EN : out   std_logic;
```

11.7.20 component_020

This rule checks the comments at the end of the port and generic assignments in component declarations are aligned. This rule is useful if component_019 is disabled.

Violation

```
WR_EN : in    std_logic;  -- Enables write to RAM
RD_EN : out   std_logic; -- Enable reads from RAM
```

Fix

```
WR_EN : in    std_logic;  -- Enables write to RAM
RD_EN : out   std_logic;  -- Enable reads from RAM
```

11.8 Concurrent Rules

11.8.1 concurrent_001

This rule checks the indent of concurrent assignments.

Violation

```
architecture RTL of FIFO is
begin

     wr_en <= '0';
rd_en <= '1';
```

Fix

```
architecture RTL of FIFO is
begin

  wr_en <= '0';
  rd_en <= '1';
```

11.8.2 concurrent_002

This rule checks for a single space after the **<=** operator.

Violation

```
wr_en <=    '0';
rd_en <=   '1';
```

Fix

```
wr_en <= '0';
rd_en <= '1';
```

11.8.3 concurrent_003

This rule checks alignment of multiline concurrent assignments. Succesive lines should align to the space after the assignment operator. However, there is a special case if there are parenthesis in the assignment. If the parenthesis are not closed on the same line, then the next line will be aligned to the parenthesis. Aligning to the parenthesis improves readability.

Violation

```
wr_en <= '0' when q_wr_en = '1' else
     '1';

w_foo <= I_FOO when ((I_BAR = '1') and
         (I_CRUFT = '1')) else
         '0';

O_FOO <= (1 => q_foo(63 downto 32),
         0 => q_foo(31 downto  0));

n_foo <= resize(unsigned(I_FOO) +
         unsigned(I_BAR), q_foo'length);
```

Fix

```
wr_en <= '0' when q_wr_en = '1' else
         '1';

w_foo <= I_FOO when ((I_BAR = '1') and
                     (I_CRUFT = '1')) else
         '0';

O_FOO <= (1 => q_foo(63 downto 32),
          0 => q_foo(31 downto  0));

n_foo <= resize(unsigned(I_FOO) +
                unsigned(I_BAR), q_foo'length);
```

11.8.4 concurrent_004

This rule checks for at least a single space before the **<=** operator.

Violation

```
wr_en<= '0';
```

Fix

```
wr_en <= '0';
```

11.8.5 concurrent_005

This rule checks for labels on concurrent assignments. Labels on concurrents are optional and do not provide additional information.

Violation

```
WR_EN_OUTPUT : WR_EN <= q_wr_en;
RD_EN_OUTPUT : RD_EN <= q_rd_en;
```

Fix

```
WR_EN <= q_wr_en;
RD_EN <= q_rd_en;
```

11.8.6 concurrent_006

This rule checks the alignment of the **<=** operator over multiple consecutive lines.

Violation

```
wr_en <= '0';
rd_en   <= '1';
data <= (others => '0');
```

Fix

```
wr_en   <= '0';
rd_en   <= '1';
data    <= (others => '0');
```

11.8.7 concurrent_007

This rule checks for code after the **else** keyword.

Violation

```
wr_en <= '0' when overflow = '0' else '1';
```

Fix

```
wr_en <= '0' when overflow = '0' else
         '1';
```

11.8.8 concurrent_008

This rule checks the alignment of inline comments in sequential concurrent statements.

Violation

```
wr_en   <= '0';     -- Write enable
rd_en   <= '1';  -- Read enable
data    <= (others => '0');   -- Write data
```

Fix

```
wr_en   <= '0';               -- Write enable
rd_en   <= '1';               -- Read enable
data    <= (others => '0');   -- Write data
```

11.9 Constant Rules

11.9.1 constant_001

This rule checks the indent of a constant declaration.

Violation

```
architecture RTL of FIFO is

constant size : integer := 1;
    constant width : integer := 32
```

Fix

```
architecture RTL of FIFO is

  constant size : integer := 1;
  constant width : integer := 32
```

11.9.2 constant_002

This rule checks the **constant** keyword is has proper case.

Note: The default is lowercase.

Refer to the section Configuring Uppercase and Lowercase Rules for information on changing the default case.

Violation

```
CONSTANT size : integer := 1;
```

Fix

```
constant size : integer := 1;
```

11.9.3 constant_003

This rule checks for spaces after the **constant** keyword.

Violation

```
constant    size : integeri := 1;
```

Fix

```
constant size : integer := 1;
```

Note: The number of spaces after the **constant** keyword is configurable. Use the following YAML file example to change the default number of spaces.

```
```

rule:

constant_003: spaces: 3

11.9.4 constant_004

This rule checks the constant name is lower case.

Violation

```
constant SIZE : integer := 1;
```

Fix

```
constant size : integer := 1;
```

11.9.5 constant_005

This rule checks for a single space after the :.

Violation

```
constant size  :integer := 1;
constant wdith :     integer := 32;
```

Fix

```
constant size  : integer := 1;
constant width : integer := 32;
```

11.9.6 constant_006

This rule checks for at least a single space before the :.

Violation

```
constant size: integer := 1;
constant width     : integer := 32;
```

Fix

```
constant size : integer := 1;
constant width     : integer := 32;
```

11.9.7 constant_007

This rule checks the **:=** is on the same line at the **constant** keyword.

Violation

```
constant size : integer
   := 1;
```

Fix

```
constant size : integer := 1;
```

11.9.8 constant_009

This rule checks the :'s are in the same column for all constants in the architecture declarative region.

Violation

```
constant size : integer := 1;
constant width   : integer := 32
```

Fix

```
constant size    : integer := 1;
constant width   : integer := 32
```

11.9.9 constant_010

This rule checks for a single space before the := keyword in constant declarations. Having a space makes it clearer where the assignment occurs on the line.

Violation

```
constant size : integer:= 1;
constant width : integer   := 10;
```

Fix

```
constant size : integer := 1;
constant width : integer := 10;
```

11.9.10 constant_011

This rule checks the constant type has proper case.

Note: The default is lowercase.

Refer to the section Configuring Uppercase and Lowercase Rules for information on changing the default case.

Violation

```
constant size : INTEGER := 1;
```

Fix

```
constant size : integer := 1;
```

11.9.11 constant_012

This rule checks the indent of multiline constants that contain arrays.

Violation

```
constant rom : romq_type :=
(
         0,
     65535,
     32768
  );
```

Fix

```
constant rom : romq_type :=
(
  0,
  65535,
  32768
);
```

11.9.12 constant_013

This rule checks for consistent capitalization of constant names.

Violation

```
architecture RTL of ENTITY1 is

  constant c_size  : integer := 5;
  constant c_ones  : std_logic_vector(c_size - 1 downto 0) := (others => '1');
  constant c_zeros : std_logic_vector(c_size - 1 downto 0) := (others => '0');

  signal data : std_logic_vector(c_size - 1 downto 0);

begin

  data <= C_ONES;

  PROC_NAME : process () is
  begin

    data <= C_ones;

    if (sig2 = '0') then
      data <= c_Zeros;
    end if;

  end process PROC_NAME;

end architecture RTL;
```

Fix

```
architecture RTL of ENTITY1 is

```

(continues on next page)

(continued from previous page)

```
  constant c_size  : integer := 5;
  constant c_ones  : std_logic_vector(c_size - 1 downto 0) := (others => '1');
  constant c_zeros : std_logic_vector(c_size - 1 downto 0) := (others => '0');

  signal data : std_logic_vector(c_size - 1 downto 0);

begin

  data <= c_ones;

  PROC_NAME : process () is
  begin

    data <= c_ones;

    if (sig2 = '0') then
      data <= c_zeros;
    end if;

  end process PROC_NAME;

end architecture RTL;
```

11.9.13 constant_014

This rule checks the indent of multiline constants that do not contain arrays.

Violation

```
constant width : integer := a + b +
  c + d;
```

Fix

```
constant width : integer := a + b +
                            c + d;
```

11.9.14 constant_015

This rule checks for valid prefixes on constant identifiers.

Note: The default constant prefix is "c_".

Refer to the section Configuring Prefix and Suffix Rules for information on changing the allowed prefixes.

Violation

```
constant my_const : integer;
```

Fix

```
constant c_my_const : integer;
```

11.10 Entity Rules

11.10.1 entity_001

This rule checks the indent of the **entity** keyword.

Violation

```
library ieee;

  entity FIFO is
```

Fix

```
library ieee;

entity FIFO is
```

11.10.2 entity_002

This rule checks for a single space after the **entity** keyword.

Violation

```
entity    FIFO is
```

Fix

```
entity FIFO is
```

11.10.3 entity_003

This rule checks for a blank line above the entity keyword.

Violation

```
library ieee;
entity FIFO is
```

Fix

```
library ieee;

entity FIFO is
```

11.10.4 entity_004

This rule checks the **entity** keyword has proper case.

Note: The default is lowercase.

Refer to the section Configuring Uppercase and Lowercase Rules for information on changing the default case.

Violation

```
ENTITY FIFO is
```

Fix

```
entity FIFO is
```

11.10.5 entity_005

This rule checks the **is** keyword is on the same line as the **entity** keyword.

Violation

```
entity FIFO

entity FIFO
  is
```

Fix

```
entity FIFO is

entity FIFO is
```

11.10.6 entity_006

This rule checks the **is** keyword has proper case in the entity declaration.

Note: The default is lowercase.

Refer to the section Configuring Uppercase and Lowercase Rules for information on changing the default case.

Violation

```
entity FIFO IS
```

Fix

```
entity FIFO is
```

11.10.7 entity_007

This rule checks for a single space before the **is** keyword.

Violation

```
entity FIFO    is
```

Fix

```
entity FIFO is
```

11.10.8 entity_008

This rule checks the entity name has proper case in the entity declaration.

Note: The default is uppercase.

Refer to the section Configuring Uppercase and Lowercase Rules for information on changing the default case.

Violation

```
entity fifo is
```

Fix

```
entity FIFO is
```

11.10.9 entity_009

This rule checks the indent of the **end** keyword.

Violation

```
  WR_EN : in    std_logic;
  RD_EN : in    std_logic
);
  end entity FIFO;
```

Fix

```
    WR_EN : in    std_logic;
    RD_EN : in    std_logic
  );
end entity FIFO;
```

11.10.10 entity_010

This rule checks the **end** keyword has proper case.

Note: The default is lowercase.

Refer to the section Configuring Uppercase and Lowercase Rules for information on changing the default case.

Violation

```
END entity FIFO;
```

Fix

```
end entity FIFO;
```

11.10.11 entity_011

This rule checks for a single space after the **end** keyword.

Violation

```
end    entity FIFO;
```

Fix

```
end entity FIFO;
```

11.10.12 entity_012

This rule checks the case of the entity name in the **end entity** statement.

Note: The default is uppercase.

Refer to the section Configuring Uppercase and Lowercase Rules for information on changing the default case.

Violation

```
end entity fifo;
```

Fix

```
end entity FIFO;
```

11.10.13 entity_013

This rule checks for a single space after the **entity** keyword in the closing of the entity declaration.

Violation

```
end entity    FIFO;
```

Fix

```
end entity FIFO;
```

11.10.14 entity_014

This rule checks the **entity** keyword has proper case in the closing of the entity declaration.

Note: The default is lowercase.

Refer to the section Configuring Uppercase and Lowercase Rules for information on changing the default case.

Violation

```
end ENTITY FIFO;
```

Fix

```
end entity FIFO;
```

11.10.15 entity_015

This rule checks for the keyword **entity** in the **end entity** statement.

Violation

```
end FIFO;

end;
```

Fix

```
end entity FIFO;

end entity;
```

11.10.16 entity_016

This rule checks for blank lines above the **end entity** keywords.

Violation

```
    WR_EN : in    std_logic;
    RD_EN : in    std_logic
  );


end entity FIFO;
```

Fix

```
    WR_EN : in    std_logic;
    RD_EN : in    std_logic
  );
end entity FIFO;
```

11.10.17 entity_017

This rule checks for alignment of the :'s in for every port in the entity.

Violation

```
WR_EN : in    std_logic;
RD_EN : in    std_logic;
OVERLFLOW : out   std_logic;
```

Fix

```
WR_EN     : in    std_logic;
RD_EN     : in    std_logic;
OVERLFLOW : out   std_logic;
```

11.10.18 entity_018

This rule checks for alignment of inline comments in the entity

Violation

```
WR_EN      : in    std_logic;       -- Wrte enable
RD_EN      : in    std_logic; -- Read enable
OVERLFLOW : out   std_logic;   -- FIFO has overflowed
```

Fix

```
WR_EN      : in    std_logic;       -- Wrte enable
RD_EN      : in    std_logic;       -- Read enable
OVERLFLOW : out   std_logic;       -- FIFO has overflowed
```

11.10.19 entity_019

This rule checks for the entity name in the **end entity** statement.

Violation

```
end entity;
```

Fix

```
end entity ENTITY_NAME;
```

11.11 File Rules

11.11.1 file_001

This rule checks the indent of **file** declarations.

Violation

```
architecture RTL of FIFO is

file defaultImage : load_file_type open read_mode is load_file_name;

file defaultImage : load_file_type open read_mode
is load_file_name;

begin
```

Fix

```
architecture RTL of FIFO is

  file defaultImage : load_file_type open read_mode is load_file_name;

  file defaultImage : load_file_type open read_mode
    is load_file_name;

begin
```

11.11.2 file_002

This rule checks the **file** keyword has proper case.

Note: The default is lowercase.

Refer to the section Configuring Uppercase and Lowercase Rules for information on changing the default case.

Violation

```
architecture RTL of FIFO is

  FILE defaultImage : load_file_type open read_mode is load_file_name;

begin
```

Fix

```
architecture RTL of FIFO is

  file defaultImage : load_file_type open read_mode is load_file_name;

begin
```

11.11.3 file_003

This rule checks for spaces after the **file** keyword.

Violation

```
file   defaultImage : load_file_type open read_mode is load_file_name;
```

Fix

```
file defaultImage : load_file_type open read_mode is load_file_name;
```

Note: The number of spaces after the **file** keyword is configurable. Use the following YAML file example to change the default number of spaces.

```

```

rule:

  file_003: spaces: 3

11.12 For Loop Rules

11.12.1 for_loop_001

This rule checks the indentation of the **for** keyword.

Violation

```
FIFO_PROC : process () is
begin

for index in 4 to 23 loop

  end loop;

end process;
```

Fix

```
FIFO_PROC : process () is
begin

  for index in 4 to 23 loop

  end loop;

end process;
```

11.12.2 for_loop_002

This rule checks the indentation of the **end loop** keywords.

Violation

```
FIFO_PROC : process () is
begin

  for index in 4 to 23 loop

     end loop;

end process;
```

Fix

```
FIFO_PROC : process () is
begin

  for index in 4 to 23 loop

  end loop;

end process;
```

11.12.3 for_loop_003

This rule checks the proper case of the label on a foor loop.

Note: The default is uppercase.

Refer to the section Configuring Uppercase and Lowercase Rules for information on changing the default case.

Violation

```
label : for index in 4 to 23 loop
Label : for index in 0 to 100 loop
```

Fix

```
LABEL : for index in 4 to 23 loop
LABEL : for index in 0 to 100 loop
```

11.12.4 for_loop_004

This rule checks if a label exists on a for loop that a single space exists between the label and the :.

Violation

```
LABEL: for index in 4 to 23 loop
LABEL    : for index in 0 to 100 loop
```

Fix

```
LABEL : for index in 4 to 23 loop
LABEL : for index in 0 to 100 loop
```

11.12.5 for_loop_005

This rule checks if a label exists on a for loop that a single space exists after the :.

Violation

```
LABEL :    for index in 4 to 23 loop
LABEL :  for index in 0 to 100 loop
```

Fix

```
LABEL : for index in 4 to 23 loop
LABEL : for index in 0 to 100 loop
```

11.13 Function Rules

11.13.1 function_001

This rule checks the indentation of the **function** keyword.

Violation

```
architecture RTL of FIFO is

    function overflow (a: integer) return integer is


function underflow (a: integer) return integer is
```

(continues on next page)

(continued from previous page)

```
begin
```

Fix

```
architecture RTL of FIFO is

  function overflow (a: integer) return integer is

  function underflow (a: integer) return integer is

begin
```

11.13.2 function_002

This rule checks a single space exists after the **function** keyword.

Violation

```
function     overflow (a: integer) return integer is
```

Fix

```
function overflow (a: integer) return integer is
```

11.13.3 function_003

This rule checks for a single space after the function name and the (.'

Violation

```
function overflow   (a: integer) return integer is

function underflow(a: integer) return integer is
```

Fix

```
function overflow (a: integer) return integer is

function underflow (a: integer) return integer is
```

11.13.4 function_004

This rule checks the **begin** keyword has proper case.

Note: The default is lowercase.

Refer to the section Configuring Uppercase and Lowercase Rules for information on changing the default case.

Violation

```
function overflow (a: integer) return integer is
BEGIN
```

Fix

```
function overflow (a: integer) return integer is
begin
```

11.13.5 function_005

This rule checks the **function** keyword has proper case.

Note: The default is lowercase.

Refer to the section Configuring Uppercase and Lowercase Rules for information on changing the default case.

Violation

```
FUNCTION overflow (a: integer) return integer is
```

Fix

```
function overflow (a: integer) return integer is
```

11.13.6 function_006

This rule checks for a blank line above the **function** keyword.

Violation

```
architecture RTL of FIFO is
  function overflow (a: integer) return integer is
```

Fix

```
architecture RTL of FIFO is

  function overflow (a: integer) return integer is
```

11.13.7 function_007

This rule checks for a blank line below the end of the function declaration.

Violation

```
function overflow (a: integer) return integer is
end;
signal wr_en : std_logic;
```

Fix

```
function overflow (a: integer) return integer is
end;

signal wr_en : std_logic;
```

11.13.8 function_008

This rule checks the indent of function parameters on multiple lines.

Violation

```
function func_1 (a : integer; b : integer;
            c : unsigned(3 downto 0);
    d : std_logic_vector(7 downto 0);
       e : std_logic) return integer is
begin

end;
```

Fix

```
function func_1 (a : integer; b : integer;
  c : unsigned(3 downto 0);
  d : std_logic_vector(7 downto 0);
  e : std_logic) return integer is
begin

end;
```

11.13.9 function_009

This rule checks for a function parameter on the same line as the function keyword when the parameters are on multiple lines.

Violation

```
function func_1 (a : integer; b : integer;
  c : unsigned(3 downto 0);
  d : std_logic_vector(7 downto 0);
  e : std_logic) return integer is
begin

end;
```

Fix

```
function func_1 (
  a : integer; b : integer;
  c : unsigned(3 downto 0);
  d : std_logic_vector(7 downto 0);
  e : std_logic) return integer is
begin

end;
```

11.13.10 function_010

This rule checks for consistent capitalization of function names.

Violation

```
architecture RTL of FIFO is

  function func_1 ()

begin

  OUT1 <= Func_1;

  PROC1 : process () is
  begin

     sig1 <= FUNC_1;

  end process;

end architecture RTL;
```

Violation

```
architecture RTL of FIFO is

  function func_1 ()

begin

  OUT1 <= func_1;

  PROC1 : process () is
  begin

     sig1 <= func_1;

  end process;

end architecture RTL;
```

11.14 Generate Rules

11.14.1 generate_001

This rule checks the indent of the generate declaration.

Violation

```
architecture RTL of FIFO is
begin

RAM_ARRAY: for i in 0 to 7 generate

      RAM_ARRAY: for i in 0 to 7 generate
```

Fix

```
architecture RTL of FIFO is
begin

  RAM_ARRAY: for i in 0 to 7 generate

  RAM_ARRAY: for i in 0 to 7 generate
```

11.14.2 generate_002

This rule checks for a single space between the label and the :.

Violation

```
RAM_ARRAY: for i in 0 to 7 generate
```

Fix

```
RAM_ARRAY : for i in 0 to 7 generate
```

11.14.3 generate_003

This rule checks for a blank line after the **end generate** keywords.

Violation

```
end generate RAM_ARRAY;
wr_en <= '1';
```

Fix

```
end generate RAM_ARRAY;

wr_en <= '1';
```

11.14.4 generate_004

This rule checks for a blank line before the **generate** keyword.

Violation

```
wr_en <= '1';
RAM_ARRAY: for i in 0 to 7 generate
```

Fix

```
wr_en <= '1';

RAM_ARRAY: for i in 0 to 7 generate
```

11.14.5 generate_005

This rule checks the generate label has proper case.

Note: The default is uppercase.

Refer to the section Configuring Uppercase and Lowercase Rules for information on changing the default case.

Violation

```
ram_array: for i in 0 to 7 generate
```

Fix

```
RAM_ARRAY: for i in 0 to 7 generate
```

11.14.6 generate_006

This rule checks the indent of the **begin** keyword.

Violation

```
RAM_ARRAY: for i in 0 to 7 generate
   begin
```

Fix

```
RAM_ARRAY: for i in 0 to 7 generate
begin
```

11.14.7 generate_007

This rule checks the indent of the **end generate** keyword.

Violation

```
RAM_ARRAY: for i in 0 to 7 generate
begin
  end generate RAM_ARRAY;
```

Fix

```
RAM_ARRAY: for i in 0 to 7 generate
begin
end generate RAM_ARRAY;
```

11.14.8 generate_008

This rule checks for a single space after the **end** keyword.

Violation

```
end   generate RAM_ARRAY;
```

Fix

```
end generate RAM_ARRAY;
```

11.14.9 generate_009

This rule checks the **end** keyword has proper case.

Note: The default is lowercase.

Refer to the section Configuring Uppercase and Lowercase Rules for information on changing the default case.

Violation

```
END generate RAM_ARRAY;
```

Fix

```
end generate RAM_ARRAY;
```

11.14.10 generate_010

This rule checks the **generate** keyword has the proper case in the **end generate** line.

Note: The default is lowercase.

Refer to the section Configuring Uppercase and Lowercase Rules for information on changing the default case.

Violation

```
end GENERATE RAM_ARRAY;
```

Fix

```
end generate RAM_ARRAY;
```

11.14.11 generate_011

This rule checks the **end generate** line has a label.

Violation

```
end generate;
```

Fix

```
end generate RAM_ARRAY;
```

11.14.12 generate_012

This rule checks the **end generate** label has proper case.

Note: The default is uppercase.

Refer to the section Configuring Uppercase and Lowercase Rules for information on changing the default case.

Violation

```
end generate ram_array;
```

Fix

```
end generate RAM_ARRAY;
```

11.14.13 generate_013

This rule checks for a single space after the **generate** keyword and the label in the **end generate** keywords.

Violation

```
end generate    RAM_ARRAY;
```

Fix

```
end generate RAM_ARRAY;
```

11.14.14 generate_014

This rule checks for a single space between the : and the **for** keyword.

Violation

```
RAM_ARRAY :for i in 0 to 7 generate
RAM_ARRAY :   for i in 0 to 7 generate
```

Fix

```
RAM_ARRAY : for i in 0 to 7 generate
RAM_ARRAY : for i in 0 to 7 generate
```

11.14.15 generate_015

This rule checks the generate label and the **generate** keyword are on the same line. Keeping the label and generate on the same line reduces excessive indenting.

Violation

```
RAM_ARRAY :
  for i in 0 to 7 generate
```

Fix

```
RAM_ARRAY : for i in 0 to 7 generate
```

11.14.16 generate_016

This rule checks the alignment of the **when** keyword in generic case statements.

Violation

```
GEN_LABEL : case condition generate
  when 0 =>
    when 1 =>
when 2 =>
```

Fix .. code-block:: vhdl

    GEN_LABEL [case condition generate] when 0 => when 1 => when 2 =>

11.15 Generic Rules

11.15.1 generic_001

This rule checks for blank lines above the **generic** keyword.

Violation

```
entity FIFO is



  generic (
```

Fix

```
entity FIFO is
  generic (
```

11.15.2 generic_002

This rule checks the indent of the **generic** keyword.

Violation

```
entity FIFO is
     generic (

entity FIFO is
generic (
```

Fix

```
entity FIFO is
  generic (

entity FIFO is
  generic (
```

11.15.3 generic_003

This rule checks for a single space between the **generic** keyword and the (.

Violation

```
generic     (

generic(
```

Fix

```
generic (

generic (
```

11.15.4 generic_004

This rule checks the indent of generic declarations.

Violation

```
generic (
WIDTH : integer := 32;
       DEPTH : integer := 512
)
```

Fix

```
generic (
  WIDTH : integer := 32;
  DEPTH : integer := 512
)
```

11.15.5 generic_005

This rule checks for a single space after the colon in a generic declaration.

Violation

```
WIDTH :integer := 32;
```

Fix

```
WIDTH : integer := 32;
```

11.15.6 generic_006

This rule checks for a single space after the default assignment.

Violation

```
WIDTH : integer :=32;
DEPTH : integer :=     512;
```

Fix

```
WIDTH : integer := 32;
DEPTH : integer := 512;
```

11.15.7 generic_007

This rule checks the generic names have proper case.

Note: The default is uppercase.

Violation

```
width : integer := 32;
```

Fix

```
WIDTH : integer := 32;
```

11.15.8 generic_008

This rule checks the indent of the closing parenthesis.

Violation

```
DEPTH : integer := 512
);
```

Fix

```
  DEPTH : integer := 512
);
```

11.15.9 generic_009

This rule checks the **generic** keyword has proper case.

Note: The default is lowercase.

Refer to the section Configuring Uppercase and Lowercase Rules for information on changing the default case.

Violation

```
GENERIC (
```

Fix

```
generic (
```

11.15.10 generic_010

This rule checks the closing parenthesis is on a line by itself.

Violation

```
DEPTH : integer := 512);
```

Fix

```
  DEPTH : integer := 512
);
```

11.15.11 generic_012

This rule checks the alignment of :'s for every generic.

Violation

```
ADDRESS_WIDTH : integer := 10;
DATA_WIDTH : integer := 32;
DEPTH : integer := 512;
```

Fix

```
ADDRESS_WIDTH : integer := 10;
DATA_WIDTH    : integer := 32;
DEPTH         : integer := 512;
```

11.15.12 generic_013

This rule checks for the **generic** keyword on the same line as a generic declaration.

Violation

```
generic (DEPTH : integer := 512;
```

Fix

```
generic (
  DEPTH : integer := 512;
```

11.15.13 generic_014

This rule checks for at least a single space before the :.

Violation

```
ADDRESS_WIDTH: integer := 10;
DATA_WIDTH : integer := 32;
DEPTH: integer := 512;
```

Fix

```
ADDRESS_WIDTH : integer := 10;
DATA_WIDTH : integer := 32;
DEPTH : integer := 512;
```

11.15.14 generic_015

This rule checks the alignment of the **:=** operator in generic declarations.

Violation

```
ADDRESS_WIDTH : integer    := 10;
DATA_WIDTH    : integer := 32;
DEPTH         : integer  := 512;
```

Fix

```
ADDRESS_WIDTH : integer    := 10;
DATA_WIDTH    : integer    := 32;
DEPTH         : integer    := 512;
```

11.15.15 generic_016

This rule checks for multiple generics defined on a single line.

Violation

```
generic (
  WIDTH : std_logic := '0';DEPTH : std_logic := '1'
);
```

Fix

```
generic (
  WIDTH : std_logic := '0';
  DEPTH : std_logic := '1'
);
```

11.15.16 generic_017

This rule checks the generic type has proper case if it is a VHDL keyword.

Note: The default is lowercase.

Violation

```
generic (
  WIDTH : STD_LOGIC := '0';
  DEPTH : Std_logic := '1'
);
```

Fix

```
generic (
  WIDTH : std_logic := '0';
  DEPTH : std_logic := '1'
);
```

11.15.17 generic_018

This rule checks the **generic** keyword is on the same line as the (.

Violation

```
generic
 (
```

Fix

```
generic (
```

11.15.18 generic_019

This rule checks for blank lines before the); of the generic declaration.

Violation

```
generic (
  WIDTH : std_logic := '0';
  DEPTH : Std_logic := '1'


);
```

Fix

```
generic (
  WIDTH : std_logic := '0';
  DEPTH : Std_logic := '1'
);
```

11.15.19 generic_020

This rule checks for valid prefixes on generic identifiers.

Note: The default generic prefix is "G_".

Refer to the section Configuring Prefix and Suffix Rules for information on changing the allowed prefixes.

Violation

```
generic(MY_GEN : integer);
```

Fix

```
generic(G_MY_GEN : integer);
```

11.16 If Rules

11.16.1 if_001

This rule checks the indent of the **if** keyword.

Violation

```
  if (a = '1') then
  b <= '0'
elsif (c = '1') then
  d <= '1';
else
  e <= '0';
end if;
```

Fix

```
if (a = '1') then
  b <= '0'
elsif (c = '1') then
  d <= '1';
else
  e <= '0';
end if;
```

11.16.2 if_002

This rule checks the bolean expression is enclosed in ().

Violation

```
if a = '1' then
```

Fix

```
if (a = '1') then
```

11.16.3 if_003

This rule checks for a single space between the **if** keyword and the (.

Violation

```
if(a = '1') then

if    (a = '1') then
```

Fix

```
if (a = '1') then

if (a = '1') then
```

11.16.4 if_004

This rule checks for a single space between the) and the **then** keyword.

Violation

```
if (a = '1')then

if (a = '1')     then
```

Fix

```
if (a = '1') then

if (a = '1') then
```

11.16.5 if_005

This rule checks for a single space between the **elsif** keyword and the (.

Violation

```
elsif(c = '1') then

elsif    (c = '1') then
```

Fix

```
elsif (c = '1') then

elsif (c = '1') then
```

11.16.6 if_006

This rule checks for empty lines after the **then** keyword.

Violation

```
if (a = '1') then


  b <= '0'
```

Fix

```
if (a = '1') then
  b <= '0'
```

11.16.7 if_007

This rule checks for empty lines before the **elsif** keyword.

Violation

```
  b <= '0'



elsif (c = '1') then
```

Fix

```
  b <= '0'
elsif (c = '1') then
```

11.16.8 if_008

This rule checks for empty lines before the **end if** keywords.

Violation

```
  e <= '0';


end if;
```

Fix

```
  e <= '0';
end if;
```

11.16.9 if_009

This rule checks the alignment of multiline boolean expressions.

Violation

```
if (a = '0' and b = '1' and
      c = '0') then
```

Fix

```
if (a = '0' and b = '1' and
    c = '0') then
```

11.16.10 if_010

This rule checks for empty lines before the **else** keyword.

Violation

```
  d <= '1';


else
```

Fix

```
  d <= '1';
else
```

11.16.11 if_011

This rule checks for empty lines after the **else** keyword.

Violation

```
else


  e <= '0';
```

Fix

```
else
  e <= '0';
```

11.16.12 if_012

This rule checks the indent of the **elsif** keyword.

Violation

```
if (a = '1') then
  b <= '0'
  elsif (c = '1') then
  d <= '1';
else
  e <= '0';
end if;
```

Fix

```
if (a = '1') then
  b <= '0'
elsif (c = '1') then
  d <= '1';
else
  e <= '0';
end if;
```

11.16.13 if_013

This rule checks the indent of the **else** keyword.

Violation

```
if (a = '1') then
  b <= '0'
elsif (c = '1') then
  d <= '1';
  else
  e <= '0';
end if;
```

Fix

```
if (a = '1') then
  b <= '0'
elsif (c = '1') then
  d <= '1';
else
  e <= '0';
end if;
```

11.16.14 if_014

This rule checks the indent of the **end if** keyword.

Violation

```
if (a = '1') then
  b <= '0'
elsif (c = '1') then
  d <= '1';
else
  e <= '0';
  end if;
```

Fix

```
if (a = '1') then
  b <= '0'
elsif (c = '1') then
  d <= '1';
else
  e <= '0';
end if;
```

11.16.15 if_015

This rule checks for a single space between the **end if** keywords.

Violation

```
end    if;
```

Fix

```
end if;
```

11.16.16 if_020

This rule checks the **end if** keyword is on it's own line.

Violation

```
if (a = '1') then c <= '1'; else c <= '0'; end if;
```

Fix

```
if (a = '1') then c <= '1'; else c <= '0';
end if;
```

11.16.17 if_021

This rule checks the **else** keyword is on it's own line.

Violation

```
if (a = '1') then c <= '1'; else c <= '0'; end if;
```

Fix

```
if (a = '1') then c <= '0';
else c <= '1'; end if;
```

11.16.18 if_022

This rule checks for code after the **else** keyword.

Violation

```
if (a = '1') then c <= '1'; else c <= '0'; end if;
```

Fix

```
if (a = '1') then c <= '1'; else
  c <= '0'; end if;
```

11.16.19 if_023

This rule checks the **elsif** keyword is on it's own line.

Violation

```
if (a = '1') then c <= '1'; else c <= '0'; elsif (b = '0') then d <= '0'; end if;
```

Fix

```
if (a = '1') then c <= '1'; else c <= '0';
elsif (b = '0') then d <= '0'; end if;
```

11.16.20 if_024

This rule checks for code after the **then** keyword.

Violation

```
if (a = '1') then c <= '1';
```

Fix

```
if (a = '1') then
  c <= '1';
```

11.16.21 if_025

This rule checks the **if** keyword has proper case.

Note: The default is lowercase.

Refer to the section Configuring Uppercase and Lowercase Rules for information on changing the default case.

Violation

```
IF (a = '1') then
```

Fix

```
if (a = '1') then
```

11.16.22 if_026

This rule checks the **elsif** keyword has proper case.

Note: The default is lowercase.

Refer to the section Configuring Uppercase and Lowercase Rules for information on changing the default case.

Violation

```
ELSIF (a = '1') then
```

Fix

```
elsif (a = '1') then
```

11.16.23 if_027

This rule checks the **else** keyword has proper case.

Note: The default is lowercase.

Refer to the section Configuring Uppercase and Lowercase Rules for information on changing the default case.

Violation

```
ELSE
```

Fix

```
else
```

11.16.24 if_028

This rule checks the **end if** keywords have proper case.

Note: The default is lowercase.

Violation

```
END if;

end IF;

END IF;
```

Fix

```
end if;

end if;

end if;
```

11.16.25 if_029

This rule checks the **then** keyword has proper case.

Note: The default is lowercase.

Violation

```
if (a = '1') THEN
```

Fix

```
if (a = '1') then
```

11.16.26 if_030

This rule checks for at least a single blank line after the **end if**. In the case of nested **if** statements, the rule will be enfoced on the last **end if**.

Violation

```
if (A = '1') then
  B <= '0';
end if;
C <= '1';
```

Fix

```
if (A = '1') then
  B <= '0';
end if;

C <= '1';
```

11.16.27 if_031

This rule checks for at least a single blank line before the **if**, unless there is a comment. In the case of nested **if** statements, the rule will be enfoced on the first **if**.

Violation

```
C <= '1';
if (A = '1') then
  B <= '0';
end if;

-- This is a comment
if (A = '1') then
  B <= '0';
end if;
```

Fix

```
C <= '1';

if (A = '1') then
  B <= '0';
end if;

-- This is a comment
if (A = '1') then
  B <= '0';
end if;
```

11.17 Instantiation Rules

11.17.1 instantiation_001

This rule checks for the proper indentation of instantiations.

Violation

```
   U_FIFO : FIFO
port map (
         WR_EN    => wr_en,
 RD_EN    => rd_en,
       OVERFLOW => overflow
              );
```

Fix

```
U_FIFO : FIFO
  port map (
    WR_EN    => wr_en,
    RD_EN    => rd_en,
    OVERFLOW => overflow
  );
```

11.17.2 instantiation_002

This rule checks for a single space after the :.

Violation

```
U_FIFO :FIFO
```

Fix

```
U_FIFO : FIFO
```

11.17.3 instantiation_003

This rule checks for a single space before the :.

Violation

```
U_FIFO: FIFO
```

Fix

```
U_FIFO : FIFO
```

11.17.4 instantiation_004

This rule checks for a blank line above the instantiation.

Note: Comments are allowed above the instantiation.

Violation

```
WR_EN <= '1';
U_FIFO : FIFO

-- Instantiate another FIFO
U_FIFO2 : FIFO
```

Fix

```
WR_EN <= '1';

U_FIFO : FIFO

-- Instantiate another FIFO
U_FIFO2 : FIFO
```

11.17.5 instantiation_005

This rule checks the instantiation declaration and the **port map** keywords are not on the same line.

Violation

```
U_FIFO : FIFO port map (
```

Fix

```
U_FIFO : FIFO
  port map (
```

11.17.6 instantiation_006

This rule checks the **port map** keywords have proper case.

Note: The default is lowercase.

Violation

```
PORT MAP (
```

Fix

```
port map (
```

11.17.7 instantiation_007

This rule checks the closing) for the port map is on it's own line.

Violation

```
WR_EN => wr_en);
```

Fix

```
  WR_EN => wr_en
);
```

11.17.8 instantiation_008

This rule checks the instance name has proper case.

Note: The default is uppercase.

Refer to the section Configuring Uppercase and Lowercase Rules for information on changing the default case.

Violation

```
u_fifo : FIFO
```

Fix

```
U_FIFO : FIFO
```

11.17.9 instantiation_009

This rule checks the entity name has proper case.

Note: The default is uppercase.

Violation

```
U_FIFO : fifo
```

Fix

```
U_FIFO : FIFO
```

11.17.10 instantiation_010

This rule checks the alignment of the **=>** operator for every port in instantiation.

Violation

```
U_FIFO : FIFO
  port map (
    WR_EN => wr_en,
    RD_EN => rd_en,
    OVERFLOW => overflow
  );
```

Fix

```
U_FIFO : FIFO
  port map (
    WR_EN    => wr_en,
    RD_EN    => rd_en,
    OVERFLOW => overflow
  );
```

11.17.11 instantiation_011

This rule checks the port name is uppercase. Indexes on ports will not be uppercased.

Violation

```
U_FIFO : FIFO
  port map (
    wr_en              => wr_en,
    rd_en              => rd_en,
    OVERFLOW           => overflow,
    underflow(c_index) => underflow
  );
```

Fix

```
U_FIFO : FIFO
  port map (
    WR_EN              => wr_en,
    RD_EN              => rd_en,
    OVERFLOW           => overflow,
    UNDERFLOW(c_index) => underflow
  );
```

11.17.12 instantiation_012

This rule checks the instantiation declaration and the **generic map** keywords are not on the same line.

Violation

```
U_FIFO : FIFO generic map (
```

Fix

```
U_FIFO : FIFO
  generic map (
```

11.17.13 instantiation_013

This rule checks the **generic map** keywords have proper case.

Note: The default is lowercase.

Violation

```
GENERIC MAP (
```

Fix

```
generic map (
```

11.17.14 instantiation_014

This rule checks for the closing parenthesis *)* on generic maps are on their own line.

Violation

```
INSTANCE_NAME : ENTITY_NAME
  generic map (
    GENERIC_1 => 0,
    GENERIC_2 => TRUE,
    GENERIC_3 => FALSE)
```

Fix

```
INSTANCE_NAME : ENTITY_NAME
  generic map (
    GENERIC_1 => 0,
    GENERIC_2 => TRUE,
    GENERIC_3 => FALSE
  )
```

11.17.15 instantiation_015

This rule checks the alignment of the **=>** operator for every generic.

Violation

```
U_FIFO : FIFO
  generic map (
    DEPTH => 512,
    WIDTH    => 32
  )
```

Fix

```
U_FIFO : FIFO
  generic map (
    DEPTH    => 512,
    WIDTH    => 32
  )
```

11.17.16 instantiation_016

This rule checks generic names have proper case.

Note: The default is uppercase.

Violation

```
U_FIFO : FIFO
  generic map (
    depth => 512,
    width => 32
  )
```

Fix

```
U_FIFO : FIFO
  generic map (
    DEPTH => 512,
    WIDTH => 32
  )
```

11.17.17 instantiation_017

This rule checks if the **generic map** keywords and a generic assignment are on the same line.

Violation

```
generic map (DEPTH => 512,
  WIDTH => 32
)
```

Fix

```
generic map (
  DEPTH => 512,
  WIDTH => 32
)
```

11.17.18 instantiation_018

This rule checks for a single space between the **map** keyword and the (.

Violation

```
generic map(

generic map    (
```

Fix

```
generic map (

generic map (
```

11.17.19 instantiation_019

This rule checks for a blank line below the end of the instantiation declaration.

Violation

```
U_FIFO : FIFO
  port map (
    WR_EN    => wr_en,
    RD_EN    => rd_en,
    OVERFLOW => overflow
  );
U_RAM : RAM
```

Fix

```
U_FIFO : FIFO
  port map (
    WR_EN    => wr_en,
    RD_EN    => rd_en,
    OVERFLOW => overflow
  );

U_RAM : RAM
```

11.17.20 instantiation_020

This rule checks for a port assignment on the same line as the **port map** keyword.

Violation

```
U_FIFO : FIFO
  port map (WR_EN    => wr_en,
    RD_EN    => rd_en,
    OVERFLOW => overflow
  );
```

Fix

```
U_FIFO : FIFO
  port map (
    WR_EN    => wr_en,
    RD_EN    => rd_en,
    OVERFLOW => overflow
  );
```

11.17.21 instantiation_021

This rule checks multiple port assignments on the same line.

Violation

```
port map (
  WR_EN => w_wr_en, RD_EN => w_rd_en,
  OVERFLOW => w_overflow
);
```

Fix

```
port map (
  WR_EN => w_wr_en,
  RD_EN => w_rd_en,
  OVERFLOW => w_overflow
);
```

11.17.22 instantiation_022

This rule checks for a single space after the **=>** operator in port maps.

Violation

```
U_FIFO : FIFO
  port map (
    WR_EN    =>   wr_en,
    RD_EN    =>rd_en,
    OVERFLOW =>     overflow
  );
```

Fix

```
U_FIFO : FIFO
  port map (
    WR_EN    => wr_en,
    RD_EN    => rd_en,
    OVERFLOW => overflow
  );
```

11.17.23 instantiation_023

This rule checks for comments at the end of the port and generic assignments in instantiations. These comments represent additional maintainence. They will be out of sync with the entity at some point. Refer to the entity for port types, port directions and purpose.

Violation

```
WR_EN => w_wr_en;    -- out : std_logic
RD_EN => w_rd_en;    -- Reads data when asserted
```

Fix

```
WR_EN => w_wr_en;
RD_EN => w_rd_en;
```

11.17.24 instantiation_024

This rule checks for positional generics and ports. Positional ports and generics are subject to problems when the position of the underlying component changes.

Violation

```
port map (
  WR_EN, RD_EN, OVERFLOW
);
```

Fix

Use explicit port mapping.

```
port map (
  WR_EN    => WR_EN;
  RD_EN    => RD_EN;
  OVERFLOW => OVERFLOW
);
```

11.17.25 instantiation_025

This rule checks the (is on the same line as the **port map** keywords.

Violation

```
port map
(
  WR_EN    => WR_EN,
  RD_EN    => RD_EN,
  OVERFLOW => OVERFLOW
);
```

Fix

Use explicit port mapping.

```
port map (
  WR_EN    => WR_EN,
  RD_EN    => RD_EN,
  OVERFLOW => OVERFLOW
);
```

11.17.26 instantiation_026

This rule checks the (is on the same line as the **generic map** keywords.

Violation

```
generic map
(
  WIDTH => 32,
  DEPTH => 512
)
```

Fix

Use explicit port mapping.

```
generic map (
  WIDTH => 32,
  DEPTH => 512
)
```

11.17.27 instantiation_027

This rule checks the **entity** keyword has proper case in direct instantiations.

Note: The default is lowercase.

Violation

```
INSTANCE_NAME : ENTITY library.ENTITY_NAME
```

Fix

```
INSTANCE_NAME : entity library.ENTITY_NAME
```

11.17.28 instantiation_028

This rule checks the entity name has proper case in direct instantiations.

Note: The default is uppercase.

Violation

```
INSTANCE_NAME : entity library.entity_name
```

Fix

```
INSTANCE_NAME : entity library.ENTITY_NAME
```

11.17.29 instantiation_029

This rule checks for alignment of inline comments in an instantiation

Violation

```
WR_EN     => write_enable,         -- Wrte enable
RD_EN     => read_enable,     -- Read enable
OVERLFLOW => overflow,          -- FIFO has overflowed
```

Fix

```
WR_EN     => write_enable,         -- Wrte enable
RD_EN     => read_enable,          -- Read enable
OVERLFLOW => overflow,             -- FIFO has overflowed
```

11.17.30 instantiation_030

This rule checks for a single space after the **=>** keyword in generic maps.

Violation

```
generic map
(
  WIDTH =>    32,
  DEPTH => 512
)
```

Fix

```
generic map
(
  WIDTH => 32,
  DEPTH => 512
)
```

11.17.31 instantiation_031

This rule checks the component keyword has proper case in component instantiations that use the **component** keyword.

Note: The default is lowercase.

Violation

```
INSTANCE_NAME : COMPONENT ENTITY_NAME
```

Fix

```
INSTANCE_NAME : component ENTITY_NAME
```

Note: This rule is off by default. If this rule is desired, then enable this rule and disable instantiation_033.

```
{
  "rule":{
    "instantiation_031":{
       "disable":"False"
    },
    "instantiation_033":{
       "disable":"True"
    }
  }
}
```

11.17.32 instantiation_032

This rule checks for a single space after the **component** keyword if it is used.

Violation

```
INSTANCE_NAME : component ENTITY_NAME
INSTANCE_NAME : component   ENTITY_NAME
INSTANCE_NAME : component  ENTITY_NAME
```

Fix

```
INSTANCE_NAME : component ENTITY_NAME
INSTANCE_NAME : component ENTITY_NAME
INSTANCE_NAME : component ENTITY_NAME
```

Note: This rule is off by default. If this rule is desired, then enable this rule and disable instantiation_033.

```
{
  "rule":{
    "instantiation_032":{
       "disable":"False"
    },
    "instantiation_033":{
       "disable":"True"
    }
  }
}
```

11.17.33 instantiation_033

This rule checks for the **component** keyword and will remove it.

The component keyword is optional and does not provide clarity.

Violation

```
INSTANCE_NAME : component ENTITY_NAME
```

Fix

```
INSTANCE_NAME : ENTITY_NAME
```

11.18 Length Rules

These rules cover the length of lines in the VHDL file.

11.18.1 length_001

This rule checks the length of the line.

Violation

```
wr_en <= '1' when a = '1' else '0' when b = '0' else c when d = '1' else f; -- This␣
↪is a comment.
```

Fix

Note: The user must fix this violation. Refer to the section Configuring Line Length for information on changing the default.

11.19 Library Rules

11.19.1 library_001

This rule checks the indent of the **library** keyword. Indenting helps in comprehending the code.

Violation

```
library ieee;
   library fifo_dsn;
```

Fix

```
library ieee;
library fifo_dsn;
```

11.19.2 library_002

This rule checks for excessive spaces after the **library** keyword.

Violation

```
library    ieee;
```

Fix

```
library ieee;
```

11.19.3 library_003

This rule checks for a blank line above the **library** keyword.

Violation

```
library ieee;
library fifo_dsn;
```

Fix

```
library ieee;

library fifo_dsn;
```

11.19.4 library_004

This rule checks the **library** keyword has proper case.

Note: The default is lowercase.

Refer to the section Configuring Uppercase and Lowercase Rules for information on changing the default case.

Violation

```
Library ieee;

LIBRARY fifo_dsn;
```

Fix

```
library ieee;

library fifo_dsn;
```

11.19.5 library_005

This rule checks the **use** keyword has proper case.

Note: The default is lowercase.

Refer to the section Configuring Uppercase and Lowercase Rules for information on changing the default case.

Violation

```
library ieee;
  USE ieee.std_logic_1164.all;
  Use ieee.std_logic_unsigned.all;
```

Fix

```
library ieee;
  use ieee.std_logic_1164.all;
  use ieee.std_logic_unsigned.all;
```

11.19.6 library_006

This rule checks for excessive spaces after the **use** keyword.

Violation

```
library ieee;
  use    ieee.std_logic_1164.all;
  use   ieee.std_logic_unsigned.all;
```

Fix

```
library ieee;
  use ieee.std_logic_1164.all;
  use ieee.std_logic_unsigned.all;
```

11.19.7 library_007

This rule checks for blank lines above the **use** keyword.

Violation

```
library ieee;

  use ieee.std_logic_1164.all;

  use ieee.std_logic_unsigned.all;
```

Fix

```
library ieee;
  use ieee.std_logic_1164.all;
  use ieee.std_logic_unsigned.all;
```

11.19.8 library_008

This rule checks the indent of the **use** keyword.

Violation

```
library ieee;
use ieee.std_logic_1164.all;
     use ieee.std_logic_unsigned.all;
```

Fix

```
library ieee;
  use ieee.std_logic_1164.all;
  use ieee.std_logic_unsigned.all;
```

11.20 Package Rules

11.20.1 package_001

This rule checks the indent of the package declaration.

Violation

```
library ieee;

  package FIFO_PKG is
```

Fix

```
library ieee;

package FIFO_PKG is
```

11.20.2 package_002

This rule checks for a single space between **package** and **is** keywords.

Violation

```
package   FIFO_PKG   is
```

Fix

```
package FIFO_PKG is
```

11.20.3 package_003

This rule checks for a blank line above the **package** keyword.

Violation

```
library ieee;
package FIFO_PKG is
```

Fix

```
library ieee;

package FIFO_PKG is
```

11.20.4 package_004

This rule checks the package keyword has proper case.

Note: The default is lowercase.

Refer to the section Configuring Uppercase and Lowercase Rules for information on changing the default case.

Violation

```
PACKAGE FIFO_PKG is
```

Fix

```
package FIFO_PKG is
```

11.20.5 package_005

This rule checks the **is** keyword is on the same line as the **package** keyword.

Violation

```
package FIFO_PKG
is
```

Fix

```
package FIFO_PKG is
```

11.20.6 package_006

This rule checks the **end package** keywords have proper case.

Note: The default is lowercase.

Violation

```
END PACKAGE FIFO_PKG;
```

Fix

```
end package FIFO_PKG;
```

11.20.7 package_007

This rule checks for the **package** keyword on the end package declaration.

Violation

```
end FIFO_PKG;
```

Fix

```
end package FIFO_PKG;
```

11.20.8 package_008

This rule checks the package name has proper case on the end package declaration.

Note: The default is uppercase.

Violation

```
end package fifo_pkg;
```

Fix

```
end package FIFO_PKG;
```

11.20.9 package_009

This rule checks for a single space between the **end** and **package** keywords and package name.

Violation

```
end   package   FIFO_PKG;
```

Fix

```
end package FIFO_PKG;
```

11.20.10 package_010

This rule checks the package name has proper case in the package declaration.

Note: The default is uppercase.

Violation

```
package fifo_pkg is
```

Fix

```
package FIFO_PKG is
```

11.20.11 package_011

This rule checks for a blank line below the **package** keyword.

Violation

```
package FIFO_PKG is
  constant width : integer := 32;
```

Fix

```
package FIFO_PKG is

  constant width : integer := 32;
```

11.20.12 package_012

This rule checks for a blank line above the **end package** keyword.

Violation

```
  constant depth : integer := 512;
end package FIFO_PKG;
```

Fix

```
  constant depth : integer := 512;

end package FIFO_PKG;
```

11.20.13 package_013

This rule checks the **is** keyword has proper case.

Note: The default is lowercase.

Violation

```
package FIFO_PKG IS
```

Fix

```
package FIFO_PKG is
```

11.20.14 package_014

This rule checks the package name exists on the same line as the **end package** keywords.

Violation

```
end package;
```

Fix

```
end package FIFO_PKG;
```

11.20.15 package_015

This rule checks the indent of the end package declaration.

Violation

```
package FIFO_PKG is

   end package FIFO_PKG;
```

Fix

```
package FIFO_PKG is

end package FIFO_PKG;
```

11.21 Port Rules

11.21.1 port_001

This rule checks for a blank line above the **port** keyword.

Violation

```
entity FIFO is

  port (
```

Fix

```
entity FIFO is
  port (
```

11.21.2 port_002

This rule checks the indent of the **port** keyword.

Violation

```
entity FIFO is
port (
```

Fix

```
entity FIFO is
  port (
```

11.21.3 port_003

This rule checks for a single space after the **port** keyword and (.

Violation

```
port    (

port(
```

Fix

```
port (

port (
```

11.21.4 port_004

This rule checks the indent of port declarations.

Violation

```
port (
WR_EN    : in    std_logic;
     RD_EN    : in    std_logic;
  OVERFLOW : out   std_logic
);
```

Fix

```
port (
  WR_EN    : in    std_logic;
  RD_EN    : in    std_logic;
  OVERFLOW : out   std_logic
);
```

11.21.5 port_005

This rule checks for a single space after the : in **in** and **inout** ports.

Violation

```
port (
  WR_EN     : in    std_logic;
  RD_EN     :   in    std_logic;
  OVERFLOW : out   std_logic;
  DATA     :inout std_logic
);
```

Fix

```
port (
  WR_EN     : in    std_logic;
  RD_EN     : in    std_logic;
  OVERFLOW : out   std_logic;
  DATA     : inout std_logic
);
```

11.21.6 port_006

This rule checks for a single space after the : in the **out** ports.

Violation

```
port (
  WR_EN     : in    std_logic;
  RD_EN     : in    std_logic;
  OVERFLOW :out   std_logic
);
```

Fix

```
port (
  WR_EN     : in    std_logic;
  RD_EN     : in    std_logic;
  OVERFLOW : out   std_logic
);
```

11.21.7 port_007

This rule checks for four spaces after the **in** keyword.

Violation

```
port (
  WR_EN     : in std_logic;
  RD_EN     : in        std_logic;
  OVERFLOW : out   std_logic
);
```

Fix

```
port (
  WR_EN    : in    std_logic;
  RD_EN    : in    std_logic;
  OVERFLOW : out   std_logic
);
```

11.21.8 port_008

This rule checks for three spaces after the **out** keyword.

Violation

```
port (
  WR_EN    : in    std_logic;
  RD_EN    : in    std_logic;
  OVERFLOW : out std_logic
);
```

Fix

```
port (
  WR_EN    : in    std_logic;
  RD_EN    : in    std_logic;
  OVERFLOW : out   std_logic
);
```

11.21.9 port_009

This rule checks for a single space after the **inout** keyword.

Violation

```
port (
  WR_EN    : in    std_logic;
  RD_EN    : in    std_logic;
  DATA     : inout    std_logic
);
```

Fix

```
port (
  WR_EN    : in    std_logic;
  RD_EN    : in    std_logic;
  DATA     : inout std_logic
);
```

11.21.10 port_010

This rule checks port names are uppercase. If an index exists on a port, the case of the index will not be checked.

Violation

```
port (
  wr_en              : in    std_logic;
  rd_en              : in    std_logic;
  OVERFLOW           : out   std_logic;
  underflow(c_index) : out   std_logic
);
```

Fix

```
port (
  WR_EN              : in    std_logic;
  RD_EN              : in    std_logic;
  OVERFLOW           : out   std_logic;
  UNDERFLOW(c_index) : out   std_logic
);
```

11.21.11 port_011

This rule checks for valid prefixes on port identifiers.

Note: The default port prefixes are "I_", "O_", "IO_".

Refer to the section Configuring Prefix and Suffix Rules for information on changing the allowed prefixes.

Violation

```
port (
  WR_EN    : in    std_logic;
  RD_EN    : in    std_logic;
  OVERFLOW : out   std_logic;
  DATA     : inout std_logic
);
```

Fix

```
port (
  I_WR_EN    : in    std_logic;
  I_RD_EN    : in    std_logic;
  O_OVERFLOW : out   std_logic;
  IO_DATA    : inout std_logic
);
```

11.21.12 port_012

This rule checks for default assignments on port declarations.

Violation

```
port (
  I_WR_EN    : in    std_logic := '0';
  I_RD_EN    : in    std_logic := '0';
  O_OVERFLOW : out   std_logic;
  IO_DATA    : inout std_logic := (others => 'Z')
);
```

Fix

```
port (
  WR_EN_I     : in    std_logic;
  RD_EN_I     : in    std_logic;
  OVERFLOW_O  : out   std_logic;
  DATA_IO     : inout std_logic
);
```

11.21.13 port_013

This rule checks for multiple ports declared on a single line.

Violation

```
port (
  WR_EN     : in    std_logic;RD_EN     : in    std_logic;
  OVERFLOW  : out   std_logic;DATA      : inout std_logic
);
```

Fix

```
port (
  WR_EN     : in    std_logic;
  RD_EN     : in    std_logic;
  OVERFLOW  : out   std_logic;
  DATA      : inout std_logic
);
```

11.21.14 port_014

This rule checks the closing parenthesis of the port map are on a line by itself.

Violation

```
port (
  WR_EN     : in    std_logic;
  RD_EN     : in    std_logic;
  OVERFLOW  : out   std_logic;
  DATA      : inout std_logic);
```

Fix

```
port (
  WR_EN     : in    std_logic;
  RD_EN     : in    std_logic;
  OVERFLOW  : out   std_logic;
  DATA      : inout std_logic
);
```

11.21.15 port_015

This rule checks the indent of the closing parenthesis for port maps.

Violation

```
port (
  WR_EN     : in    std_logic;
  RD_EN     : in    std_logic;
  OVERFLOW  : out   std_logic;
  DATA      : inout std_logic
  );
```

Fix

```
port (
  WR_EN     : in    std_logic;
  RD_EN     : in    std_logic;
  OVERFLOW  : out   std_logic;
  DATA      : inout std_logic
);
```

11.21.16 port_016

This rule checks for a port definition on the same line as the **port** keyword.

Violation

```
port (WR_EN     : in    std_logic;
  RD_EN     : in    std_logic;
  OVERFLOW  : out   std_logic;
  DATA      : inout std_logic
);
```

Fix

```
port (
  WR_EN     : in    std_logic;
  RD_EN     : in    std_logic;
  OVERFLOW  : out   std_logic;
  DATA      : inout std_logic
);
```

11.21.17 port_017

This rule checks the **port** keyword has proper case.

Note: The default is lowercase.

Refer to the section Configuring Uppercase and Lowercase Rules for information on changing the default case.

Violation

```
PORT (
```

Fix

```
port (
```

11.21.18 port_018

This rule checks the port type has proper case if it is a VHDL keyword.

Note: The default is lowercase.

Violation

```
port (
  WR_EN     : in    STD_LOGIC;
  RD_EN     : in    std_logic;
  OVERFLOW  : out   t_OVERFLOW;
  DATA      : inout STD_LOGIC_VECTOR(31 downto 0)
);
```

Fix

```
port (
  WR_EN     : in    std_logic;
  RD_EN     : in    std_logic;
  OVERFLOW  : out   t_OVERFLOW;
  DATA      : inout std_logic_vector(31 downto 0)
);
```

11.21.19 port_019

This rule checks the port direction has proper case.

Note: The default is lowercase.

Violation

```
port (
  WR_EN     : IN    std_logic;
  RD_EN     : in    std_logic;
  OVERFLOW  : OUT   std_logic;
  DATA      : INOUT std_logic
);
```

Fix

```
port (
  WR_EN     : in    std_logic;
  RD_EN     : in    std_logic;
  OVERFLOW  : out   std_logic;
  DATA      : inout std_logic
);
```

11.21.20 port_020

This rule checks for at least one space before the :.

Violation

```
port (
  WR_EN     : in    std_logic;
  RD_EN     : in    std_logic;
  OVERFLOW: out   std_logic;
  DATA      : inout std_logic
);
```

Fix

```
port (
  WR_EN     : in    std_logic;
  RD_EN     : in    std_logic;
  OVERFLOW : out   std_logic;
  DATA      : inout std_logic
);
```

11.21.21 port_021

This rule checks the **port** keyword is on the same line as the (.

Violation

```
port
(
```

Fix

```
port (
```

11.21.22 port_022

This rule checks for blank lines after the **port** keyword.

Violation

```
port (


  WR_EN     : in    std_logic;
  RD_EN     : in    std_logic;
  OVERFLOW: out   std_logic;
  DATA      : inout std_logic
);
```

Fix

```
port (
  WR_EN     : in    std_logic;
  RD_EN     : in    std_logic;
  OVERFLOW : out   std_logic;
  DATA      : inout std_logic
);
```

11.21.23 port_023

This rule checks for missing modes in port declarations.

Note: This must be fixed by the user. VSG makes no assumption on the direction of the port.

Violation

```
port (
  WR_EN     : std_logic;
  RD_EN     : std_logic;
  OVERFLOW  : std_logic;
  DATA      : inout std_logic
);
```

Fix

```
port (
  WR_EN     : in    std_logic;
  RD_EN     : in    std_logic;
  OVERFLOW  : out   std_logic;
  DATA      : inout std_logic
);
```

11.21.24 port_024

This rule checks for blank lines before the close parenthesis in port declarations.

Violation

```
port (
  WR_EN     : std_logic;
  RD_EN     : std_logic;
  OVERFLOW  : std_logic;
  DATA      : inout std_logic


);
```

Fix

```
port (
  WR_EN     : in    std_logic;
  RD_EN     : in    std_logic;
  OVERFLOW  : out   std_logic;
  DATA      : inout std_logic
);
```

11.21.25 port_025

This rule checks for valid suffixes on port identifiers.

Note: The default port suffixes are "_I", "_O", "_IO".

Refer to the section Configuring Prefix and Suffix Rules for information on changing the allowed suffixes.

Violation

```
port (
  WR_EN    : in    std_logic;
  RD_EN    : in    std_logic;
  OVERFLOW : out   std_logic;
  DATA     : inout std_logic
);
```

Fix

```
port (
  WR_EN_I    : in    std_logic;
  RD_EN_I    : in    std_logic;
  OVERFLOW_O : out   std_logic;
  DATA_IO    : inout std_logic
);
```

11.22 Procedure Rules

There are three forms a procedure: with parameters, without parameters, and a package declaration:

with parameters

```
procedure AVERAGE_SAMPLES (
   constant a : in integer;
   signal b : in std_logic;
   variable c : in std_logic_vector(3 downto 0);
   signal d : out std_logic) is
begin
end procedure AVERAGE_SAMPLES;
```

without parameters

```
procedure AVERAGE_SAMPLES is
begin
end procedure AVERAGE_SAMPLES;
```

package declaration

```
procedure AVERAGE_SAMPLES;

procedure AVERAGE_SAMPLES (
   constant a : in integer;
   signal b : in std_logic;
   variable c : in std_logic_vector(3 downto 0);
   signal d : out std_logic);
```

11.22.1 procedure_001

This rule checks the indent of the **procedure** keyword.

Violation

```
  procedure AVERAGE_SAMPLES (
  constant a : in integer;
  signal b : in std_logic;
  variable c : in std_logic_vector(3 downto 0);
  signal d : out std_logic ) is
begin
end procedure AVERAGE_SAMPLES;
```

Fix

```
procedure AVERAGE_SAMPLES (
  constant a : in integer;
  signal b : in std_logic;
  variable c : in std_logic_vector(3 downto 0);
  signal d : out std_logic ) is
begin
end procedure AVERAGE_SAMPLES;
```

11.22.2 procedure_002

This rule checks the indent of the **begin** keyword.

Violation

```
procedure AVERAGE_SAMPLES (
  constant a : in integer;
  signal b : in std_logic;
  variable c : in std_logic_vector(3 downto 0);
  signal d : out std_logic ) is
  begin
end procedure AVERAGE_SAMPLES;
```

Fix

```
procedure AVERAGE_SAMPLES (
  constant a : in integer;
  signal b : in std_logic;
  variable c : in std_logic_vector(3 downto 0);
  signal d : out std_logic ) is
begin
end procedure AVERAGE_SAMPLES;
```

11.22.3 procedure_003

This rule checks the indent of the **end** keyword.

Violation

```
procedure AVERAGE_SAMPLES (
  constant a : in integer;
  signal b : in std_logic;
  variable c : in std_logic_vector(3 downto 0);
  signal d : out std_logic ) is
begin
  end procedure AVERAGE_SAMPLES;
```

Fix

```
procedure AVERAGE_SAMPLES (
  constant a : in integer;
  signal b : in std_logic;
  variable c : in std_logic_vector(3 downto 0);
  signal d : out std_logic ) is
begin
end procedure AVERAGE_SAMPLES;
```

11.22.4 procedure_004

This rule checks the indent of parameters.

Violation

```
procedure AVERAGE_SAMPLES (
constant a : in integer;
    signal b : in std_logic;
   variable c : in std_logic_vector(3 downto 0);
 signal d : out std_logic ) is
begin
end procedure AVERAGE_SAMPLES;
```

Fix

```
procedure AVERAGE_SAMPLES (
  constant a : in integer;
  signal b : in std_logic;
  variable c : in std_logic_vector(3 downto 0);
  signal d : out std_logic ) is
begin
end procedure AVERAGE_SAMPLES;
```

11.22.5 procedure_005

This rule checks the indent of line between the **is** and **begin** keywords

Violation

```
procedure AVERAGE_SAMPLES (
  constant a : in integer;
  signal d : out std_logic ) is
variable var_1 : integer;
    variable var_1 : integer;
begin
end procedure AVERAGE_SAMPLES;
```

Fix

```
procedure AVERAGE_SAMPLES (
  constant a : in integer;
  signal b : in std_logic;
  variable c : in std_logic_vector(3 downto 0);
  signal d : out std_logic ) is
```

(continues on next page)

(continued from previous page)

```
  variable var_1 : integer;
  variable var_1 : integer;
begin
end procedure AVERAGE_SAMPLES;
```

11.22.6 procedure_006

This rule checks the indent of the closing parenthesis if it is on it's own line.

Violation

```
procedure AVERAGE_SAMPLES (
  constant a : in integer;
  signal d : out std_logic
  ) is
```

Fix

```
procedure AVERAGE_SAMPLES (
  constant a : in integer;
  signal d : out std_logic
) is
```

11.22.7 procedure_007

This rule checks for consistent capitalization of procedure names.

Violation

```
architecture RTL of ENTITY1 is

  procedure AVERAGE_SAMPLES (
    constant a : in integer;
    signal d : out std_logic
  ) is

begin

  PROC1 : process () is
  begin

    Average_samples();

  end process PROC1;

end architecture RTL;
```

Fix

```
architecture RTL of ENTITY1 is

  procedure AVERAGE_SAMPLES (
    constant a : in integer;
    signal d : out std_logic
```

(continues on next page)

(continued from previous page)

```
  ) is

begin

  PROC1 : process () is
  begin

    AVERAGE_SAMPLES();

  end process PROC1;

end architecture RTL;
```

11.23 Process Rules

11.23.1 process_001

This rule checks the indent of the process declaration.

Violation

```
architecture RTL of FIFO is

begin

PROC_A : process (rd_en, wr_en, data_in, data_out,
```

Fix

```
architecture RTL of FIFO is

begin

  PROC_A : process (rd_en, wr_en, data_in, data_out,
```

11.23.2 process_002

This rule checks for a single space after the **process** keyword.

Violation

```
PROC_A : process(rd_en, wr_en, data_in, data_out,

PROC_A : process    (rd_en, wr_en, data_in, data_out,
```

Fix

```
PROC_A : process (rd_en, wr_en, data_in, data_out,

PROC_A : process (rd_en, wr_en, data_in, data_out,
```

11.23.3 process_003

This rule checks the indent of the **begin** keyword.

Violation

```
PROC_A : process (rd_en, wr_en, data_in, data_out,
                  rd_full, wr_full
                 ) is
  begin
```

Fix

```
PROC_A : process (rd_en, wr_en, data_in, data_out,
                  rd_full, wr_full
                 ) is
begin
```

11.23.4 process_004

This rule checks the **begin** keyword has proper case.

Note: The default is lowercase.

Violation

```
PROC_A : process (rd_en, wr_en, data_in, data_out,
                  rd_full, wr_full
                 ) is
BEGIN
```

Fix

```
PROC_A : process (rd_en, wr_en, data_in, data_out,
                  rd_full, wr_full
                 ) is
begin
```

11.23.5 process_005

This rule checks the **process** keyword has proper case.

Note: The default is lowercase.

Violation

```
PROC_A : PROCESS (rd_en, wr_en, data_in, data_out,
```

Fix

```
PROC_A : process (rd_en, wr_en, data_in, data_out,
```

11.23.6 process_006

This rule checks the indent of the **end process** keywords.

Violation

```
PROC_A : process (rd_en, wr_en, data_in, data_out,
                  rd_full, wr_full
                 ) is
begin

  end process PROC_A;
```

Fix

```
PROC_A : process (rd_en, wr_en, data_in, data_out,
                  rd_full, wr_full
                 ) is
begin

end process PROC_A;
```

11.23.7 process_007

This rule checks for a single space after the **end** keyword.

Violation

```
end   process PROC_A;
```

Fix

```
end process PROC_A;
```

11.23.8 process_008

This rule checks the **end** keyword has proper case.

Note: The default is lowercase.

Refer to the section Configuring Uppercase and Lowercase Rules for information on changing the default case.

Violation

```
END process PROC_A;
```

Fix

```
end process PROC_A;
```

11.23.9 process_009

This rule checks the **process** keyword has proper case in the **end process** line.

Note: The default is lowercase.

Violation

```
end PROCESS PROC_A;
```

Fix

```
end process PROC_A;
```

11.23.10 process_010

This rule checks the **begin** keyword is on it's own line.

Violation

```
PROC_A : process (rd_en, wr_en, data_in, data_out,
                  rd_full, wr_full
                 ) is begin
```

Fix

```
PROC_A : process (rd_en, wr_en, data_in, data_out,
                  rd_full, wr_full
                 ) is
begin
```

11.23.11 process_011

This rule checks for a blank line after the **end process** keyword.

Violation

```
end process PROC_A;
WR_EN <= wr_en;
```

Fix

```
end process PROC_A;

WR_EN <= wr_en;
```

11.23.12 process_012

This rule checks for the existence of the **is** keyword on the same line as the closing parenthesis of the sensitivity list.

Violation

```
PROC_A : process (rd_en, wr_en, data_in, data_out,
                  rd_full, wr_full
                 )
begin

PROC_A : process (rd_en, wr_en, data_in, data_out,
                  rd_full, wr_full
                 )
is begin
```

Fix

```
PROC_A : process (rd_en, wr_en, data_in, data_out,
                  rd_full, wr_full
                 ) is
begin

PROC_A : process (rd_en, wr_en, data_in, data_out,
                  rd_full, wr_full
                 ) is
begin
```

11.23.13 process_013

This rule checks the **is** keyword has proper case.

Note: The default is lowercase.

Violation

```
PROC_A : process (rd_en, wr_en, data_in, data_out,
                  rd_full, wr_full
                 ) IS
begin
```

Fix

```
PROC_A : process (rd_en, wr_en, data_in, data_out,
                  rd_full, wr_full
                 ) is
begin
```

11.23.14 process_014

This rule checks for a single space before the **is** keyword.

Violation

```
PROC_A : process (rd_en, wr_en, data_in, data_out,
                  rd_full, wr_full
                 )     is
begin
```

Fix

```
PROC_A : process (rd_en, wr_en, data_in, data_out,
                  rd_full, wr_full
                 ) is
begin
```

11.23.15 process_015

This rule checks for a blank line or comment above the **process** declaration.

Violation

```
-- This process performs FIFO operations.
PROC_A : process (rd_en, wr_en, data_in, data_out,

WR_EN <= wr_en;
PROC_A : process (rd_en, wr_en, data_in, data_out,
```

Fix

```
-- This process performs FIFO operations.
PROC_A : process (rd_en, wr_en, data_in, data_out,

WR_EN <= wr_en;

PROC_A : process (rd_en, wr_en, data_in, data_out,
```

11.23.16 process_016

This rule checks the process has a label.

Violation

```
process (rd_en, wr_en, data_in, data_out,
         rd_full, wr_full
        ) is
begin
```

Fix

```
PROC_A : process (rd_en, wr_en, data_in, data_out,
                  rd_full, wr_full
                 ) is
begin
```

11.23.17 process_017

This rule checks the process label has proper case.

Note: The default is uppercase.

Violation

```
proc_a : process (rd_en, wr_en, data_in, data_out,
                  rd_full, wr_full
                 ) is
begin
```

Fix

```
PROC_A : process (rd_en, wr_en, data_in, data_out,
                  rd_full, wr_full
                 ) is
begin
```

11.23.18 process_018

This rule checks the **end process** line has a label. The closing label will be added if the opening process label exists.

Violation

```
end process;
```

Fix

```
end process PROC_A;
```

11.23.19 process_019

This rule checks the **end process** label is uppercase.

Violation

```
end process proc_a;
```

Fix

```
end process PROC_A;
```

11.23.20 process_020

This rule checks the indentation of multiline sensitivity lists.

Violation

```
PROC_A : process (rd_en, wr_en, data_in, data_out,
                     rd_full, wr_full,
             overflow, underflow
                 ) is begin
```

Fix

```
PROC_A : process (rd_en, wr_en, data_in, data_out,
                  rd_full, wr_full,
                  overflow, underflow
                 ) is
begin
```

11.23.21 process_021

This rule checks for blank lines between the end of the sensitivity list and before the **begin** keyword.

Violation

```
PROC_A : process (rd_en, wr_en, data_in, data_out,
                  rd_full, wr_full
                 ) is



begin
```

Fix

```
PROC_A : process (rd_en, wr_en, data_in, data_out,
                  rd_full, wr_full
                 ) is
begin
```

11.23.22 process_022

This rule checks for a blank line below the **begin** keyword.

Violation

```
PROC_A : process (rd_en, wr_en, data_in, data_out,
                  rd_full, wr_full
                 ) is
begin
  rd_en <= '0';
```

Fix

```
PROC_A : process (rd_en, wr_en, data_in, data_out,
                  rd_full, wr_full
                 ) is
begin

  rd_en <= '0';
```

11.23.23 process_023

This rule checks for a blank line above the **end process** keyword.

Violation

```
  wr_en <= '1';
end process PROC_A;
```

Fix

```
  wr_en <= '1';

end process PROC_A;
```

11.23.24 process_024

This rule checks for a single space after the process label.

Violation

```
PROC_A: process (rd_en, wr_en, data_in, data_out,
                 rd_full, wr_full
                ) is
begin
```

Fix

```
PROC_A : process (rd_en, wr_en, data_in, data_out,
                  rd_full, wr_full
                 ) is
begin
```

11.23.25 process_025

This rule checks for a single space after the : and before the **process** keyword.

Violation

```
PROC_A :process (rd_en, wr_en, data_in, data_out,
                 rd_full, wr_full
                ) is begin
```

Fix

```
PROC_A : process (rd_en, wr_en, data_in, data_out,
                  rd_full, wr_full
                 ) is
begin
```

11.23.26 process_026

This rule checks for blank lines between the end of the sensitivity list and process declarative lines.

Violation

```
PROC_A : process (rd_en, wr_en, data_in, data_out,
                  rd_full, wr_full
                 ) is
  -- Keep track of the number of words in the FIFO
  variable word_count : integer;
begin
```

Fix

```
PROC_A : process (rd_en, wr_en, data_in, data_out,
                  rd_full, wr_full
                 ) is

  -- Keep track of the number of words in the FIFO
```

(continues on next page)

(continued from previous page)

```
  variable word_count : integer;
begin
```

11.23.27 process_027

This rule checks for blank lines between process declarative lines and the **begin** keyword.

Violation

```
PROC_A : process (rd_en, wr_en, data_in, data_out,
                  rd_full, wr_full
                 ) is

  -- Keep track of the number of words in the FIFO
  variable word_count : integer;
begin
```

Fix

```
PROC_A : process (rd_en, wr_en, data_in, data_out,
                  rd_full, wr_full
                 ) is

  -- Keep track of the number of words in the FIFO
  variable word_count : integer;

begin
```

11.23.28 process_028

This rule checks the alignment of the closing parenthesis of a sensitivity list. Parenthesis on multiple lines should be in the same column.

Violation

```
PROC_A : process (rd_en, wr_en, data_in, data_out,
                  rd_full, wr_full
                    )
```

Fix

```
PROC_A : process (rd_en, wr_en, data_in, data_out,
                  rd_full, wr_full
                 )
```

11.23.29 process_029

This rule checks for **rising_edge** and **falling_edge** in processes.

Violation

```
if (rising_edge(CLK)) then

if (falling_edge(CLK)) then
```

Fix

```
if (CLK'event and CLK = '1') then

if (CLK'event and CLK = '0') then
```

11.23.30 process_030

This rule checks for a single signal per line in a sensitivity list that is not the last one. The sensitivity list is required by the compiler, but provides no useful information to the reader. Therefore, the vertical spacing of the sensitivity list should be minimized. This will help with code readability.

Note: This rule is left to the user to fix.

Violation

```
PROC_A : process (rd_en,
                  wr_en,
                  data_in,
                  data_out,
                  rd_full,
                  wr_full
                 )
```

Fix

```
PROC_A : process (rd_en, wr_en, data_in, data_out,
                  rd_full, wr_full
                 )
```

11.23.31 process_031

This rule checks for alignment of identifiers and colons of constant, variable, and file.

Violation

```
PROC_1 : process(A) is

 variable     var1 : boolean;
 constant  cons1 : integer;
 file           file1 : load_file_file open read_mode is load_file_name;

begin

end process PROC_1;
```

Fix

```
PROC_1 : process(A) is

 variable var1  : boolean;
 constant cons1 : integer;
 file     file1 : load_file_file open read_mode is load_file_name;

begin

end process PROC_1;
```

11.23.32 process_032

This rule checks the process label is on the same line as the process keyword.

Violation

```
PROC_1 :

process(A) is
```

Fix

```
PROC_1 : process(A) is
```

11.24 Range Rules

These rules cover the range definitions in signals, constants, ports and other cases where ranges are defined.

11.24.1 range_001

This rule checks the case of the **downto** keyword.

Note: The default is lowercase.

Refer to the section Configuring Uppercase and Lowercase Rules for information on changing the default case.

Violation

```
signal SIG1 : std_logic_vector(3 DOWNTO 0);
signal SIG2 : std_logic_vector(16 downTO 1);
```

Fix

```
signal SIG1 : std_logic_vector(3 downto 0);
signal SIG2 : std_logic_vector(16 downTO 1);
```

11.24.2 range_002

This rule checks the case of the **to** keyword.

Note: The default is lowercase.

Refer to the section Configuring Uppercase and Lowercase Rules for information on changing the default case.

Violation

```
signal SIG1 : std_logic_vector(3 TO 0);
signal SIG2 : std_logic_vector(16 tO 1);
```

Fix

```
signal SIG1 : std_logic_vector(3 to 0);
signal SIG2 : std_logic_vector(16 to 1);
```

11.25 Sequential Rules

11.25.1 sequential_001

This rule checks the indent of sequential statements.

Violation

```
begin

    wr_en <= '1';
rd_en <= '0';
```

Fix

```
begin

  wr_en <= '1';
  rd_en <= '0';
```

11.25.2 sequential_002

This rule checks for a single space after the **<=** operator.

Violation

```
wr_en <=     '1';
rd_en <='0';
```

Fix

```
wr_en <= '1';
rd_en <= '0';
```

11.25.3 sequential_003

This rule checks for at least a single space before the **<=** operator.

Violation

```
wr_en<= '1';
rd_en   <= '0';
```

Fix

```
wr_en <= '1';
rd_en   <= '0';
```

11.25.4 sequential_004

This rule checks the alignment of multiline sequential statements.

Violation

```
overflow <= wr_en and
  rd_en;
```

Fix

```
overflow <= wr_en and
            rd_en;
```

11.25.5 sequential_005

This rule checks the alignment of the **<=** operators over consecutive sequential lines.

Violation

```
wr_en <= '1';
rd_en   <= '0';
```

Fix

```
wr_en   <= '1';
rd_en   <= '0';
```

11.25.6 sequential_006

This rule checks for comments within multiline sequential statements.

Violation

```
overflow <= wr_en and
 --         rd_address(0)
            rd_en;
```

Fix

```
overflow <= wr_en and
            rd_en;
```

11.26 Signal Rules

11.26.1 signal_001

This rule checks the indent of signal declarations.

Violation

```
architecture RTL of FIFO is

signal wr_en : std_logic;
     signal rd_en : std_logic;

begin
```

Fix

```
architecture RTL of FIFO is

  signal wr_en : std_logic;
  signal rd_en : std_logic;

begin
```

11.26.2 signal_002

This rule checks the **signal** keyword has proper case.

Note: The default is lowercase.

Refer to the section Configuring Uppercase and Lowercase Rules for information on changing the default case.

Violation

```
SIGNAL wr_en : std_logic;
```

Fix

```
signal wr_en : std_logic;
```

11.26.3 signal_003

This rule checks for spaces after the **signal** keyword.

Violation

```
signal     wr_en : std_logic;
```

Fix

```
signal wr_en : std_logic;
```

Note: The number of spaces after the **signal** keyword is configurable. Use the following YAML file example to change the default number of spaces.

```
```

rule:

signal_003: spaces: 3

11.26.4 signal_004

This rule checks the signal name has proper case.

Note: The default is lowercase.

Violation

```
signal WR_EN : std_logic;
```

Fix

```
signal wr_en : std_logic;
```

11.26.5 signal_005

This rule checks for a single space after the colon.

Violation

```
signal wr_en :    std_logic;
signal rd_en :std_logic;
```

Fix

```
signal wr_en : std_logic;
signal rd_en : std_logic;
```

11.26.6 signal_006

This rule checks for at least a single space before the colon.

Violation

```
signal wr_en: std_logic;
signal rd_en   : std_logic;
```

Fix

```
signal wr_en : std_logic;
signal rd_en   : std_logic;
```

11.26.7 signal_007

This rule checks for default assignments in signal declarations.

Violation

```
signal wr_en : std_logic := '0';
```

Fix

```
signal wr_en : std_logic;
```

11.26.8 signal_008

This rule checks for valid prefixes on signal identifiers.

Note: Default signal prefix is "s_".

Refer to the section Configuring Prefix and Suffix Rules for information on changing the allowed prefixes.

Violation

```
signal wr_en : std_logic;
signal rd_en : std_logic;
```

Fix

```
signal s_wr_en : std_logic;
signal s_rd_en : std_logic;
```

11.26.9 signal_009

This rule has be renumbered signal_013.

11.26.10 signal_010

This rule checks the signal type has proper case if it is a VHDL keyword.

Note: This rule is disabled by default. The default is lowercase.

Refer to the section Configuring Uppercase and Lowercase Rules for information on changing the default case.

Violation

```
signal wr_en   : STD_LOGIC;
signal rd_en   : Std_logic;
signal cs_f    : t_User_Defined_Type;
```

Fix

```
signal wr_en    : std_logic;
signal rd_en    : std_logic;
signal cs_f     : t_User_Defined_Type;
```

11.26.11 signal_011

This rule checks the signal type is lowercase.

Violation

```
signal wr_en    : STD_LOGIC;
signal rd_en    : Std_logic;
signal cs_f     : t_User_Defined_Type;
```

Fix

```
signal wr_en    : std_logic;
signal rd_en    : std_logic;
signal cs_f     : t_user_defined_type;
```

11.26.12 signal_012

This rule checks multiple signal assignments on a single line are column aligned.

Note: The :'s will be aligned with rule *signal_009*. This rule will only cover two signals on a single line.

Violation

```
signal wr_en, wr_en_f             : std_logic;
signal rd_en_f, rd_en             : std_logic;
signal chip_select, chip_select_f : t_user_defined_type;
```

Fix

```
signal wr_en,       wr_en_f       : std_logic;
signal rd_en_f,     rd_en         : std_logic;
signal chip_select, chip_select_f : t_user_defined_type;
```

11.26.13 signal_013

This rule checks the colons are aligned for all signals in the architecture declarative region.

Violation

```
signal wr_en : std_logic;
signal rd_en   : std_logic;
```

Fix

```
signal wr_en   : std_logic;
signal rd_en   : std_logic;
```

11.26.14 signal_014

This rule checks for consistent capitalization of signal names.

Violation

```
architecture RTL of ENTITY1 is

  signal sig1 : std_logic;
  signal sig2 : std_logic;

begin

  PROC_NAME : process (siG2) is
  begin

    siG1 <= '0';

    if (SIG2 = '0') then
      sIg1 <= '1';
    elisif (SiG2 = '1') then
      SIg1 <= '0';
    end if;

  end process PROC_NAME;

end architecture RTL;
```

Fix

```
architecture RTL of ENTITY1 is

  signal sig1 : std_logic;
  signal sig2 : std_logic;

  PROC_NAME : process (sig2) is
  begin

    sig1 <= '0';

    if (sig2 = '0') then
      sig1 <= '1';
    elisif (sig2 = '1') then
      sig1 <= '0';
    end if;

  end process PROC_NAME;

end architecture RTL;
```

signal_015

This rule checks for multiple signal names defined in a single signal declaration.

Note: By default, this rule will only flag more than two signal declarations. Refer to the section Configuring Number of Signals in Signal Declaration for information on changing the default.

Violation

```
signal sig1, sig2
  sig3, sig4,
  sig5
  : std_logic;
```

Fix

```
signal sig1 : std_logic;
signal sig2 : std_logic;
signal sig3 : std_logic;
signal sig4 : std_logic;
signal sig5 : std_logic;
```

signal_016

This rule checks the signal declaration is on a single line.

Violation

```
signal sig1
  : std_logic;

signal sig2 :
  std_logic;
```

Fix

```
signal sig1 : std_logic;

signal sig2 : std_logic;
```

11.27 Source File Rules

11.27.1 source_file_001

This rule checks for the existance of the source file passed to VSG.

Violation

Source file passed to VSG does not exist. This violation will be reported at the command line in the normal output. It will also be reported in the junit file if the –junit option is used.

Fix

Pass correct file name to VSG.

11.28 Subtype Rules

11.28.1 subtype_001

This rule checks for indentation of the subtype keyword. Proper indentation enhances comprehension.

The indent amount can be controlled by the **indentSize** attribute on the rule. **indentSize** defaults to 2.

Violation

```
architecture RTL of FIFO is

     subtype read_size is range 0 to 9;
subtype write_size is range 0 to 9;

begin
```

Fix

```
architecture RTL of FIFO is

  subtype read_size is range 0 to 9;
  subtype write_size is range 0 to 9;

begin
```

11.28.2 subtype_002

This rule checks for consistent capitalization of subtype names.

Violation

```
subtype read_size is range 0 to 9;
subtype write_size is range 0 to 9;

signal read  : READ_SIZE;
signal write : write_size;

constant read_sz  : read_size := 8;
constant write_sz : WRITE_size := 1;
```

Fix

```
subtype read_size is range 0 to 9;
subtype write_size is range 0 to 9;

signal read  : read_size;
signal write : write_size;

constant read_sz  : read_size := 8;
constant write_sz : write_size := 1;
```

11.28.3 subtype_003

This rule checks for spaces after the **subtype** keyword.

Violation

```
subtype   state_machine is (IDLE, WRITE, READ, DONE);
```

Fix

```
subtype state_machine is (IDLE, WRITE, READ, DONE);
```

Note: The number of spaces after the **subtype** keyword is configurable. Use the following YAML file example to change the default number of spaces.

```

```

rule:

    subtype_003: spaces: 3

11.28.4 subtype_004

This rule checks for valid prefixes in user defined subtype identifiers.

Note: The default new subtype prefix is "st_".

Refer to the section Configuring Prefix and Suffix Rules for information on changing the allowed prefixes.

Violation

```
subtype my_subtype is range 0 to 9;
```

Fix

```
subtype st_my_subtype is range 0 to 9;
```

11.29 Type Rules

11.29.1 type_001

This rule checks the indent of the **type** declaration.

Violation

```
architecture RTL of FIFO is

    type state_machine is (IDLE, WRITE, READ, DONE);

begin
```

Fix

```
architecture RTL of FIFO is

  type state_machine is (IDLE, WRITE, READ, DONE);

begin
```

11.29.2 type_002

This rule checks the **type** keyword has proper case.

Note: The default is lowercase.

Refer to the section Configuring Uppercase and Lowercase Rules for information on changing the default case.

Violation

```
TYPE state_machine is (IDLE, WRITE, READ, DONE);
```

Fix

```
type state_machine is (IDLE, WRITE, READ, DONE);
```

11.29.3 type_003

This rule checks for spaces after the **type** keyword.

Violation

```
type   state_machine is (IDLE, WRITE, READ, DONE);
```

Fix

```
type state_machine is (IDLE, WRITE, READ, DONE);
```

Note: The number of spaces after the **signal** keyword is configurable. Use the following YAML file example to change the default number of spaces.

```

```

rule:

    type_003: spaces: 3

11.29.4 type_004

This rule checks the type name has proper case.

Note: The default is lowercase.

Refer to the section Configuring Uppercase and Lowercase Rules for information on changing the default case.

Violation

```
type STATE_MACHINE is (IDLE, WRITE, READ, DONE);
```

Fix

```
type state_machine is (IDLE, WRITE, READ, DONE);
```

11.29.5 type_005

This rule checks the indent of multiline enumerated types.

Violation

```
type state_machine is (
IDLE,
  WRITE,
READ,
   DONE);
```

Fix

```
type state_machine is (
  IDLE,
  WRITE,
  READ,
  DONE);
```

11.29.6 type_006

This rule checks for a single space before the **is** keyword.

Violation

```
type state_machine    is (IDLE, WRITE, READ, DONE);
```

Fix

```
type state_machine is (IDLE, WRITE, READ, DONE);
```

11.29.7 type_007

This rule checks for a single space after the **is** keyword.

Violation

```
type state_machine is     (IDLE, WRITE, READ, DONE);
```

Fix

```
type state_machine is (IDLE, WRITE, READ, DONE);
```

11.29.8 type_008

This rule checks the closing parenthesis of multiline enumerated types is on it's own line.

Violation

```
type state_machine is (
  IDLE,
  WRITE,
  READ,
  DONE);
```

Fix

```
type state_machine is (
  IDLE,
  WRITE,
  READ,
  DONE
);
```

11.29.9 type_009

This rule checks for an enumerate type after the open parenthesis on multiline enumerated types.

Violation

```
type state_machine is (IDLE,
  WRITE,
  READ,
  DONE
);
```

Fix

```
type state_machine is (
  IDLE,
  WRITE,
  READ,
  DONE
);
```

11.29.10 type_010

This rule checks for a blank line above the **type** declaration.

Violation

```
signal wr_en : std_logic;
type state_machine is (IDLE, WRITE, READ, DONE);
```

Fix

```
signal wr_en : std_logic;

type state_machine is (IDLE, WRITE, READ, DONE);
```

11.29.11 type_011

This rule checks for a blank line below the **type** declaration.

Violation

```
type state_machine is (IDLE, WRITE, READ, DONE);
signal sm : state_machine;
```

Fix

```
type state_machine is (IDLE, WRITE, READ, DONE);

signal sm : state_machine;
```

11.29.12 type_012

This rule checks the indent of record elements in record types.

Violation

```
type interface is record
  data : std_logic_vector(31 downto 0);
chip_select : std_logic;
    wr_en : std_logic;
end record;
```

Fix

```
type interface is record
  data : std_logic_vector(31 downto 0);
  chip_select : std_logic;
  wr_en : std_logic;
end record;
```

11.29.13 type_013

This rule checks the **is** keyword in type definitions has proper case.

Note: The default is lowercase.

Refer to the section Configuring Uppercase and Lowercase Rules for information on changing the default case.

Violation

```
type interface IS record
type interface Is record
type interface is record
```

Fix

```
type interface is record
type interface is record
type interface is record
```

11.29.14 type_014

This rule checks for consistent capitalization of type names.

Violation

```
type state_machine is (IDLE, WRITE, READ, DONE);

signal sm : State_Machine;
```

Fix

```
type state_machine is (IDLE, WRITE, READ, DONE);

signal sm : state_machine;
```

11.29.15 type_015

This rule checks for valid prefixes in user defined type identifiers.

Note: The default new type prefix is "t_".

Refer to the section Configuring Prefix and Suffix Rules for information on changing the allowed prefixes.

Violation

```
type my_type is range -5 to 5 ;
```

Fix

```
type t_my_type is range -5 to 5 ;
```

11.30 Variable Rules

11.30.1 variable_001

This rule checks the indent of variable declarations.

Violation

```
PROC : process () is

variable count : integer;
      variable counter : integer;

begin
```

Fix

```
PROC : process () is

  variable count : integer;
```

(continues on next page)

(continued from previous page)

```
  variable counter : integer;

begin
```

11.30.2 variable_002

This rule checks the **variable** keyword has proper case.

Note: The default is lowercase.

Refer to the section Configuring Uppercase and Lowercase Rules for information on changing the default case.

Violation

```
VARIABLE count : integer;
```

Fix

```
variable count : integer;
```

11.30.3 variable_003

This rule checks for a single space after the **variable** keyword.

Violation

```
variable     count : integer;
```

Fix

```
variable count : integer;
```

11.30.4 variable_004

This rule checks the variable name has proper case.

Note: The default is lowercase.

Refer to the section Configuring Uppercase and Lowercase Rules for information on changing the default case.

Violation

```
variable COUNT : integer;
```

Fix

```
variable count : integer;
```

11.30.5 variable_005

This rule checks there is a single space after the colon.

Violation

```
variable count   :integer;
variable counter :     integer;
```

Fix

```
variable count   : integer;
variable counter : integer;
```

11.30.6 variable_006

This rule checks for at least a single space before the colon.

Violation

```
variable count: integer;
variable counter : integer;
```

Fix

```
variable count : integer;
variable counter : integer;
```

11.30.7 variable_007

This rule checks for default assignments in variable declarations.

Violation

```
variable count : integer := 32;
```

Fix

```
variable count : integer;
```

11.30.8 variable_009

This rule checks the alignment of colons over multiple lines in the architecture declarative region.

Violation

```
architecture ARCH of ENTITY1 is

  variable count : integer;
  variable counter : integer;

begin
```

Fix

```
architecture ARCH of ENTITY1 is

  variable count   : integer;
  variable counter : integer;

begin
```

11.30.9 variable_010

This rule checks the variable type has proper case.

Note: The default is lowercase.

Violation

```
variable count : INTEGER;
```

Fix

```
variable count : integer;
```

11.30.10 variable_011

This rule checks for consistent capitalization of variable names.

Violation

```
architecture RTL of ENTITY1 is

  shared variable var1 : std_logic;
  shared variable var2 : std_logic;

begin

  PROC_NAME : process () is

    variable var3 : std_logic;
    variable var4 : std_logic;

  begin

    Var1 <= '0';

    if (VAR2 = '0') then
      vaR3 <= '1';
    elisif (var2 = '1') then
      VAR4 <= '0';
    end if;

  end process PROC_NAME;

end architecture RTL;
```

Fix

```
PROC_NAME : process () is

  variable var1 : std_logic;
  variable var2 : std_logic;
  variable var3 : std_logic;
  variable var4 : std_logic;

begin

  var1 <= '0';

  if (var2 = '0') then
    var3 <= '1';
  elisif (var2 = '1') then
    var4 <= '0';
  end if;

end process PROC_NAME;
```

11.30.11 variable_012

This rule checks for valid prefixes on variable identifiers.

Note: The default variable prefix is "v_".

Refer to the section Configuring Prefix and Suffix Rules for information on changing the allowed prefixes.

Violation

```
variable my_var : natural;
```

Fix

```
variable v_my_var : natural;
```

11.31 Variable Assignment Rules

11.31.1 variable_assignment_001

This rule checks the indent of a variable assignment.

Violation

```
PROC : process () is
begin

    counter := 0;
count := counter + 1;
```

Fix

```
PROC : process () is
begin

  counter := 0;
  count   := counter + 1;
```

11.31.2 variable_assignment_002

This rule checks for a single space after the assignment.

Violation

```
counter :=0;
count   :=     counter + 1;
```

Fix

```
counter := 0;
count   := counter + 1;
```

11.31.3 variable_assignment_003

This rule checks for at least a single space before the assignment.

Violation

```
counter:= 0;
count := counter + 1;
```

Fix

```
counter := 0;
count := counter + 1;
```

11.31.4 variable_assignment_004

This rule checks the alignment of multiline variable assignments.

Violation

```
counter := 1 + 4 + 10 + 25 +
     30 + 35;
```

Fix

```
counter := 1 + 4 + 10 + 25 +
           30 + 35;
```

11.31.5 variable_assignment_005

This rule checks the alignment of **:=** operators over multiple lines.

Violation

```
counter := 0;
count := counter + 1;
```

Fix

```
counter := 0;
count   := counter + 1;
```

11.31.6 variable_assignment_006

This rule checks for comments in multiline variable assignments.

Violation

```
counter := 1 + 4 + 10 + 25 +
           -- Add in more stuff
           30 + 35;
```

Fix

```
counter := 1 + 4 + 10 + 25 +
           30 + 35;
```

11.32 While Loop Rules

11.32.1 while_loop_001

This rule checks for indentation of the **while** keyword. Proper indentation enhances comprehension.

Violation

```
begin

while (temp /= 0) loop
    temp := temp/2;
  end loop;
```

Fix

```
begin

  while (temp /= 0) loop
    temp := temp/2;
  end loop;
```

11.32.2 while_loop_002

This rule checks for indentation of the **end loop** keyword. The **end loop** must line up with the **while** keyword. Proper indentation enhances comprehension.

Violation

```
begin

  while (temp /= 0) loop
    temp := temp/2;
      end loop;
```

Fix

```
begin

  while (temp /= 0) loop
    temp := temp/2;
  end loop;
```

11.33 Whitespace Rules

11.33.1 whitespace_001

This rule checks for spaces at the end of lines.

Violation

```
entity FIFO is
```

Fix

```
entity FIFO is
```

11.33.2 whitespace_002

This rule checks for tabs.

Violation

```
port (
    WR_EN : in    std_logic;
```

Fix

```
port (
  WR_EN : in    std_logic;
```

11.33.3 whitespace_003

This rule checks for spaces before semicolons.

Violation

```
WR_EN : in    std_logic      ;
```

Fix

```
WR_EN : in    std_logic;
```

11.33.4 whitespace_004

This rule checks for spaces before commas.

Violation

```
WR_EN => wr_en    ,
RD_EN => rd_en,
```

Fix

```
WR_EN => wr_en,
RD_EN => rd_en,
```

11.33.5 whitespace_005

This rule checks for spaces after an open parenthesis.

Note: Spaces before numbers are allowed.

Violation

```
signal data        : std_logic_vector(31 downto 0);
signal byte_enable : std_logic_vector( 3 downto 0);
signal width       : std_logic_vector(  G_WIDTH - 1 downto 0);
```

Fix

```
signal data        : std_logic_vector(31 downto 0);
signal byte_enable : std_logic_vector( 3 downto 0);
signal width       : std_logic_vector(G_WIDTH - 1 downto 0);
```

11.33.6 whitespace_006

This rule checks for spaces before a close parenthesis.

Violation

```
signal data        : std_logic_vector(31 downto 0    );
signal byte_enable : std_logic_vector( 3 downto 0 );
signal width       : std_logic_vector(G_WIDTH - 1 downto 0);
```

Fix

```
signal data        : std_logic_vector(31 downto 0);
signal byte_enable : std_logic_vector( 3 downto 0);
signal width       : std_logic_vector(G_WIDTH - 1 downto 0);
```

11.33.7 whitespace_007

This rule checks for spaces after a comma.

Violation

```
PROC : process (wr_en,rd_en,overflow) is
```

Fix

```
PROC : process (wr_en, rd_en, overflow) is
```

11.33.8 whitespace_008

This rule checks for spaces after the **std_logic_vector** keyword.

Violation

```
signal data    : std_logic_vector (7 downto 0);
signal counter : std_logic_vector    (7 downto 0);
```

Fix

```
signal data    : std_logic_vector(7 downto 0);
signal counter : std_logic_vector(7 downto 0);
```

11.33.9 whitespace_010

This rule checks for spaces before and after the concate (&) operator.

Violation

```
a <= b&c;
```

Fix

```
a <= b & c;
```

11.33.10 whitespace_011

This rule checks for spaces before and after math operators +, -, /, and *.

Violation

```
a <= b+c;
a <= b-c;
a <= b/c;
a <= b*c;
a <= b**c;
a <= (b+c)-(d-e);
```

Fix

```
a <= b + c;
a <= b - c;
a <= b / c;
a <= b * c;
a <= b ** c;
a <= (b + c) - (d - e);
```

11.33.11 whitespace_012

This rule enforces a maximum number of consecutive blank lines.

Violation

a <= b;

c <= d;

Fix

a <= b;

c <= d;

Note: The default is set to 1. This can be changed by setting the *numBlankLines* attribute to another number.

```
{
    "rule":{
        "whitespace_012":{
            "numBlankLines":3
        }
    }
}
```

11.33.12 whitespace_013

This rule checks for spaces before and after logical operators.

Violation

if (a = ‘1’)and(b = ‘0’) if (a = ‘0’)or (b = ‘1’)

Fix

if (a = ‘1’) and (b = ‘0’) if (a = ‘0’) or (b = ‘1’)

11.34 Wait Rules

11.34.1 wait_001

This rule checks for indentation of the **wait** keyword. Proper indentation enhances comprehension.

Violation

```
begin

    wait for 10ns;
 wait on a,b;
       wait until a = '0';
```

Fix

```
begin

  wait for 10ns;
  wait on a,b;
  wait until a = '0';
```

11.35 When Rules

These rules cover the usage of **when** keywords in sequential and concurrent statements.

11.35.1 when_001

This rule checks the **else** keyword is not at the beginning of a line. The else should be at the end of the preceeding line.

Violation

```
wr_en <= '1' when a = '1' -- This is comment
         else '0' when b = '0'
         else c when d = '1'
         else f;
```

Fix

```
wr_en <= '1' when a = '1' else -- This is a comment
         '0' when b = '0' else
         c when d = '1' else
         f;
```

11.36 With Rules

11.36.1 with_001

This rule checks for **with** statements.

Violation

```
with buttons select
```

Fix

Refactor **with** statement into a process.

CHAPTER 12

API

VSG was written to be included in other tools. The command line script provides one means of using VSG. It also provides an example of how to use the API.

There are two main modules you will use when incorporating VSG into another program:

12.1 vsg.vhdlFile

This is one of two classes you will use when incorporating vsg into another python program.

class vsg.vhdlFile.**vhdlFile**(*filecontent*)

Holds contents of a VHDL file. When a vhdlFile object is created, the contents of the file must be passed to it. A line object is created for each line read in. Then the line object attributes are updated.

Parameters:

filecontent: (list)

Returns:

fileobject

12.2 vsg.rule_list

This is one of two classes you will use when incorporating vsg into another python program.

class vsg.rule_list.**rule_list**(*oVhdlFile*, *sLocalRulesDirectory=None*)

Contains a list of all rules to be checked. It loads all base rules. Localized rules are loaded if specified.

Parameters:

oVhdlFile: (vhdlFile object)

sLocalRulesDirectory: (string) (optional)

check_rules()
Analyzes all rules in increasing phase order. If there is a violation in a phase, analysis is halted.

Parameters: None

configure(*configurationFile*)
Configures individual rules based on dictionary passed.

Parameters:

configurationFile: (dictionary)

extract_junit_testcase(*sVhdlFileName*)
Creates JUnit XML file listing all violations found.

Parameters:

sVhdlFileName (string)

Returns: (junit testcase object)

fix(*iPhase*)
Applies fixes to all violations found.

Parameters:

iPhase: (integer)

get_configuration()
Returns a dictionary with every rule and how it is configured.

Parameters:

None

Returns: (dictionary)

report_violations(*sOutputFormat*)
Prints out violations to stdout.

Parameters:

sOutputFormat (string)

Use the following modules when writing rules:

12.3 vsg.check

12.4 vsg.fix

This module contains functions for rules to fix issues.

vsg.fix.**enforce_one_space_after_word**(*self*, *oLine*, *sWord*)
Adds a space after a word.

Parameters:

self: (rule object)

oLine: (line object)

sWord: (string)

vsg.fix.**enforce_one_space_before_word**(*self*, *oLine*, *sWord*, *fWholeWord=False*)
Adds a space before word.

Parameters:

self: (rule object)

oLine: (line object)

sWord: (string)

vsg.fix.**enforce_spaces_after_word**(*self*, *oLine*, *sWord*, *iSpaces*)
Adds a space after a word.

Parameters:

self: (rule object)

oLine: (line object)

sWord: (string)

iSpaces: (integer)

vsg.fix.**identifier_alignment**(*self*, *oFile*)
Aligns identifiers and colons across multiple lines.

Parameters:

self: (rule object)

oFile: (vhdlFile object)

vsg.fix.**indent**(*self*, *oLine*)
Fixes indent violations.

Parameters:

self: (rule object)

oLine: (line object)

vsg.fix.**insert_blank_line_above**(*self*, *oFile*, *iLineNumber*)
This function inserts a blank line above the line specified by iLineNumber.

Parameters:

self: (rule object)

oFile: (vhdlFile object)

iLineNumber: (integer)

vsg.fix.**insert_blank_line_below**(*self*, *oFile*, *iLineNumber*)
This function inserts a blank line below the line specified by iLineNumber.

Parameters:

self: (rule object)

oFile: (vhdlFile object)

iLineNumber: (integer)

vsg.fix.**keyword_alignment**(*self*, *oFile*)
Aligns keywords across multiple lines.

Parameters:

self: (rule object)

oFile: (vhdlFile object)

vsg.fix.**lower_case**(*oLine*, *sKeyword*)
Changes word to lowercase.

Parameters:

self: (rule object)

oLine: (line object)

sKeyword: (string)

vsg.fix.**multiline_alignment**(*self*, *oFile*, *iLineNumber*)
Indents successive lines of multiline statements.

Parameters:

self: (rule object)

oFile: (vhdlFile object)

iLineNumber: (integer)

vsg.fix.**remove_blank_lines_above**(*self*, *oFile*, *iLineNumber*, *sUnless=None*)
This function removes blank lines above a linenumber. If sUnless is specified, a single blank line will be left if a line with the sUnless attribute is encountered.

Parameters:

self: (rule object)

oFile: (vhdlFile object)

iLineNumber: (integer)

sUnless: (string) (optional)

vsg.fix.**remove_blank_lines_below**(*self*, *oFile*, *iLineNumber*, *sUnless=None*)
This function removes blank lines below a linenumber. If sUnless is specified, a single blank line will be left if a line with the sUnless attribute is encountered.

Parameters:

self: (rule object)

oFile: (vhdlFile object)

iLineNumber: (integer)

sUnless: (string) (optional)

vsg.fix.**replace_is_keyword**(*oFile*, *iLineNumber*)
This function removes the is keyword from a line if it starts with is. If the line is empty, it is replaced with a blank line.

Parameters:

oFile: (vhdlFile object)

iLineNumber: (integer)

vsg.fix.**upper_case**(*oLine*, *sKeyword*)
Changes word to lowercase.

Parameters:

self: (rule object)

oLine: (line object)

sKeyword: (string)

vsg.fix.**upper_case_with_parenthesis**(*self*, *oLine*, *sKeyword*)
Changes word to lowercase.

Parameters:

self: (rule object)

oLine: (line object)

sKeyword: (string)

12.5 vsg.utilities

This module provides functions for rules to use.

vsg.utils.**begin_of_line_index**(*oLine*)
Finds the left most non whitespace character. Returns the index of the first non whitespace character.

Parameters:

oLine: (line object)

Returns: (integer)

vsg.utils.**change_word**(*oLine*, *sWord*, *sNewWord*, *iMax=1*)
Changes one word in the line to another.

Parameters:

oLine: (line object)

sWord: (string)

sNewWord: (string)

vsg.utils.**clear_keyword_from_line**(*oLine*, *sKeyword*)
Removes a keyword from a line.

Parameters:

oLine: (line object)

sKeyword: (string)

vsg.utils.**copy_line**(*oFile*, *iLineNumber*)
Creates a copy of the line at iLineNumber and inserts it below iLineNumber.

Parameters:

oFile: (vhdlFile object)

iLineNumber: (integer)

vsg.utils.**end_of_line_index**(*oLine*)
Finds the end of the code on a line ignoring comments. Returns the index of the last code character.

Parameters:

oLine: (line object)

Returns: (integer)

vsg.utils.**extract_class_identifier_list**(*oLine*)
Returns a class identifiers list.

Parameters:

oLine: (line object)

Returns: (list of strings)

vsg.utils.**extract_class_name**(*oLine*)
Returns the name of a type in a type declaration.

Parameters:

oLine: (line object)

Returns: (one element list of strings)

vsg.utils.**extract_component_identifier**(*oLine*)
Returns the entity identifier.

Parameters:

oLine: (line object)

Returns: (one element list of strings)

vsg.utils.**extract_end_label**(*oLine*)
Returns the end label.

Parameters:

oLine: (line object)

Returns: (one element or empty list of strings)

vsg.utils.**extract_entity_identifier**(*oLine*)
Returns the entity identifier.

Parameters:

oLine: (line object)

Returns: (one element list of strings)

vsg.utils.**extract_first_keyword**(*oLine*)
Returns first keyword from line.

Parameters:

oLine: (line object)

Returns: (one element list of strings)

vsg.utils.**extract_generics**(*oLine*)
Returns a generics list.

Parameters:

oLine: (line object)

Returns: (list of strings)

vsg.utils.**extract_label**(*oLine*)
Returns the label.

Parameters:

oLine: (line object)

Returns: (one element list of strings)

vsg.utils.**extract_non_keywords**(*sString*)

Returns a keyword list with the following removed: :'s commas semicolons vhdl keywords double quotes numbers ticks comments

Parameters:

sString: (string)

Returns: (list of strings)

vsg.utils.**extract_port_name**(*oLine*)
Returns port name from line.

Parameters:

oLine: (line object)

Returns: (one element list of strings)

vsg.utils.**extract_port_names_from_port_map**(*oLine*)
Returns port names from port assignment inside port map. Parameters:

oLine: (line object)

Returns: (list of strings)

vsg.utils.**extract_type_identifier**(*oLine*)
Returns the type identifier from type declaration.

Parameters:

oLine: (line object)

Returns: (one element list of strings)

vsg.utils.**extract_type_name**(*oLine*)
Returns the name of a type in various declarations.

Parameters:

oLine: (line object)

Returns: (zero or one element list of strings)

vsg.utils.**extract_type_name_from_port**(*oLine*)
Returns the name of a type in port declaration.

Parameters:

oLine: (line object)

Returns: (one element list of strings)

vsg.utils.**extract_type_name_from_port_vhdl_only**(*oLine*)
Returns the name of a VHDL only types in port declaration.

Parameters:

oLine: (line object)

Returns: (one element list of strings)

vsg.utils.**extract_type_name_vhdl_only**(*oLine*)
Returns the name of a VHDL only types in various declarations.

Parameters:

oLine: (line object)

Returns: (one element or empty list of strings)

vsg.utils.**extract_words**(*oLine*, *words*)
Returns words from line. Case insensitive, however returned words preserve their case.

Parameters:

oLine: (line object)

words: (list of words to extract)

Returns: (list of strings)

vsg.utils.**get_first_word**(*oLine*)
Returns the first word from a line at iIndex.

Parameters:

oLine: (line object)

Returns: (string)

vsg.utils.**get_word**(*oLine*, *iIndex*)
Returns a word from a line at iIndex.

Parameters:

oLine: (line object)

iIndex: (integer)

Returns: (string)

vsg.utils.**insert_line**(*oFile*, *iIndex*)
Inserts a blank line at iIndex into the file line list.

Parameters:

oFile: (File Object)

iIndex: (integer)

Returns: Nothing

vsg.utils.**is_number**(*sString*)
Returns boolean if the string passed is a number.

Parameters:

sLine: (string)

Returns: boolean

vsg.utils.**is_port_mode**(*sWord*)
Returns True if given word is a valid port mode.

Returns False if given word is not a valid port mode.

Parameters:

sWord: (string)

Returns: (boolean)

vsg.utils.**is_vhdl_keyword**(*sWord*)
Returns True if given word is a VHDL keyword.

Returns False if given word is not a VHDL keyword.

Parameters:

sWord: (string)

Returns: (boolean)

vsg.utils.**reclassify_line**(*oFile*, *iLineNumber*)
Updates the following attributes on the current and next line:

- isFunctionReturn
- insideVariableAssignment
- isVariableAssignmentEnd
- isVariableAssignment
- insideSequential
- isSequentialEnd
- isSequential
- hasComment
- hasInlineComment
- commentColumn

Parameters:

oFile: (vhdlFile object)

iLineNumber: (integer)

vsg.utils.**remove_blank_line**(*oFile*, *iLineNumber*)
Removes a line if it is blank.

Parameters:

oFile: (vhdlFile object)

iLineNumber: (integer)

vsg.utils.**remove_closing_parenthesis_and_semicolon**(*oLine*)
Parameters:

oLine: (line object)

Returns: (line object)

vsg.utils.**remove_comment**(*sString*)
Returns a string without comments.

Parameters:

sString: (string)

Returns: (string)

vsg.utils.**remove_comment_attributes_from_line**(*oLine*)
Sets all comment attributes on a line to indicate no comment is present.

Parameters:

oLine: (line object)

vsg.utils.**remove_line**(*oFile*, *iLineNumber*)
Removes a line from the file line list.

Parameters:

oFile: (File Object)

iLineNumber : (integer)

Returns: Nothing

vsg.utils.**remove_lines**(*oFile*, *iStartLine*, *iEndLine*)
Removes a series of lines from the file line list.

Parameters:

oFile: (File Object)

iStartLine: (integer)

iEndLine: (integer)

Returns: Nothing

vsg.utils.**remove_parenthesis_from_word**(*sWord*)
Removes parenthesis from words:

Hello(there) => Hello Hello => Hello

Parameters:

sWord: (string)

Returns: (string)

vsg.utils.**remove_text_after_word**(*sKeyword*, *sWord*)
Removes all text after a keyword.

Parameters:

sKeyword: (string)

sWord: (string)

vsg.utils.**replace_word_by_index**(*oLine*, *iIndex*, *sWord*)
Replaces text in a line at a given index with a given word.

Parameters:

oLine: (Line Object)

iIndex: (integer)

sWord: (string)

Returns: Nothing

vsg.utils.**search_for_and_remove_keyword**(*oFile*, *iLineNumber*, *sKeyword*)
Searches for a keyword on lines below the current line and removes it if discovered.

Parameters:

oFile: (vhdlFile object)

iLineNumber: (integer)

sKeyword: (string)

vsg.utils.**split_line_after_word**(*oFile*, *iLineNumber*, *sWord*)
Splits the line after the word given and inserts it after the current line.

Parameters:

oFile: (vhdlFile object)

iLineNumber: (integer)

sWord: (string)

vsg.utils.**split_line_before_word**(*oFile*, *iLineNumber*, *sWord*)
Splits the line before the word given and inserts it after the current line.

Parameters:

oFile: (vhdlFile object)

iLineNumber: (integer)

sWord: (string)

vsg.utils.**strip_semicolon_from_word**(*sWord*)
Removes trailing semicolon from a word:

case; => case entity; => entity

Parameters:

sWord: (string)

Returns: (string)

vsg.utils.**update_comment_line_attributes**(*oLine*)
Updates the following attributes on a line:

self.isComment self.hasComment self.hasInLineComment self.commentColumn

Parameters:

oLine: (Line Object)

Returns: Nothing

CHAPTER 13

Contributing

I welcome any contributions to this project. No matter how small or large.

There are several ways to contribute:

1. Bug reports
2. Code base improvements
3. Feature requests
4. Pull requests

13.1 Bug Reports

I used code from open cores to develop VSG. It provided many different coding styles to process. There are bound to be some corner cases or incorrect assumptions in the code. If you run into anything that is not handled correctly, please submit an issue. When creating the issue, use the **bug** label to highlight it. Fixing bugs is prioritized over feature enhancements.

13.2 Code Base Improvements

VSG started out to solve a problem and learn how to code in Python. The learning part is still on going, and I am sure the code base could be improved. I run the code through *Codacy* and *Code Climate*, and they are very helpful. However, I would appreciate any suggestions to improve the code base.

Create an issue and use the **refactor** label for any code which could be improved.

13.3 Feature Requests

Let me know if there is anything I could add to VSG easier to use. The following features were not in my original concept of VSG.

- fix
- fix_phase
- output_format
- backup

Fix is probably the most important feature of VSG. I added it when someone said it would be nice if VSG just fixed the problems it found. There may be other important features, I just have not thought of them yet.

If you have an idea for a new feature, create an issue with the **enhancement** label.

13.4 Pull Requests

Pull requests are always welcome. I am trying to follow a Test Driven Development (TDD) process. Currently there are over 1000 tests. If you do add a new feature or fix a bug, I would appreciate a new or updated test to go along with the change.

I use *Travis CI* to run all the tests. I also use *Codacy* and *Code Climate* to check for code style issues. I use *Codcov* to check the code coverage of the tests.

Travis CI will run these tools when a pull request is made. The results will be available on the pull request Github page.

13.5 Running Tests

Before submitting a pull request, you can run the existing tests locally. These are the same tests Travis CI will run.

To run the tests issue the following command when using python 2.7:

```
python -m unittest discover
```

To run the tests using python 3 use the following command:

```
python -m unittest
```

After issuing the command the tests will be executed.

```
vhdl-style-guide$ python -m unittest discover
................................................
................................................
................................................
................................................
................................................
................................................
................................................
................................................
................................................
................................................
```

(continues on next page)

(continued from previous page)

```
................................................
................................................
................................................
................................................
................................................
................................................
................................................
................................................
................................................
................................................
................................................
................................................
.......................................
------------------------------------------------
Ran 1170 tests in 6.424s

OK
```

CHAPTER 14

Frequency Asked Questions

14.1 How do I allow my user defined type definitions to not be lower cased?

Rules *type_004* and *signal_011* enforce a lower case consistency for user defined types. Rule *signal_010* enforces lower case signal types for VHDL base types. If you want to allow for any case in user defined types, then we need to:

- Disable type_004
- Disable signal_011
- Enable signal_010

Use the following configuration and pass it to VSG when you analyze your code:

```
{
    "rule":{
        "type_004":{
            "disable":true
        },
        "signal_011":{
            "disable":true
        },
        "signal_010":{
            "disable":false
        }
    }
}
```

14.2 How do I align *signal_003*, *constant_003*, *file_003*, *type_003*, *subtype_003*, and *file_003*?

The default behavior of VSG is to minimize horizontal spacing whenever possible. This would result in the following code formatting:

```
signal wr_en : std_logic;
constant size : integer := 1;
type state_machine is (IDLE, WRITE, READ, DONE);
subtype read_size is range 0 to 9;
file defaultImage : load_file_type open read_mode is load_file_name;
```

If you would rather the code formatting as follows:

```
signal   wr_en : std_logic;
constant size : integer := 1;
type     state_machine is (IDLE, WRITE, READ, DONE);
subtype  read_size is range 0 to 9;
file     defaultImage : load_file_type open read_mode is load_file_name;
```

Then use the following YAML code in a configuration file:

```
rule:
  signal_003:
    spaces: 3
  type_003:
    spaces: 5
  subtype_003:
    spaces: 2
  file_003:
    spaces: 5
```

Python Module Index

v

Index

R

S

U

V