Compliance Checking for OVM Interface OVCs

Sharon Rosenberg (Cadence Design Systems)

September 2009

Introduction

As many companies standardize on the Open Verification Methodology (OVM), they want to answer the question – "how do we ensure all of our projects are consistently applying the methodology and adhering to it's key guidelines". Following standard methodology guidelines results in higher coding efficiency and a consistent user experience for both creating and using verification components.. Methodology consistency and standardization are the best ways to promote reuse, to reduce learning curve for adopting unfamiliar verification IP (VIP), and to ensure new engineers to your team are able to quickly learn how to write good, reusable verification code.. At the same time, multiple vendors are producing commercial VIP, training modules, and other tools around the OVM. How do we enable a single mainstream use-model to ensure consistency in the growing OVM ecosystem?

An OVM Compliance Checklist can achieve all of these goals by clearly defining a set of discrete rules and recommendations for measuring compliance to the methodology.

The Cadence OVM Compliance Checklist

Cadence has contributed an OVM compliance checklist to OVMworld.org website. The checklist was initially released for OVM ***e*** (***e***RM) in 2003 and included a static analysis tool to enforce compliance. The original OVM ***e*** compliance checklist has been used successfully for the past six years by many companies to ensure methodology compliance for both internally developed and commercial VIP. With the more recent introduction of OVM SystemVerilog, the checklist has been enhanced to support a consistent methodology across both ***e*** and SystemVerilog and OVM SystemVerilog users, have already used it successfully for some time. The checklist is derived from the OVM User Guide, which is part of the OVM release on the OVMWorld.org site, and defines the methodology concepts.

Where to Get it?

The OVM compliance checklist can be downloaded at: http://www.ovmworld.org/contributions.php.

Please contact ovm_contributions@cadence.com for questions and suggestions.

The Structure of the OVM Compliance Checklist

While it is possible to view OVM as a set of library features that can be used in multiple ways, OVM is first and foremost a complete proven methodology and a set of concepts to maximize productivity and reuse. As such, some features are critical for concept level consistency and reuse, while other features provide further automation, guidelines or enablers which may not be as critical depending on the level of intended verification component reuse. For example preserving the recommended topology of reusable environments, agents and configuration attributes is a fundamental requirement for consistency. Using the field macro automation for printing is a recommendation – it ensures consistent log files for VIP integrators who use multiple OVCs from different sources as well as saving time for VIP developers. To reflect these differences the checklist introduces both rules and recommendations (tagged as [recommended]). The rules must be followed since they represent guidelines to make a reusable verification component compliant to the OVM. While recommendations are optional, they do ensure optimal reuse and consistency. Some checks are marked as [Statistical] indicating that they are only providing statistical information without specifying the success criteria.

Keep in mind that these requirements were collected and tuned over years by many customers over many projects. If the reasoning for a check is not clear or you have an enhancement suggestion, it is best to follow the checklist and in parallel express your concerns with the OVM development team.

The OVM compliance checklist contains the following categories:

- Packaging and Name Space – guidelines on how to package environments in a consistent way for easy shipping and delivery.
- Architecture – Checks to ensure similar high-level topology for OVM environments. This is critical for understanding a new OVC, its configuration and class hierarchy.
- Reset and Clock – various reset and clock topics
- Checking – touches the self checking aspects of reusable OVC
- Sequences – practices on creating reusable sequence library and correct setup
- Messaging – defines message methodology to allow user to efficiently debug environment and reduce support requirements
- Documentation – captures the requirement for a complete documentation requirements
- General Deliverables – more delivery requirements
- End of tests – minimal end-of-test requirements
- OVM SV specific compliance checks – checks that are specific to OVM SV implementation.

Each check has a unique identification code which is used for documentation, queries for more details, and cross tool consistency.

Automated OVM Compliance Checking

Many customers who have been using the OVM and applying the compliance checklist have requested an automated tool for checking VIP against the checklist. AMIQ, an EDA company that is part of the growing OVM ecosystem, has enhanced a tool called"DVT to automate OVM compliance checking. DVT is an Eclipse-based integrated development environment (IDE) targeted at increasing productivity of OVM SystemVerilog and ***e*** developers.

DVT also provides a broad range of OVM compliance review features based on the OVM Compliance Checklist, including:

- Overview of the environment architecture
- Customizable checks (grouped in categories including architecture, stimuli, checking, coverage, messaging, reset handling, packaging, etc.)
- Statistics (sequence library, checks, coverage groups, etc.)
- Graphical user interface (filters, search, etc.)
- Direct jump to problematic source code
- Integrated review and development (checks are refreshed as you fix errors)
-

After performing the analysis and adjusting the reusable environment accordingly, the VIP developer can export an OVM compliance HTML report to be delivered with the verification IP.

For more information on DVT please refer to http://www.dvteclipse.com

Conclusions and Next Steps

As verification continues to be one of the biggest challenges and largest time consumers in getting a new design to market, verification efficiency is critical. Verification IP reuse is one of best ways to signicantly improve verification efficiency. As more companies standardize on the OVM, the associated OVM Compliance Checklist offers the means to assure smooth VIP exchange and reuse in your project's ecosystem.

OVM Interface OVC Compliance Checks

September 2009

Version 1.0

This document lists OVM compliance checks defined for OVM verification environments. The compliance checklist was requested by corporations and OVM users wishing to ensure consistency, similar user experience, and compliance to the official OVM SystemVerilog User Guide provided with the OVM2.0.2 release and OVM concepts. Static commercial tools such as DVT allow forcing these checks on user environments

Most of the requirements are aligned between ***e*** and SV, except for the last section which is specific to SV.

The checks are divided into several categories and each check has a unique identification code. Some of the checks are marked as [Recommended] to indicate that they are not required. Some are [Statistical] indicating that they only provide statistical information, without specifying the success criteria.

Packaging and Name Space Compliance Checks

Packaging checks are mostly static checks of the code and directory structure. The demo checks require simulation.

PKDF [Recommended]

Valid distribution format—verify that the package is distributed according to OVM recommendations, as a compressed tar file with the name including the package name and the version number: <package>_version_<version>.tar.gz

PKDR

Demo script running—verify that the package contains a *demo.sh* file to demonstrate the package, located at the top level of the package. Verify that the demo is running OK. Check log file results for errors and compare to expected result if available.

PKCL

Loading and running with golden OVC—verify that the verification component can run properly with the golden OVCs (e.g. xbus) loaded. No errors, name conflicts etc.

PKCT

Configuration example—verify that there is a testbench example in the examples directory that shows how to instantiate and exercise the OVC.

PKER

Examples documented—verify that there is an EXAMPLES_README.txt file in the examples directory detailing the contents of the directory. Verify that all the test files are mentioned in the readme file.

PKIM

Include statements—verify that all include/import statements refer to included files using local or package-relative path. No include statements should assume absolute path or path to another package outside the scope of this package.

PKLD

Legal directory name—verify that the package directory is named consistently with the package name. The package top directory name must be identical to the file name prefix used for the source files.

PKVD

Valid directory structure—verify that the package's directory is structured correctly (i.e. contains the example and docs directories and the source directory sv/***e***/sc)

PKTP

Top and package files—verify that the OVC has both a top level source file and a top level package file in the language specific subdirectory (e.g. /sv).

- The top-level package file of the OVC is named <package>_pkg.sv. Check that this file has the package definition which allows using this OVC as a package. Check that this file has only `include and define statements, and it has or includes the version definition. If the source files are structured in several sub-directories, each sub-directory should have a similar header file, prefixed by the name of the sub-directory.

PKFN

Valid file names—verify that all source code filenames in the package start with the package name.

PKGE

Global extends—verify that the package contains no code outside the package scope that may conflict with user code

PKVR

Valid README file—verify that there is a legal PACKAGE_README.txt file at the top level of the package.

PKVN

Version number—verify that the version number in the PACKAGE_README.txt file matches the version number in the top file.

PKSS

Simulator support—verify that the simulators supported by the demo are documented in the PACKAGE_README.txt file

PKSD

Simulator support documented—verify that the simulators supported by the package are documented in the package documentation.

Architecture Compliance Checks

These checks require simulation to instantiate the OVC and check its architecture.

An "agent" is any component derived from ovm_agent or a derivative of it. Active agents are agents with their "is_active" field set to OVM_ACTIVE.

An "env" is any component derived from ovm_env or a derivative of it.

A "monitor" is any component derived from ovm_monitor or a derivative of it.

A "driver" is any component derived from ovm_driver or a derivative of it.

A "sequencer" is any component derived from ovm_sequencer or a derivative of it. Driver sequencers are directly connected to drivers and generate the items for the drivers, while virtual sequencers are not connected to drivers and generate sequences or sequence items to drive other sequencers.

ARAA

Active agents—verify that all ACTIVE agents have a driver in them. Exception: Active agents that contain other agents are not required to have a driver in them.

ARIN

Instantiation—verify that all agents are instantiated in reusable OVC envs (as opposed to testbench classes) or other super-agents.

ARMN

Monitors—verify that all monitors are instantiated in envs or agents. Elements of the monitor can be implemented within the SystemVerilog interface (e.g. events and protocol checks)

ARBF

Driver—verify that all drivers are instantiated only in ACTIVE agents.

ARSA

Sequencers (Sequence drivers) in ACTIVE agents—verify that all sequencers are instantiated in ACTIVE agents. Also verify that all ACTIVE agents have a sequencer. Exception: active agents that contain other agents are not required to have a sequencer in them.

ARBS

Driver driving signals—verify that all DUT signals are driven only by drivers. No DUT signals should be driven by components that are not derived from ovm_driver.

ARMP

The monitors are passive sub-components that should never drive DUT signals

ARIO [recommended]

Interface OVC should be complete and include both the active and reactive components. For example, the same UVC should be able to emulate both master and slave. If due to time pressure some logic is not implemented, it is encouraged to create empty sub-components for the missing logic and document it.

Reset and Clock Compliance Checks

These checks must be based on some input from the developer TBD. If the names of the tests, checks and coverage are declared by the developer, they can provide useful information to the user and facilitate automatic checking.

RSDC

Working with DUT supplied clock—verify that the package works with DUT supplied clock.

RSRS

Reset support—verify that the package correctly responds to resets (of any length) generated within the DUT at the start of the test . The developer must provide a test and assertion demonstrating this.

RSCH

Reset checks—verify that there are sufficient checks to ensure that the DUT behaves correctly after reset is de-asserted. There must be checks verifying correct behavior of the DUT when the first reset is deasserted, as well as checks verifying proper behavior when reset occurs at some arbitrary point in the simulation.

RSRC

Multiple reset checks—verify that there are sufficient specific checks relating to the DUT's response to assertion/deassertion of reset. Name a test that shows multiple resets, and a coverage point which checks for multiple resets.

RSMR [Recommended]

Multiple resets—verify that the package manages multiple resets during the test. The developer should provide a test demonstrating this.

RSPR [Recommended]

Programmable resets—verify that the package provides a mechanism for generating programmable reset(s) and check that this feature can be disabled if necessary.

Checking Compliance Checks

Checking here relates to the various means to verify the correctness of the DUT, including assertions, dut_error, and check and expect (in e).

CHPC

Protocol checking—check that the package provides sufficient DUT checking (for example, protocol checkers) to cover all possible DUT errors.

CHCA [recommended]

Declarative protocol assertions should be placed within the SystemVerilog interfaces. Should be able to control the enabling, disabling for these assertions using the standard coverage and checking configuration attributes.

CHEM [Recommended]

Error messages—verify that all error messages provide sufficient detail for the user to identify the area and instance of the package/DUT that produced the error.

CHED

Error message documentation—check that all checks (both for DUT errors and user errors) are sufficiently documented. Every check must have a name/tag and the documentation must list and describe all of them.

CHCE [Statistical]

Checks and expects—how many checks, expects, assertions are there? Count the number of dut_errors (ovm_error_*) and assertions to provide some indication about the quality of checking provided by the OVC.

CHEX [Recommended]

Named expects—check that all expects and assertions are named.

CHSC [Recommended]

Scoreboard—check that the package provides (where appropriate) toolkits to enable the user to code complex data flow checking tasks. Check that the package provides a scoreboard either already integrated, or as a generic tool.

CVCG [Statistical]

Coverage groups—how many coverage groups are defined?

CVED

Should be able to disable collection of coverage

CVCI [Statistical]

Coverage items—how many coverage items are defined?

CVCR [Statistical]

Coverage crosses—how many coverage crosses are defined?

CVRS [Recommended]

Coverage results—check that the coverage report produced after testing this OVC is included in the package. Preferred approach: Supply the coverage result files and a shell script called "show_cover.sh", all in a directory called “coverage”. Alternative approach: provide an ASCII coverage report.

CVPL

Verification plan—verify that there is vPlan provided with the OVC.

CVVN

Always use verbosity OVM_NONE for warnings, errors and fatal messages. This ensures that bugs are not being overlooked due to command-line verbosity changes (Library default is not always OVM_NONE).

```
ovm_report_warning(get_type_name(),"Item out of range and will be dropped!",OVM_NONE);
```

Sequences Compliance Checks

SQEI

Error injection—verify that the sequence items provide sufficient ability to inject errors into the generated data stream(s)

SQSS [Statistical]

Sequencer statistics—how many sequencers exist? How many of them are driver sequencers and how many virtual sequencers?

SQPT [Statistical]

Predefined sequence types—how big is the sequence library? For each sequencer type, how many "kinds" of sequences are defined in it (other than the 3 default ones)?
The sequence definition requires the usage of the ovm_sequence_utils macro which associates the new sequence type with a sequencer type. The new sequence becomes part of the “library” for this sequencer and can be used by it. The check is intended to count the number of sequences provided in the verification component for each sequencer (on top of the 3 predefined ones).

```
class simple_seq_do extends ovm_sequence #(simple_item);
  rand int count;
  constraint c1 { count >0; count <50; }
  // Constructor
  function new(string name="simple_seq_do");
    super.new(name);
  endfunction
  // OVM automation macros for sequences
  `ovm_sequence_utils(simple_seq_do, simple_sequencer)
  // The body() task is the actual logic of the sequence.
```

```
  virtual task body();
    repeat(count)
    `ovm_do(req)
  endtask : body
endclass : simple_seq_do
```

SQSD [Recommended]

Sub-sequencers—verify that all virtual sequencers have sub-sequencers.

Sub-sequencers are associated with the virtual sequencer by declaring the sub-sequencer fields in the declaration of the virtual sequencer and then performing the assignment in the connect phase of a parent component, which also created the virtual sequencer.

```
class simple_virtual_sequencer extends ovm_sequencer;
  eth_sequencer eth_seqr; // sub-sequencer component
  cpu_sequencer cpu_seqr;
  …
endclass

class simple_tb extends ovm_env;
  cpu_env_c cpu0; // Reuse a cpu verification component.
  eth_env_c eth0; // Reuse an ethernet verification component.
  simple_virtual_sequencer v_sequencer;
  ... // Constructor and OVM automation macros
  virtual function void build();
    super.build();
    …
    eth0 = eth_env_c::type_id::create("eth0", this);
    cpu0 = cpu_env_c::type_id::create("cpu0", this);
    // Build the virtual sequencer.
    v_sequencer = simple_virtual_sequencer::type_id::create("v_sequencer",
                  this);
  endfunction : build
  // Connect virtual sequencer to sub-sequencers.
  function void connect();
    v_sequencer.cpu_seqr = cpu0.master[0].sequencer;
    v_sequencer.eth_seqr = eth0.tx_rx_agent.sequencer;
  endfunction : connect
endclass: simple_tb
```

Messaging Compliance Checks

MSMA [Statistical]

Message actions—how many message/ovm_report_info actions are there? At what verbosity levels are these actions?

MSUA [recommended]

It is recommended to use the field macro automation for class printing (print() and sprint()). This allows environment integrators to quickly read and understand log files of OVCs coming from different resources. If a user decides to manually implement the print() and sprint(), it is recommended to produce a similar output to what the automated print methods introduce.

MSUM [recommended]

For performance reasons, it is recommended to conditionally execute printing informational messages when data items are being printed. (For OVM e users, this is not a requirement).

```
if (ovm_report_enabled(OVM_HIGH))
   ovm_report_info(get_type_name(),
      $psprintf("Monitor has collected a transaction %s", item.sprint()),
      OVM_HIGH);
```

OVM macros that wrap the ovm_report_* functions are provided on the OVM Contributions area.

```
`ovm_info(get_type_name(),
      $psprintf("Monitor has collected a transaction %s", item.sprint()),
      OVM_HIGH);
```

Documentation Compliance Checks

Most documentation checks will be manual as the formats and contents vary significantly.

DCBF

Driver documentation—verify that all drivers are documented (API and behavior).

DCCF

Constrainable fields—verify that the documentation describes all the knobs (user-constrainable fields) and indicates what their default values are.

DCUF

Applicable fields—verify that the documentation describes all the fields that users may use to control the knobs.

DCDC

Documentation—verify that documentation files exist in the "docs" directory. Check the docs directory for .txt, .doc, .pdf, .fm.

DCRN

Release notes—verify that release notes are provided in the "docs" directory.

DCEX

Examples—verify that the documentation gives sufficient examples to cover the most likely user scenarios.

DCFC

Features and controls—verify that the documentation covers all features and controls of the OVC.

DCID

Installation and demo—verify that the documentation describes the installation and demo processes

DCPA

Package architecture—verify that the documentation describes the architecture of the package and give an overview of its intended use.

DCPR [Recommended]

Proper documentation—verify that concepts introduced before being referred to.

DCRP

Recommended practice—verify that the documentation clearly differentiates between what is good and bad practice when using the package.

DCRS

Reset—verify that the documentation explains whether the package manages multiple resets during the test.

DCSD

Sequencer documentation—verify that all sequencer test interfaces are sufficiently documented.

DCSP

Support policies—verify that the documentation clearly defines the support polices for the package and indicate contact information for obtaining support.

DCST

OVC structure—verify that diagrams are provided to explain the structure of the OVC, typical environments and configurations, how to use scoreboards, the class diagram of the main units and classes.

General Deliverables Compliance Checks

GDNF

New functionality—verify that new functionality added since previous release is properly documented in the release notes.

GDNM

OVC name—verify that the OVC name in the user manual and the package are consistent.

GDPN

Protocol name and version—verify that the name and version number of protocol or architecture the OVC models is consistent.

End of Test Compliance Checks

ET

End of test—verify that there is a call to global_stop; verify there is no call to $finish from user code. Once objection mechanism is implemented (in OVM 2.0?) the check is modified to proper use of objection e.g. if objection is raised but never dropped by a component.

OVM-SV Specific Compliance Checks

These notes are based on the assumption there will be one class declaration per file, and for sequence or test libraries each sequence or test will be defined on a single file and the library will be created by a set on `include.

All checks should be done before code compilation (ncvlog) because for most of them, the error will be generated during runtime, which are more difficult to debug

Please also note these rules are not taking into consideration ML (multi-language), i.e. for example connection by TLM across language domains.

OVM1

Verify that every data item class declaration is extending from ovm_sequence_item or a derivative of ovm_sequence_item.

OVM2 [Recommended]

For the declaration of monitor, sequencer, driver, agent, environment, test and data items, check respectively the existence and consistence of:

```
`ovm_object_utils (<class name declaration>) // for classes derived from ovm_sequence_item
`ovm_component_utils (<class name declaration>) // for all the others except sequencer
`ovm_sequencer_utils (<class name declaration>) // for sequencer
`ovm_sequence_utils (<class name declaration>, <sequencer name>) // for ovm sequences
```

- If `ovm_*_utils_begin macro is used the existence of `ovm_*_utils_end must be checked
- If the pair of _begin/_end is satisfied, between the two macro call the `ovm_field_<type> (<variable or instance name match>, <OVM_FLAG>) may (but not necessarily) exist

Refer to OVM reference manual for all supported types, with particular attention to the enum type which requires the extra field for the enum type name, and the OVM_FLAG.

If these rules are not satisfied a warning should be issued, with error if _begin/_end pair is not respected.

Here below is an example of code to be checked (only relevant parts):

```
typedef enum bit {GOOD_PARITY, BAD_PARITY} parity_e;
class cdn_uart_frame extends ovm_sequence_item;
  rand bit start_bit;
  rand bit [7:0] payload;
  bit parity;
  rand bit [1:0] stop_bits;
  rand bit [3:0] error_bits;
  rand parity_e parity_type;
  rand int transmit_delay;
  `ovm_object_utils_begin(cdn_uart_frame)
    `ovm_field_int(start_bit, OVM_ALL_ON)
    `ovm_field_int(payload, OVM_ALL_ON)
    `ovm_field_int(parity, OVM_ALL_ON)
    `ovm_field_int(stop_bits, OVM_ALL_ON)
    `ovm_field_int(error_bits, OVM_ALL_ON)
    `ovm_field_enum(parity_e, parity_type, OVM_ALL_ON + OVM_NOCOMPARE)
    `ovm_field_int(transmit_delay, OVM_ALL_ON + OVM_DEC + OVM_NOCOMPARE + OVM_NOCOPY)
  `ovm_object_utils_end
```

OVM3

Check the existence of the new() method for classes extended from ovm_<base_class_declaration>: ovm_sequence_item, ovm_test, ovm_env, ovm_driver, etc.

If the class is inherited from a class using ovm_<base_class_declaration>, check also the usage of super.new() call inside the method new().

If rule is not satisfied, a warning should be generated.

For components use:

```
function new (string name, ovm_component parent);
    super.new(name, parent);
endfunction : new
```

For sequence and item use:

```
function new(string name = "cdn_uart_frame");
    super.new(name);
endfunction: new
```

OVM4 [Recommended]

The declaration of a sequence or virtual sequence class is derived from ovm_sequence or a derivative of ovm_sequence.

OVM5

Sequence and virtual sequence declaration contains proper usage of the `ovm_sequence_utils macro:

- `ovm_sequence_utils(seq_type, sequencer_type)
- Or `ovm_sequence_utils_begin() … `ovm_sequence_utils_end

If the rules are not satisfied an error should be generated, here below a simple code example:

```
class uart_incr_payload_seq extends ovm_sequence;
    function new(string name="uart_incr_payload_seq");
      super.new(name);
    endfunction
    // register sequence with a sequencer
   `ovm_sequence_utils(uart_incr_payload_seq, cdn_uart_sequencer)
```

OVM6

In the code of a sequence, the existence of the body() method should be verified. If the method does not exist an error should be generated. This check should be applied right after OVM4/5.

```
class uart_incr_payload_seq extends ovm_sequence;
  …
  virtual task body();
      ovm_report_info(get_type_name(),$psprintf,("Executing %0d times", cnt), OVM_LOW)
      for (int i = 0; i < cnt; i++)
      begin
        `ovm_do_with(tx_frame, {payload == start_payload +i*3; })
      end
  endtask
```

OVM7 [Recommended]

If OVM6 is satisfied, check inside the body() the usage of `ovm_do or `ovm_do_with macros (or any “sub do” macro, like `ovm_create – check OVM reference manual for all sub-macro definition).

If not satisfied, a warning should be generated. Please refer to the example added for OVM6

OVM8

If the methods pre_body() and post_body() are implemented on a sequence, they must include respectively the super.pre_body() and super.post_body(). If this rule is not satisfied an error must be generated.

OVM9 [Recommended]

All components implemented must be extended from the corresponding base class or a derivate of it, for example the monitor must be extended from ovm_monitor or its derivative.

This rule assumes the file name is aligned with the class type, for example the filename ‘cdn_apb_master_sequencer.sv’ should be used to declare ‘class apb_master_sequencer extends ovm_sequencer;’.

If this rule is not satisfied a warning should be generated.

OVM10

For sequencer or virtual sequencer declaration,

- Check the existence and consistence of `ovm_sequencer_utils (<class name declaration>).
- If `ovm_sequencer_utils_begin macro is used, the existence of `ovm_sequencer_utils_end must be checked
- If the pair of _begin/_end is satisfied, check that between the two macro calls the `ovm_field_<type> (<variable or instance name match>, <OVM_FLAG>) may optionally exist

Refer to the OVM reference manual for all supported types, with particular attention to the enum type which requires the extra field for the enum type name, and the OVM_FLAG.

```
class apb_master_sequencer extends ovm_sequencer;
  function new (string name="", ovm_component parent=null);
    super.new(name, parent);
    `ovm_update_sequence_lib_and_item(apb_transfer)
  endfunction : new
  `ovm_sequencer_utils(apb_master_sequencer)
endclass : apb_master_sequencer
```

OVM11

Sequencer classes derived from ovm_sequencer must include the proper macro in the new() method. Sequencers must have `ovm_update_sequence_lib_and_item() or `ovm_update_sequence_lib in new().

Please refer to example in OVM2.0.

If rule is not satisfied an error must be generated.

OVM15 [Recommended]

Inside a driver the method run() must exist and inside it there should be usage of get_next_item() and item_done(). If this rule is not satisfied a warning should be generated for missing get_next_item() or item_done(), an error should be generated for missing run() method.

```
class apb_master_driver extends ovm_driver;
  // Virtual interfaces, configuration units, new constructor omitted.
  task run();
    fork
      get_and_drive();
      reset_signals();
    join_none
  endtask : run

  protected task get_and_drive();
    // Method get_and_drive implementation omitted to focus only on calls to be checked
    …
    seq_item_port.get_next_item(req);
    …
    seq_item_port.item_done(rsp);
    …
  endtask : get_and_drive
```

OVM16 [Recommended]

If driver and monitor classes are implemented (passive VC will not have driver), a virtual interface declaration of is required. If the rule is not satisfied an error should be generated.

This example is related to the monitor and a similar declaration will be found in a driver:

```
virtual class cdn_uart_monitor extends ovm_monitor;
  virtual interface cdn_uart_if vif;
```

```
  …
endclass
```

OVM20

Inside a monitor class the method run() must exist and it should be checked.

If this rule is not satisfied an error should be generated.

```
virtual class cdn_uart_monitor extends ovm_monitor;
  …
  task run();
    // Implementation is irrelevant for check scope
  endtask
endclass
```

OVM22

The monitor class should at least include one covergroup and one coverpoint statement; the covergroup must be constructed in the new() method. If the covergroup is not defined a warning should be generated, but if it exists but not constructed an error should be generated.

Here below a simple example:

```
virtual class cdn_uart_monitor extends ovm_monitor;
  …
  covergroup uart_trans_frame_cg;
    stop_bits1 : coverpoint csr.nbstop;
    NUM_STOP_BITS : coverpoint csr.nbstop {
      bins ONE = {0};
      bins ONE_AND_HALF = {2'b01};
      bins TWO = {2};
      bins illegal  = {3};
    }
    …
  endgroup

  function new (string name = "", ovm_component parent = null);
    super.new(name, parent);
    uart_trans_frame_cg = new();
    uart_trans_frame_cg.set_inst_name ({get_full_name(), ".uart_trans_frame_cg"});
  endfunction: new
endclass
```

OVM23

To properly connect to scoreboard, the ovm_analysis_port should be implemented inside a monitor class and constructed in the new() method. If not found a warning should be generated. If found but not constructed an error should be generated.

```
virtual class cdn_uart_monitor extends ovm_monitor;
   ovm_analysis_port #(cdn_uart_frame) frame_collected_port;
```

```
  …
  function new (string name = "", ovm_component parent = null);
    super.new(name, parent);
    frame_collected_port = new("frame_collected_port", this);
  endfunction: new
```

OVM24 [Statistical]

Count the number of ovm_analysis_port and report it as statistic.

OVM28

In every agent check the existence of conditional code using a variable of type ovm_active_passive_enum, and verify the correct component creation/instantiation for driver and sequencer only if the configuration is active.

If the conditional code is not correct a warning should be generated.

```
class xbus_master_agent extends ovm_agent;
  xbus_master_driver driver;
  xbus_master_sequencer sequencer;
  xbus_master_monitor monitor;
  ovm_active_passive_enum is_active = OVM_ACTIVE;
  …
  function void build();
    super.build();
    monitor = xbus_master_monitor::type_id::create("monitor", this);
    if(is_active == OVM_ACTIVE) begin
      sequencer = xbus_master_sequencer::type_id::create("sequencer", this);
      driver = xbus_master_driver::type_id::create("driver", this);
    end
    …
  endfunction : build
endclass
```

OVM30 [Recommended]

In every agent check the existence of the method connect() and inside it the existence of conditional code using a variable of type ovm_active_passive_enum.

If conditional code is used verify the correct connection of virtual interface for monitor and driver, and optionally even for sequencer (rare, but sometime used). If the conditional code is not correct a warning should be generated.

```
  function void connect();
    if(is_active == OVM_ACTIVE) begin
      driver.seq_item_port.connect(sequencer.seq_item_export);
    end
  endfunction : connect
```

OVM36 [Recommended]

In the OVC SystemVerilog interface declaration at least one protocol checking assertion should be implemented. If not a warning message should be generated.

```
// SVA default clocking
```

```
wire ovm_assert_clk = sig_clock & has_checks;
default clocking master_clk @(negedge ovm_assert_clk);
endclocking

// Read and write never true at the same time
master_ReadOrWrite: assert property (
                  ((|sig_grant) |-> !(sig_read && sig_write)))
                  else
                  ovm_report_fatal("XBUS Interface","Read and Write true at the same time",OVM_NONE);
```

OVM45 [Recommended]

The class declaring the testbench(sve) is extended from ovm_env. If not an error should be generated.

```
class single_apbM_uart_RxTx_sve extends ovm_env;
  apb_env apb0;                    // APB UVC
  cdn_uart_env uart0;              // UART UVC
  cdn_uart_mod_env apb_uart0;
  …
endclass
```

OVM46 [Recommended]

The testbench(sve) class instantiates one or more env. If not a warning should be generated; please refer to example for OVM45

OVM49 [Recommended]

The test class must instantiate the testbench(sve). If not a warning should be generated.

```
class uart_incr_payload_test extends ovm_test;
  single_apbM_uart_RxTx_sve ve;
  …
endclass
```

OVM50 [Recommended]

In every monitor check the usage of begin_tr, end_tr. If not found a warning should be generated.

```
class cdn_uart_monitor extends ovm_monitor;
  …
  cdn_uart_frame cur_frame;
  …
  task collect_frame();
    …
    cur_frame = new;
    …
    // Begin Transaction Recording
    void'(this.begin_tr(cur_frame));
    …
    this.end_tr(cur_frame);
  endtask : collect_frame
```

```
endclass
```

OVM51

All VC components such as driver, monitor, sequencer, agent, env, etc. must be created using the create() call inside the build() method. If not a warning should be generated.

```
class xbus_master_agent extends ovm_agent;
  xbus_master_driver driver;
  xbus_master_sequencer sequencer;
  xbus_master_monitor monitor;
  …
  function void build();
    super.build();
    monitor = xbus_master_monitor::type_id::create("monitor", this);
    if(is_active == OVM_ACTIVE) begin
      sequencer = xbus_master_sequencer::type_id::create("sequencer", this);
      driver = xbus_master_driver::type_id::create("driver", this);
    end
    …
  endfunction : build
endclass
```

OVM52

Similar to OVM51, all objects are allocated by using the create() call.

```
    req = xbus_transfer::type_id::create("req", this);
```

OVM53

Inside the build() method of a test class, at least one set_config_*() must be used, to demonstrate the VC configuration capability. If not present a warning should be generated.

```
class u2a_a2u_full_rand_test extends ovm_test;
  …
  function void build();
    super.build();
    …
    set_config_string("ve.virtual_sequencer", "default_sequence", "concurrent_u2a_a2u_rand_trans");
  endfunction : build
endclass
```

OVM55

The method run, declared in a monitor or a driver classes, contains task calls between fork and join_none. The existence of these called tasks should also be checked; otherwise an error should be generated.

Please refer to the example on OVM15

OVM56 [Recommended]

In interface declaration and monitor class, respectively assertions and the collection of coverage should be inside a conditional block.

OVM57

All signals used by a monitor or a driver (or sometimes by a sequencer) should come from a virtual interface and not directly from the DUT. If not an error should be generated.

```
class cdn_uart_tx_driver extends ovm_driver #(cdn_uart_frame);
  …
  virtual interface cdn_uart_if vif;
  …
  // Example of method get_and_drive accessing to the signals through the virtual interface
  task get_and_drive();
    @(negedge vif.reset);
    ovm_report_info(get_type_name(),"Reset asserted",OVM_MEDIUM)
   forever begin
      @(posedge vif.clock iff (vif.reset));
      seq_item_port.get_next_item(req);
      send_tx_frame(req);
      seq_item_port.item_done();
    end
  endtask
```

OVM58

The method build() must call the super.build(). If not an error should be generated

OVM59

TLM connects must occur in the connect() phase.

OVM60

Component creation – check that all the components are created using the OVM 2.0 creation style. A warning should be generated if old style is used.

```
monitor = xbus_master_monitor::type_id::create("monitor", this);
```

OVM61

Deprecated constructs – warn if deprecated constructs are being used in the code. List of deprecated constructs TBD.