Verification Intellectual Property (VIP) Recommended Practices

Version 1.0

August 25, 2009

Copyright© 2008-2009 by Accellera. All rights reserved.

Notices

Accellera Standards documents are developed within Accellera and the Technical Committees of Accellera Organization, Inc. Accellera develops its standards through a consensus development process, approved by its members and board of directors, which brings together volunteers representing varied viewpoints and interests to achieve the final product. Volunteers are not necessarily members of Accellera and serve without compensation. While Accellera administers the process and establishes rules to promote fairness in the consensus development process, Accellera does not independently evaluate, test, or verify the accuracy of any of the information contained in its standards.

Use of an Accellera Standard is wholly voluntary. Accellera disclaims liability for any personal injury, property or other damage, of any nature whatsoever, whether special, indirect, consequential, or compensatory, directly or indirectly resulting from the publication, use of, or reliance upon this, or any other Accellera Standard document.

Accellera does not warrant or represent the accuracy or content of the material contained herein, and expressly disclaims any express or implied warranty, including any implied warranty of merchantability or suitability for a specific purpose, or that the use of the material contained herein is free from patent infringement. Accellera Standards documents are supplied "**AS IS**."

The existence of an Accellera Standard does not imply that there are no other ways to produce, test, measure, purchase, market, or provide other goods and services related to the scope of an Accellera Standard. Furthermore, the viewpoint expressed at the time a standard is approved and issued is subject to change due to developments in the state of the art and comments received from users of the standard. Every Accellera Standard is subjected to review periodically for revision and update. Users are cautioned to check to determine that they have the latest edition of any Accellera Standard.

In publishing and making this document available, Accellera is not suggesting or rendering professional or other services for, or on behalf of, any person or entity. Nor is Accellera undertaking to perform any duty owed by any other person or entity to another. Any person utilizing this, and any other Accellera Standards document, should rely upon the advice of a competent professional in determining the exercise of reasonable care in any given circumstances.

Interpretations: Occasionally questions may arise regarding the meaning of portions of standards as they relate to specific applications. When the need for interpretations is brought to the attention of Accellera, Accellera will initiate action to prepare appropriate responses. Since Accellera Standards represent a consensus of concerned interests, it is important to ensure that any interpretation has also received the concurrence of a balance of interests. For this reason, Accellera and the members of its Technical Committees are not able to provide an instant response to interpretation requests except in those cases where the matter has previously received formal consideration.

Comments for revision of Accellera Standards are welcome from any interested party, regardless of membership affiliation with Accellera. Suggestions for changes in documents should be in the form of a proposed change of text, together with appropriate supporting comments. Comments on standards and requests for interpretations should be addressed to:

Accellera Organization
1370 Trancas Street #163
Napa, CA 94558
USA

Note: Attention is called to the possibility that implementation of this standard may require use of subject matter covered by patent rights. By publication of this standard, no position is taken with respect to the existence or validity of any patent rights in connection therewith. Accellera shall not be responsible for identifying patents for which a license may be required by an Accellera standard or for conducting inquiries into the legal validity or scope of those patents that are brought to its attention.

Accellera is the sole entity that may authorize the use of Accellera-owned certification marks and/or trademarks to indicate compliance with the materials set forth herein.

Authorization to photocopy portions of any individual standard for internal or personal use must be granted by Accellera Organization, Inc., provided that permission is obtained from and any required fee is paid to Accellera. To arrange for authorization please contact Lynn Horobin, Accellera, 1370 Trancas Street #163, Napa, CA 94558, phone (707) 251-9977, e-mail lynn@accellera.org. Permission to photocopy portions of any individual standard for educational classroom use can also be obtained from Accellera.

Suggestions for improvements to the Verification Intellectual Property Recommended Practices are welcome. They should be sent to the VIP email reflector

vip-tc@lists.accellera.org

The current Working Group's website address is

www.accellera.org/activities/vip

Introduction

The purpose of this set of recommended practices is to address a significant industry need to make verification intellectual property (VIP) interoperate. To increase productivity, many companies are opting to use electronic design automation (EDA) solutions for verification methodologies. The advantage of these methodologies is they are pre-packaged to abstract away and compartmentalize many of the standard components used by verification teams. A methodology and supporting library is given to the validation teams, which is then used to construct and create a verification environment.

However, there are several competing methodologies, the two most significant being the Open Verification Methodology (OVM)[a] from Cadence and Mentor and the Verification Methodology Manual (VMM) from Synopsys. Both open source methodologies are compelling solutions, but both approach the problem of structuring and building a verification environment in significantly different ways. Ultimately, both methodologies are packaged as a set of base class libraries along with a reference document of how to use the libraries and their best practices. Much of the underlying details are hidden from end-users, thereby enabling abstraction. End-user companies typically have a handful of expert users who understand the technology well enough to use it, but not well enough to modify the underlying libraries.

Now, companies are looking seriously at being able take existing VIP from one methodology and use it with VIP from the other methodology. Typically, legacy code is written in one methodology and companies either want to migrate to the other or they need to use existing VIP from the other library. The integration at times can be straight forward, like attaching a protocol monitor, but often the work is extensive and requires a significant amount of knowledge about both methodologies to make them interoperate correctly.

This document offers a solution to the VIP interoperability issue. It starts out by stepping the user through a high-level overview, which is intended to quickly redirect the user to the practice(s) that address their

[a]For information on references, see Chapter 2.

specific circumstances. The recommended practices chapter provides solutions for the various challenges a verification environment creator faces when integrating a VIP from a different methodology. There is also a chapter containing the application programming interfaces (APIs) associated with this interoperability document, defining a proposed reference library.

This document is not intended to be read linearly, but to serve as a cookbook which guides the user through the process of creating a verification environment made of components from different methodologies.

Selection of the interoperability model

When deciding on the type of interoperability this set of recommended practices would support, the VIP Technical SubCommittee (TSC) identified two interoperability models: interconnected and encapsulated. The VIP TSC then chose to only support the interconnected model within this document.

To put users in the right context as to how interoperability can be achieved, the different interoperability models are described here. The user should really only care about the interconnected model and just be aware of any others.

a) The *interconnected model* enables taking a VIP component implemented using a class library and using it within an environment along with other VIPs implemented using another base class library. This model requires the user handling the interoperability to have a good understanding of both VMM and OVM. This model does not require changes to the original VIP interface.

   Since the VIP TSC decided to deal only with the interconnected model, this document only describes interconnected model best practices.

b) The *encapsulated model* requires wrapping a VIP component implemented using a base class library within a wrapper implemented using the other base class library, so the user is not aware when he/she is using a VIP component originally implemented in a different methodology. In this case, the wrapper is responsible for mapping the functions required by the user's methodology to the appropriate functions provided in the original methodology.

Contributors

The following individuals contributed to the creation, editing, and review of the Verification Intellectual Property Recommended Practices.

Tom Alsop	Intel Corporation	VIP Workgroup Co-Chair
Janick Bergeron	Synopsys Inc.	
Dennis Brophy	Mentor Graphics	
Joe Daniels		Technical Editor
Ken Davis	Freescale Semiconductor	
Adam Erickson	Mentor Graphics	
Joshua Filliater	Denali Software Inc.	
Tom Fitzpatrick	Mentor Graphics	
Bill Flanders	Intel Corporation	
JL Gray	Verilab	
David Jones	XtremeEDA	
Dhrubajyoti Kalita	Intel Corporation	
Adiel Khan	Synopsys Inc.	
Neil Korpusik	Sun Microsystems	
Stan Korlikoski	Cadence Design Systems	

Sanjiv Kumar	Denali Software Inc.	
Jay Lawrence	Cadence Design Systems	
Hiller Miller	Freescale Semiconductor	VIP Workgroup Co-Chair
Michael Rohleder	Freescale Semiconductor	
Sharon Rosenburg	Cadence Design Systems	
Ambar Sarkar	Paradigm Works Inc.	
Adam Sherer	Cadence Design Systems	
Mark Strickland	Cisco Systems	
Amre Sultan	XtremeEDA	
Yatin Trivedi	Synopsys Inc.	
Alex Wakefield	Synopsys Inc.	

Contents

1. Overview 1

1.1 Purpose 1
1.2 Scope 1
1.3 Recommended practices template 1
1.4 Conventional notations 2
1.5 Contents of this document 2

2. Normative references 3

3. Definitions, acronyms, and abbreviations 3

3.1 Definitions 3
3.2 Acronyms and abbreviations 4

4. Integrating verification components 5

4.1 Motivation 5
4.2 Interoperability vs. migration 5
4.3 Integrating mixed methodologies 6
4.4 Learning OVM and VMM 6
4.5 Reusing infrastructure 6
4.5.1 Choosing the level of reuse 6
4.5.2 Grouping the foreign component(s) and adapters into a container 7
4.6 Instantiating the foreign component(s) in a testbench 8
4.6.1 Instantiating and connecting components 8
4.6.2 Configuring testbenches 8
4.7 Creating reusable sequences/scenarios 9
4.7.1 Creating sequences/scenarios for a single stream 9
4.7.2 Developing system-level multi-channel sequences (multi-stream scenarios) 9
4.8 Writing tests 9
4.9 Running simulations, debugging them, and tracing any messages 9

5. Recommended practices 11

5.1 OVM-on-top phase synchronization 12
5.1.1 Practice name 12
5.1.2 Intent 12
5.1.3 Applicability 12
5.1.4 Structure 13
5.1.5 Collaboration 15
5.1.6 Implementation 15
5.1.7 Sample code 16
5.2 VMM-on-top phase synchronization 17
5.2.1 Practice name 17
5.2.2 Intent 17
5.2.3 Applicability 17
5.2.4 Structure 17
5.2.5 Collaboration 20
5.2.6 Implementation 20
5.2.7 Sample code 21

5.3 Meta-composition ... 21
5.3.1 Practice name ... 21
5.3.2 Intent ... 22
5.3.3 Applicability ... 22
5.3.4 Structure ... 22
5.3.5 Collaboration ... 23
5.3.6 Implementation ... 23
5.3.7 Sample code ... 23
5.3.8 Printing the environment topology ... 28
5.4 VIP configuration ... 29
5.4.1 Practice name ... 29
5.4.2 Intent ... 29
5.4.3 Applicability ... 29
5.4.4 Structure ... 30
5.4.5 Collaboration ... 30
5.4.6 Implementation ... 30
5.4.7 Sample code ... 30
5.5 Data conversion ... 31
5.5.1 Practice name ... 31
5.5.2 Intent ... 31
5.5.3 Applicability ... 32
5.5.4 Structure ... 32
5.5.5 Collaboration ... 32
5.5.6 Implementation ... 32
5.5.7 Sample code ... 33
5.6 TLM to channel ... 34
5.6.1 Practice name ... 34
5.6.2 Intent ... 34
5.6.3 Applicability ... 35
5.6.4 Structure ... 35
5.6.5 Collaboration ... 37
5.6.6 Implementation ... 40
5.6.7 Sample code ... 40
5.7 Channel to TLM ... 41
5.7.1 Practice name ... 41
5.7.2 Intent ... 41
5.7.3 Applicability ... 42
5.7.4 Structure ... 42
5.7.5 Collaboration ... 45
5.7.6 Implementation ... 47
5.7.7 Sample code ... 47
5.8 Analysis to channel / Channel to analysis ... 48
5.8.1 Practice name ... 48
5.8.2 Intent ... 49
5.8.3 Applicability ... 49
5.8.4 Structure ... 49
5.8.5 Collaboration ... 51
5.8.6 Implementation ... 51
5.8.7 Sample code ... 51
5.9 Notify to analysis / Analysis to notify ... 53
5.9.1 Practice name ... 53
5.9.2 Intent ... 53
5.9.3 Applicability ... 53
5.9.4 Structure ... 53

5.9.5 Collaboration ... 55
5.9.6 Implementation ... 55
5.9.7 Sample code ... 55
5.10 Callback adapter ... 57
5.10.1 Practice name ... 57
5.10.2 Intent ... 57
5.10.3 Applicability ... 58
5.10.4 Structure ... 58
5.10.5 Collaboration ... 59
5.10.6 Implementation ... 59
5.10.7 Sample code ... 59
5.11 Sequence and scenario composition ... 60
5.11.1 Practice name ... 60
5.11.2 Intent ... 61
5.11.3 Applicability ... 61
5.11.4 Structure ... 61
5.11.5 Collaboration ... 61
5.11.6 Implementation ... 62
5.11.7 Sample code ... 62
5.12 Messaging ... 65
5.12.1 Practice name ... 65
5.12.2 Intent ... 66
5.12.3 Applicability ... 67
5.12.4 Structure ... 67
5.12.5 Collaboration ... 68
5.12.6 Implementation ... 68
5.12.7 Sample code ... 68

6. Application programming interface (API) ... 71

6.1 Common parameters ... 71
6.1.1 OVM ... 71
6.1.2 OVM_REQ ... 71
6.1.3 OVM_RSP ... 71
6.1.4 VMM ... 71
6.1.5 VMM_REQ ... 71
6.1.6 VMM_RSP ... 71
6.1.7 OVM2VMM ... 71
6.1.8 OVM2VMM_REQ ... 71
6.1.9 OVM2VMM_RSP ... 72
6.1.10 VMM2OVM ... 72
6.1.11 VMM2OVM_REQ ... 72
6.1.12 VMM2OVM_RSP ... 72
6.2 avt_converter #(IN,OUT) ... 72
6.2.1 Declaration ... 72
6.2.2 Parameters ... 72
6.2.3 Methods ... 72
6.3 avt_match_ovm_id ... 73
6.3.1 Declaration ... 73
6.3.2 Parameters ... 73
6.3.3 Methods ... 73
6.4 avt_ovm_vmm_env ... 73
6.4.1 Hierarchy ... 73
6.4.2 Declaration ... 74

6.4.3 Methods ... 74
6.4.4 Variables ... 76
6.5 avt_vmm_ovm_env ... 76
6.5.1 Hierarchy ... 76
6.5.2 Declaration ... 77
6.5.3 Methods ... 77
6.5.4 Macros ... 78
6.6 avt_tlm2channel ... 78
6.6.1 Hierarchy ... 78
6.6.2 Declaration ... 78
6.6.3 Parameters ... 79
6.6.4 Communication interfaces ... 79
6.6.5 Methods ... 79
6.6.6 Variables ... 80
6.7 avt_channel2tlm ... 80
6.7.1 Hierarchy ... 80
6.7.2 Declaration ... 81
6.7.3 Parameters ... 81
6.7.4 Communication interfaces ... 81
6.7.5 Methods ... 82
6.7.6 Variables ... 82
6.8 avt_analysis_channel ... 83
6.8.1 Hierarchy ... 83
6.8.2 Declaration ... 83
6.8.3 Parameters ... 83
6.8.4 Communication interfaces ... 84
6.8.5 Methods ... 84
6.9 avt_analysis2notify ... 84
6.9.1 Hierarchy ... 84
6.9.2 Declaration ... 85
6.9.3 Parameters ... 85
6.9.4 Communication interfaces ... 85
6.9.5 Methods ... 85
6.9.6 Variables ... 85
6.10 avt_notify2analysis ... 86
6.10.1 Hierarchy ... 86
6.10.2 Declaration ... 86
6.10.3 Parameters ... 86
6.10.4 Communication interfaces ... 86
6.10.5 Methods ... 87
6.10.6 Variables ... 87

Appendix A Bibliography ... 89

1. Overview

This chapter defines the scope and purpose of the Verification Intellectual Property (VIP) recommended practices, highlights the basic concepts related to using this document, and summarizes the remainder of these recommended practices.

1.1 Purpose

This document's purpose is to assist the environment verification engineer to design and implement an Open Verification Methodology (OVM) environment that needs to import and use Verification Methodology Manual (VMM) VIP or a VMM environment that needs to import and use OVM VIP. The document can also assist the VMM and OVM VIP developer on how to best package the VIP so it can work effectively in the respective OVM and VMM environments.

There are some scenarios for which this document can assist the verification community.

- A VIP provider provides a VMM VIP to an OVM house or vice-versa.
- A VMM house decides to transition to OVM or vice-versa. With this, they can continue to use their pre-existing VIP components.
- A system house needs to integrate designs independently verified using OVM and VMM into a VMM or OVM-based top-level verification environment.
- A company that has its own base class and would like to migrate to OVM or VMM can implement a similar solution based on this one.

1.2 Scope

The scope of the document is to deal with all types of OVM and VMM IEEE Std 1800™-compliant[1] VIP.

1.3 Recommended practices template

Each "best practice" is described using the following template. See also Chapter 5.

Practice name and classification: A descriptive and unique name that helps in identifying and referring to the practice.

Also known as: Other names for the practice.

Related practices: Other practices that have some relationship with the practice; discussion of the differences between the practice and similar practices.

Intent: A description of the goal behind the practice and the reason for using it.

Motivation (forces): A scenario consisting of a problem and a context in which this practice can be used.

Consequences: A description of the results, side effects, and trade-offs caused by using the practice.

Applicability: Situations in which this practice is usable; the context for the practice.

Structure: A graphical representation of the practice. Class diagrams and interaction diagrams may be used for this purpose.

[1]For information on references, see Chapter 2.

Participants: A listing of the classes and objects used in the practice and their roles in the design.

Collaboration: A description of how classes and objects used in the practice interact with each other.

Implementation: A description of an implementation of the practice; the solution part of the practice.

Sample code: An illustration of how the practice can be used in a programming language.

1.4 Conventional notations

The meta-syntax for the description of the examples, keywords, variables, etc. uses the conventions shown in Table 1.

Table 1—Document conventions

Visual cue	Represents
`courier`	The `courier` font indicates OVM, VMM, or SystemVerilog terms or examples. For example: `subenv = new("vmm_env",this);`
bold	The **bold** font is used to indicate objects, components, and key terms—text that shall be typed exactly as it appears. For example, in the following line, "avt_ovm_vmm_env" is a component: The **avt_ovm_vmm_env** creates ...
italic	The *italic* font indicates when definitions or variables occur. For example, a "method" needs to be specified in the following line (after the "super." key term): Using a customized wrapper also requires the user to call `super.`*`method`*`()`.

1.5 Contents of this document

The organization of the remainder of this document is as follows:

- Chapter 2 provides references to applicable standards that are presumed or required for using these recommended practices.
- Chapter 3 defines terms and acronyms used throughout this document.
- Chapter 4 defines the process for integrating and using mixed methodology VIPs.
- Chapter 5 details each of the VIP recommended practices.
- Chapter 6 defines the application programming interface (API) for each of the VIP classes.

2. Normative references

The following referenced documents are indispensable for the application of this document (i.e., they must be understood and used, so each referenced document is cited in text and its relationship to this document is explained). For dated references, only the edition cited applies. For undated references, the latest edition of the referenced document (including any amendments or corrigenda) applies.

IEEE Std 1800™, IEEE Standard for SystemVerilog Unified Hardware Design, Specification and Verification Language.[2, 3]

3. Definitions, acronyms, and abbreviations

For the purposes of this document, the following terms and definitions apply. The *IEEE Standards Dictionary: Glossary of Terms & Definitions*[4] should be referenced for terms not defined in this clause.

3.1 Definitions

3.1.1 adapter: A **component** that connects one transaction-level interface to another, including converting the transaction object format, if necessary. In the context of this document, this usually refers to the connection and conversion of **transaction**s from one methodology to the other. Also referred to as a *bridge*.

3.1.2 channel: A transaction-level communication conduit. Usually refers to a `vmm_channel` instance.

3.1.3 component: A piece of VIP that provides functionality and interfaces. Also referred to as a *transactor*.

3.1.4 consumer: A verification component that receives **transaction**s from another **component**.

3.1.5 driver: A component responsible for executing or otherwise processing **transaction**s, usually interacting with the device under test (DUT) to do so.

3.1.6 export: A transaction level modeling (TLM) interface that provides the implementation of methods used for communication. Used in OVM to connect to a port.

3.1.7 foreign methodology: A verification methodology that is different from the methodology being used for the majority of the verification environment.

3.1.8 generator: A verification component that provides transactions to another **component**. Also referred to as a *producer*.

3.1.9 port: A TLM interface that defines the set of methods used for communication. Used in OVM to connect to an export.

3.1.10 primary (host) methodology: The methodology that manages the top-level operation of the verification environment and with which the user/integrator is presumably more familiar.

3.1.11 request: A **transaction** that provides information to initiate the processing of a particular operation.

---

[2]IEEE publications are available from the Institute of Electrical and Electronics Engineers, Inc., 445 Hoes Lane, Piscataway, NJ 08854, USA (http://standards.ieee.org/).

[3]The IEEE standards or products referred to in this clause are trademarks of the Institute of Electrical and Electronics Engineers, Inc.

[4]The *IEEE Standards Dictionary: Glossary of Terms & Definitions* is available at http://shop.ieee.org/.

3.1.12 response: A **transaction** that provides information about the completion or status of a particular operation.

3.1.13 scenario: A VMM object that procedurally defines a set of **transaction**s to be executed and/or controls the execution of other scenarios.

3.1.14 scoreboard: The mechanism used to dynamically predict the response of the design and check the observed response against the predicted response. Usually refers to the entire dynamic response-checking structure.

3.1.15 sequence: An OVM object that procedurally defines a set of **transaction**s to be executed and/or controls the execution of other sequences.

3.1.16 test: Specific customization of an environment to exercise required functionality of the DUT.

3.1.17 testbench: The structural definition of a set of verification components used to verify a DUT. Also referred to as a *verification environment*.

3.1.18 transaction: A class instance that encapsulates information used to communicate between two or more **component**s.

3.1.19 transactor: See *component*.

3.2 Acronyms and abbreviations

AVT Accellera VIP Technical subcommittee

API application programming interface

EDA electronic design automation

FIFO first-in, first-out

HDL hardware description language

IP intellectual property

OSCI Open SystemC Initiative

OVC OVM verification component

OVM Open Verification Methodology

DUT device under test

TLM transaction level modeling

VIP verification intellectual property

VMM Verification Methodology Manual

4. Integrating verification components

4.1 Motivation

When a design project is started, the verification architect ideally leverages existing VIP as much as possible before deciding to create new components. Building reusable VIP is time-consuming and often the new component is neither reliable nor mature enough to ensure correctness. While both OVM and VMM are self-consistent and provide guidelines and technology to ensure reusability, trying to use an OVM VIP in a VMM testbench (or vice-versa) exposes some of the different philosophies they hold. Challenges the user may experience when attempting to reuse a VIP written in a different methodology include:

- Instantiating and building of the component within the testbench
- Coordinating different simulation phases
- Configuring components to operate properly in the desired context
- Orchestrating and coordinating stimulus and other traffic between components
- Passing data types between components
- Distributing notifications across the testbench
- Issuing and controlling messages

This document provides a recommended set of best practices to address each of these challenges. This chapter introduces the process of integrating existing reusable environments into a testbench with relevant references to the more elaborate use-cases.

4.2 Interoperability vs. migration

When a verification architect examines the trade-offs in reusing an existing VIP block in a new environment that is based on a different methodology library, a decision needs to be made whether to make the block interoperate with the new environment or to migrate the VIP block by rewriting it using the primary methodology. While creating a VIP from scratch can be time consuming, large blocks of code can often be easily mapped from one methodology to the other. For example, in many cases the driving logic could be mapped from one methodology to the other.

a) Advantages of interoperating using existing components:
   1) User does not need to know the protocol or the internal implementation.
   2) The initial effort is small.
   3) This might be the only option, as sometimes users do not have access to the VIP code (encrypted or commercial code).
   4) If a different team maintains the VIP code, rewriting requires maintaining two different versions of the component.

b) Advantages of rewriting components to use the same methodology:
   1) If the company decides to move to a single methodology, long-term rewriting makes it easier to support components.
   2) If engineers will use one methodology in the long term, rewriting ensures an easier learning curve and more expertise in a single methodology.
   3) If many changes are expected in the existing component, it may be easier to rewrite it into a common methodology.

This document focuses on interoperability (the first option).

4.3 Integrating mixed methodologies

The process of integrating and using mixed methodology VIPs includes the following steps.

a) Learn the OVM and VMM methodologies (see 4.4); then, choose a primary methodology.

b) Determine the level of reuse and infrastructure needed to embed the foreign components in a testbench (see 4.5).

   1) Embed foreign components directly, along with necessary adapters. OR

   2) Group foreign components and adapters in a component written in the primary methodology to present, consolidate, and simplify the interactions.

c) Instantiate the foreign component(s) into a testbench (see 4.6).

   1) Adjust the configuration as needed.

   2) Connect components.

d) Create reusable sequences or scenarios (see 4.7):

   1) For a single channel (stream);

   2) For multi-channel (multi-stream) sequences.

e) Write tests (see 4.8):

   1) For configuration control;

   2) For layer constraints on top of transaction types;

   3) For control and coordination sequences.

f) Run simulations, debug them, and trace any messages (see 4.9).

These steps are further explained in the following sections.

4.4 Learning OVM and VMM

The recommended practices documented herein presume a working knowledge of the OVM and VMM methodologies ([B3] and [B4]). A verification team usually includes verification architects who create the testbench infrastructure and test writers who create tests and run simulations on existing testbenches. The verification architect needs to understand both methodologies and the interoperability practices specified in this document to wrap VMM IP within OVM or vice-versa. In the context of this document, the architect who implements such an interoperable environment is referred to as the *integrator*. Test writers need to focus on the control and execution of sequences/scenarios and test writing.

4.5 Reusing infrastructure

This step focuses on determining what level of reuse and infrastructure are necessary to begin integrating and interoperating.

4.5.1 Choosing the level of reuse

Verification components can be reused at various levels.

a) At the component (transactor) level—this means separate reuse of monitors, drivers, etc. This level of reuse requires the integrator to know the various component names, configuration modes, roles, and connection requirements. The integrator also needs to integrate the low-level component at every integration level.

b) At the interface level—as part of the reusable environment, the VIP developer connects the driver, generator, and monitor of a single protocol, so a cluster of pre-connected components can be used

without knowing the internal implementation or hookup; e.g., a pre-connected environment with stimuli, checking, and coverage for a PCI-E environment.

c) For an arbitrary group of classes—grouping IP components in this way is typically project-specific and may make horizontal reuse more difficult.

d) In complete testbenches—the entire testbench, including all connections, is leveraged. This kind of reuse is possible when all interfaces and functionality remain similar (such as the next generation of the same chip). This is the quickest way to migrate a full testbench between methodologies, but has limited reuse potential.

The optimal reuse level is one where substantial amounts of VIP can be grouped together in ways that are not project-specific. This allows the VIP to be leveraged in more projects, saving effort for the integrator. Even if the foreign VIP was created with transactor-level reuse in mind, it can be bundled it in a larger component, which often improves its ability to be reused.

Tests may also be reused under some circumstances, although typically sequences or scenarios are the test-level point of reuse (see 4.7).

4.5.2 Grouping the foreign component(s) and adapters into a container

Since integrating foreign VIP requires one or more adapter components (see Chapter 5), it may be convenient to group the VIP and adapter(s) into a single container component. This requires the integrator to create a parent component (a "container") in the primary methodology that directly instantiates and optionally configures the foreign VIP. The container is instantiated in the primary environment like any other component. All datatype conversion and other communication translation is performed locally inside the container and is otherwise isolated from the primary environment.

a) If VMM transactors need to be placed within an OVM testbench (see Figure 1 and 5.1):

   1) Create TLM ports/exports for the container and use adapters to connect them to the appropriate `vmm_channel` parameters of the VMM component(s) (see 5.7), and convert between `vmm_data` and `ovm_transactions` or `ovm_sequence_items` (see 5.5).

   2) Translate external notifications to analysis ports (see 5.9).

   3) Elevate configuration options to allow using the OVM `set_config_*` mechanism and the OVM factory.

   4) Call the needed component `build()` functions from the container `build()`.

   The VMM run-time phases are automatically called from the OVM's `run()` phase.

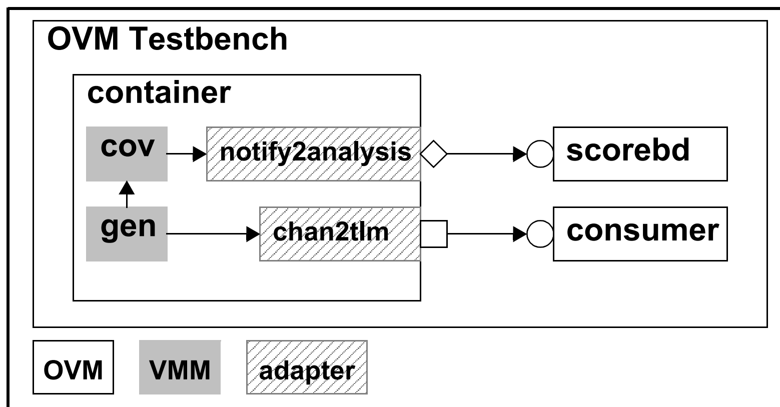

Figure 1—Incorporating VMM VIP into OVM

b) If OVM components are integrated within a VMM container (see Figure 2 and 5.2):

1) Create VMM channels and use the adapters to communicate `vmm_data` instead of `ovm_transactions` (see 5.6).
2) Translate analysis ports to VMM notifiers (see 5.9).
3) Elevate configuration options to allow use of `constructor` arguments or set values directly from the VMM parent instead of requiring the VMM parent to use the OVM `set_config_*` mechanism.
4) Call the `ovm_build()` function from the container.

The OVM phases are called at the appropriate time from the VMM environment.

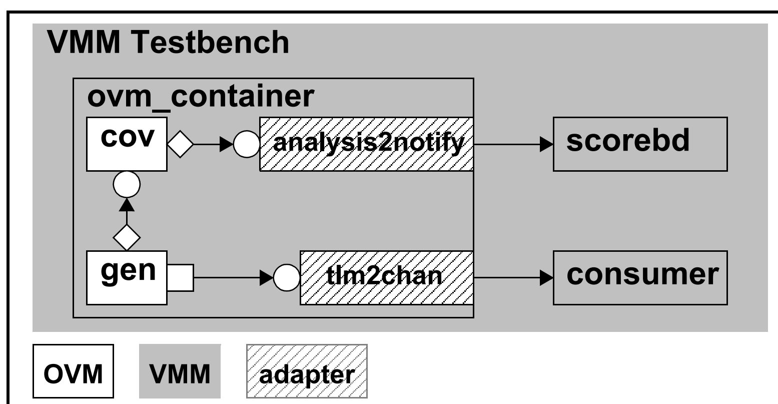

Figure 2—Incorporating OVM VIP into VMM

See also 5.3.

4.6 Instantiating the foreign component(s) in a testbench

This step focuses on instantiating, connecting, and configuring components.

4.6.1 Instantiating and connecting components

Once the container has been implemented, it needs to be instantiated in the primary environment. In OVM-based environments, the container class may be constructed directly or instantiated via the OVM factory to enable unplanned extensions, after which the TLM ports/exports are connected. In VMM-based environments, the container class is constructed with communication channels passed in as `constructor` arguments.

4.6.2 Configuring testbenches

Configuration is typically propagated top-down. The testbench top decides or randomizes the needed configuration and proliferates this to the sub-components. VMM and OVM have different ways to proliferate configuration. VMM uses the `constructor`'s arguments, while OVM uses the `set_config_*` interface. See also 5.4.

4.7 Creating reusable sequences/scenarios

Ideally, a reusable component provides a library of component-specific scenarios or sequences a test writer can use for productive test creation to describe a particular stream of transactions. The user may enhance the provided sequences/scenarios to adjust them (e.g., for device-specific memory constraints). The user may also add project-specific (reusable) sequences or scenarios. OVM and VMM each support two kinds of sequences or scenarios. VMM provides a *single channel scenario* that controls one channel and a *multi-channel scenario* that controls and coordinates between multiple scenarios and/or channels. In OVM, a *sequence* may control a single sequencer or it may control and coordinate between multiple other sequences. Such a coordinating sequence is referred to conceptually as a *virtual sequence*.

4.7.1 Creating sequences/scenarios for a single stream

To create a reusable transaction stream for a single target, the user should use the stimulus mechanism related to the target component (e.g., for a VMM generator in a primarily-OVM testbench, create `vmm_scenario`s for the VMM generator; for an OVM sequencer in a primarily-VMM testbench, create `ovm_sequence`s for the sequencer).

4.7.2 Developing system-level multi-channel sequences (multi-stream scenarios)

In some environments, a user may want to coordinate data and timing across multiple channels. Multi-channel sequences are typically created by the testbench integrator, who knows the number and nature of the interfaces in the design. This is best done in the primary methodology by calling a VMM sequence within an OVM virtual sequence or an OVM sequence within a VMM multi-stream scenario. See also 5.11.

4.8 Writing tests

While writing tests, a user may wish to modify the testbench configuration for a certain mode, layer constraints on top of data-items, determine the sequences that are executed, and much more. For this, the user should use the primary testbench test mechanism.

4.9 Running simulations, debugging them, and tracing any messages

OVM and VMM each provide a robust infrastructure for issuing messages throughout the simulation and for controlling the format and verbosity of messages. Defining the `OVM_ON_TOP` compiler directive allows VMM messages to be issued through the OVM reporting mechanism, while defining the `VMM_ON_TOP` compiler directive allows OVM messages to be issued through the VMM reporting mechanism. In either case, controlling the verbosity or other message management shall be done in the component's base methodology. See also 5.12.

5. Recommended practices

This chapter defines the VIP recommended practices; Table 2 shows their intent. All VIP recommended practices are based on the SystemVerilog syntax of IEEE Std 1800™ and assume an implementation of the class library and API documented in Chapter 6.

Table 2—Practices

Section	Practice name	Intent
5.1	*OVM-on-top phase synchronization*	Synchronize OVM and VMM phases in a primarily-OVM environment.
5.2	*VMM-on-top phase synchronization*	Synchronize OVM and VMM phases in a primarily-VMM environment.
5.3	*Meta-composition*	Compose hierarchical components with subcomponents from different methodologies.
5.4	*VIP configuration*	Set structural and run-time parameters for reusable VIP.
5.5	*Data conversion*	Convert data items from one methodology to the other.
5.6	*TLM to channel*	Establish semantic compatibility between OVM TLM ports/exports and VMM channels.
5.7	*Channel to TLM*	Establish semantic compatibility between VMM channels and OVM TLM ports/exports.
5.8	*Analysis to channel / Channel to analysis*	Establish semantic compatibility between OVM analysis ports/exports and VMM channels.
5.9	*Notify to analysis / Analysis to notify*	Establish semantic compatibility between OVM analysis ports/exports and VMM notify objects.
5.10	*Callback adapter*	Establish semantic compatibility between VMM callback methods and OVM analysis ports.
5.11	*Sequence and scenario composition*	Allow OVM sequences to call VMM scenarios and vice-versa.
5.12	*Messaging*	Allow messages generated by either methodology to be handled by the desired methodology's messaging system.

Because of fundamental differences between OVM and VMM, it is necessary for the integrator to specify which methodology to use as the primary phasing and messaging manager (see 5.12). Typically, an integrator who chooses to include VMM IP in a primarily-OVM environment would set the `OVM_ON_TOP` compiler directive while the integrator including OVM IP in a primarily-VMM environment would set the `VMM_ON_TOP` compiler directive. These directives shall be set via a tool-specific command-line argument.

The various practices defined in this chapter arise from the fundamental differences in how components in each library establish and conduct communication at the transaction level. These differences include, but are not limited to, the following.

— In OVM, communication occurs primarily through TLM[5] `ports` and `exports`, whose semantics are defined by the interface type used. Communication is established between components by con-

[5]In the context of OVM ports and exports, TLM refers not to the generic term, but to the TLM 1.0 interfaces developed and standardized by the Open SystemC Initiative (OSCI); see [B1].

necting a component's port to an interface and transaction-type compatible export in the target component.

— In VMM, communication is primarily conducted through shared `vmm_channel`s, `vmm_data` notifications, and callbacks. For `vmm_channel`s, the execution semantic is defined by matching the completion model expected by the producer with a compatible completion model provided by the consumer. For notifications and callbacks, the delivery mechanism is unique to VMM.

The role of the various adapters is to provide a bridge between the different OVM and VMM interfaces and execution semantics. In some cases, such as OVM sequences and VMM scenarios, the collaborations with other classes and mechanisms in the native library are so varied and application-specific that the implementation of the corresponding practice can not be aided by an adapter. In these cases, the practice consists of a description of how one might go about integrating the foreign VIP using the APIs and mechanisms of the foreign library.

In addition to using different transaction-level communication mechanisms, each methodology uses a different type to describe the same transaction. OVM transaction descriptors are based on the `ovm_transaction` or `ovm_sequence_item` class, whereas the VMM transaction descriptors are based on the `vmm_data` class. Thus, each practice involves translation of transactions in the source domain to the equivalent transaction in the target domain. Because each transaction type is application-specific, the integrator needs to write all conversion routines using the *Data conversion* practice (see 5.5).

5.1 OVM-on-top phase synchronization

5.1.1 Practice name

OVM-on-top phase synchronization

5.1.1.1 Also known as

Phase synchronization.

5.1.1.2 Related practices

VMM-on-top phase synchronization (see 5.2) and *Meta-composition* (see 5.3).

5.1.2 Intent

Provides a default mapping and coordination between OVM and VMM execution phases.

5.1.3 Applicability

This practice is required to ensure VMM environments, when instantiated in OVM components, are properly coordinated so the overall environment is configurable and properly built before any component begins executing, all components shut down completely, and all result-gathering, reporting, and cleanup occur only after component execution has completed.

To integrate sub environments and components, see 5.3.

5.1.3.1 Motivation

OVM and VMM support the basic concept of "phases," but each has its own specific set of phase methods. To minimize the impact on an OVM parent instantiating a VMM child, VMM VIP needs to be wrapped in an OVM component (to hide the VMM details).

5.1.3.2 Consequences

This practice automatically integrates VMM environments into the OVM phase controller mechanism, while providing an OVM-centric view of the component. All VMM environments are phased properly relative to each other. As a side effect of this practice, the VMM environment and its subcomponents become integrated into the OVM named-component hierarchy.

5.1.4 Structure

The overall structure for this practice is based on the following diagrams, prototype, and participants.

5.1.4.1 Class diagram

The class diagram for this practice is shown in Figure 3.

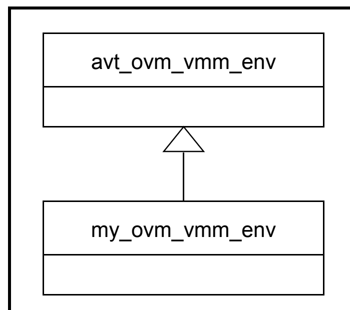

Figure 3—User-defined avt_ovm_vmm_env extension

5.1.4.2 Declaration prototype

This practice uses the following declaration prototypes.

```
class avt_ovm_vmm_env #(type ENV=vmm_env) extends avt_ovm_vmm_env_base;

class avt_ovm_vmm_env_named #(type ENV=vmm_env) extends avt_ovm_vmm_env_base;
```

The **avt_ovm_vmm_env** class (see 6.4) is used to wrap `vmm_env` subtypes that do not have a `name` argument as a part of their constructor, while the **avt_ovm_vmm_env_named** class is used to wrap any `vmm_env`'s with names.

In both cases, the `ENV` parameter specifies the type of the underlying `vmm_env`. If an instance of an environment of this type is not provided in the constructor, the **avt_ovm_vmm_env** creates one.

5.1.4.3 Interaction diagrams

The interaction diagrams for this practice are shown in Figure 4.

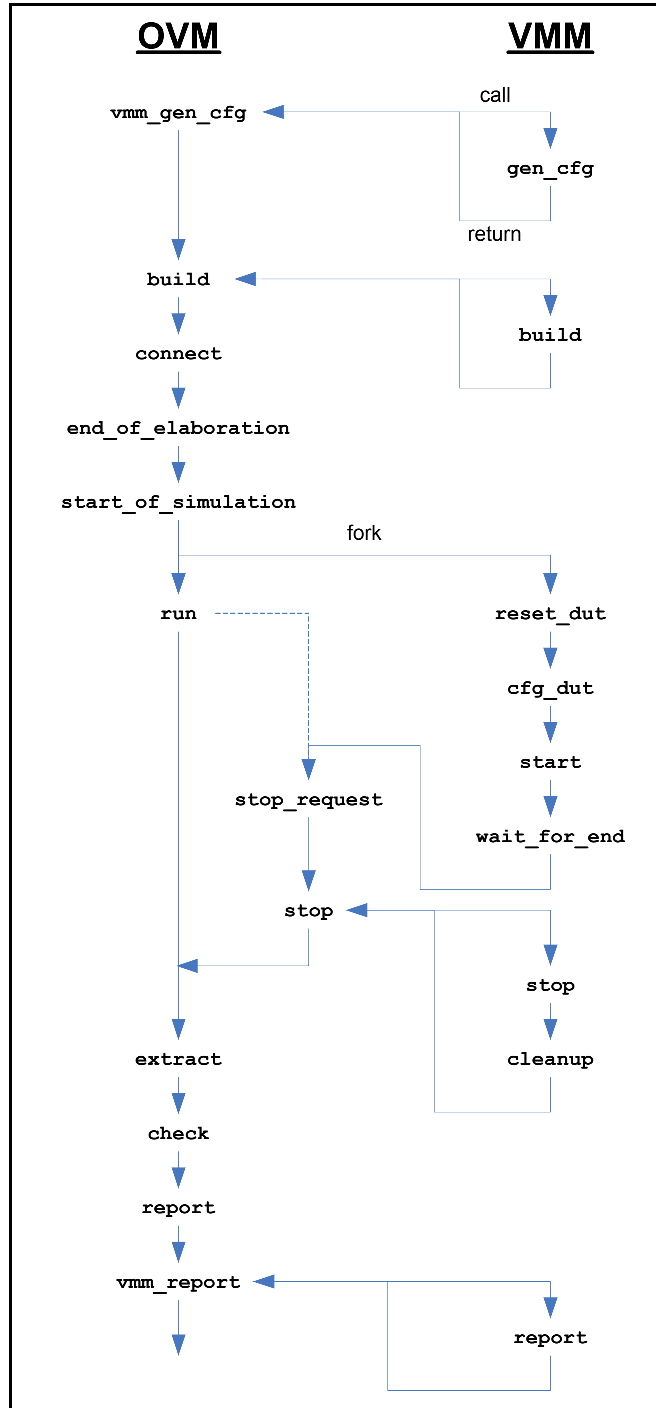

Figure 4—OVM-on-top phasing management

5.1.4.4 Participants

The integrator wraps the `vmm_env` (extension) by instantiating the appropriate wrapper component and specifying the `vmm_env` subtype as a parameter. The wrapper automatically creates an instance of the specified `vmm_env` subtype as a child of the wrapper. Which wrapper to use depends on whether an explicit

name is supplied to the vmm_env's constructor. If a name constructor argument is required, **avt_ovm_vmm_env_named** is used (see 6.4); otherwise, **avt_ovm_vmm_env** is used.

5.1.5 Collaboration

To accommodate the calling of the vmm_env phase methods in the proper order, the **avt_ovm_vmm_env** class (see 6.4) declares two new phases that are added to the OVM phase list. The **vmm_gen_cfg** phase (see 6.4.3.2) is inserted before build and it calls the vmm_env's gen_cfg() method, which shall be called before the vmm_env's build() method. From that point on, the **avt_ovm_vmm_env** is treated as any other OVM component, with its phase methods being called automatically in the proper order. Similarly, the vmm_report phase is inserted at the end of the OVM phase list to allow the VMM report task to be called.

The implementation of the **avt_ovm_vmm_env**'s run() method, which gets called in parallel with all other OVM components' run() methods, calls the underlying vmm_env's reset_dut(), cfg_dut(), start(), and wait_for_end() methods sequentially. When wait_for_end() returns, the run() method sets the **ok_to_stop** bit (see 6.4.4.1), which the ovm_env parent component (or any other OVM component, such as ovm_env or ovm_test) may use to determine that the vmm_env has completed its operation. The OVM parent may then call stop_request() to ensure all other OVM components have completed their run() methods.

Invocation of stop_request() causes the **avt_ovm_vmm_env**'s stop() method to be called, which calls the underlying vmm_env's stop() and cleanup() methods sequentially, in parallel with other OVM components executing their stop() methods (assuming they are enabled). Thus, when all OVM components, including the **avt_ovm_vmm_env**, return from their stop() methods, the OVM run() phase ends and the extract(), check(), and report() phases occur.

The vmm_report() phase gets executed after OVM's report() phase and causes the **avt_ovm_vmm_env** to call the env.report() method of the underlying vmm_env.

5.1.6 Implementation

To implement the practice, the integrator simply instantiates an **avt_ovm_vmm_env** or **avt_ovm_vmm_env_named**, as appropriate (see 6.4) in the ovm_env (or another parent ovm_component) and specifies the vmm_env type as a parameter. This effectively creates a wrapper around the vmm_env. The wrapper's parent then builds and optionally configures the wrapper as it would any other component. In the connect() method, the parent may optionally register one or more callback objects with transactors in the vmm_env. It may also choose to connect other OVM components (via the appropriate adapter) to specific vmm_channels in the vmm_env. When using the **avt_ovm_vmm_env** directly, the parent class necessarily includes VMM-specific code to enhance or otherwise modify the behavior of the underlying vmm_env.

If VMM sub environments and components (i.e., vmm_xactor) are integrated directly into an OVM parent, their methods are called directly from within the appropriate OVM phase method of the parent component.

The wrapper automatically calls the appropriate phase methods of vmm_env at the appropriate point relative to the others to ensure proper operation. Ideally, this alignment should be transparent to the user.

The integrator may also choose to extend the **avt_ovm_vmm_env** to encapsulate the VMM-specific extensions inside the wrapper—making the wrapper appear simply as an OVM component to its parent. Typically, the wrapper includes ports/exports and/or configuration "hooks" or other mechanisms to hide these details from the OVM parent. Using a customized wrapper also requires the integrator to call super.*method*() in wrapper.*method*() to ensure the underlying vmm_env.*method*() gets

called at the proper time. This also allows the integrator to perform additional operations before and after the execution of the vmm_env's methods.

If the integrator wishes to include additional components, such as communication adapters, alongside the vmm_env in the **avt_ovm_vmm_env** wrapper, the **avt_ovm_vmm_env** needs to be extended to instantiate the other components in build(), which is virtual. The new implementation of build() shall call super.build() to allow the underlying vmm_env's build() method to get called. Once super.build() returns, the underlying vmm_env is completely built.

5.1.7 Sample code

This illustrates how to implement this practice.

```
class vmm_env_ext extends 'VMM_ENV;
    ...
endclass

class ovm_comp_ext extends ovm_component;
    ...
endclass

class wrapped_vmm_env extends avt_ovm_vmm_env #(vmm_env_ext);
    `ovm_component_utils(wrapped_vmm_env)

    function new (string name, ovm_component parent=null);
        super.new(name,parent);
    endfunction
endclass

class my_ovm_env extends ovm_comp_ext;
    `ovm_component_utils(my_ovm_env)

    wrapped_vmm_env subenv;

    function new (string name, ovm_component parent=null);
        super.new(name,parent);
    endfunction

    virtual function void build();
        subenv = new("vmm_env",this);
    endfunction

endclass

module top;
    initial run_test("my_ovm_env");
endmodule
```

Notice when my_ovm_env constructs the **avt_ovm_vmm_env** in its build() method, the OVM phasing mechanism automatically executes its phases up to (but not yet including) **build**. Thus, the underlying vmm_env's gen_cfg method is called automatically at this time.

5.2 VMM-on-top phase synchronization

5.2.1 Practice name

VMM-on-top phase synchronization

5.2.1.1 Also known as

Phase synchronization.

5.2.1.2 Related practices

OVM-on-top phase synchronization (see 5.1) and *Meta-composition* (see 5.3).

5.2.2 Intent

Provides a default mapping and coordination between VMM and OVM execution phases.

5.2.3 Applicability

This practice is required to ensure OVM components (including `ovm_component`, `ovm_env`, `ovm_test`, and their extensions), when instantiated in VMM environments and/or components, have their phase methods called at the correct time relative to the execution of the top-level VMM environment phase methods.

5.2.3.1 Motivation

OVM and VMM support the basic concept of "phases," but each has its own specific set of phase methods. To present a single base class library view to the user, OVM VIP is ultimately instantiated in an extension of the `vmm_env` base class. The extension provides a set of virtual methods that enable the user to execute OVM phases in the proper order.

5.2.3.2 Consequences

This practice automatically integrates OVM components into a VMM environment. Just as the `vmm_env` is responsible for calling the appropriate methods of VMM sub environments and transactors in `build()`, `start()`, etc., so it shall also call specific methods to ensure the OVM components are built appropriately. Since the `vmm_env::reset_dut()` task is the first VMM phase that may interact with the DUT, execution of the OVM `run()` phase happens automatically in parallel with the execution of the VMM `reset_dut()` task.

5.2.4 Structure

The overall structure for this practice is based on the following diagrams, prototype, and participants.

5.2.4.1 Class diagram

The class diagram for this practice is shown in Figure 5.

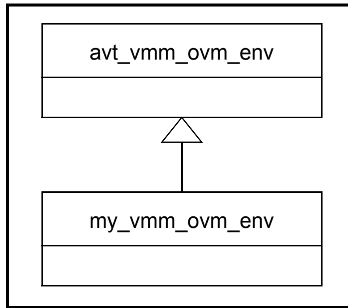

Figure 5—User-defined avt_vmm_ovm_env extension

5.2.4.2 Declaration prototype

This practice uses the following declaration prototype.

```
class avt_vmm_ovm_env extends `AVT_VMM_OVM_ENV_BASE;
```

5.2.4.3 Interaction diagrams

The interaction diagrams for this practice are shown in Figure 6.

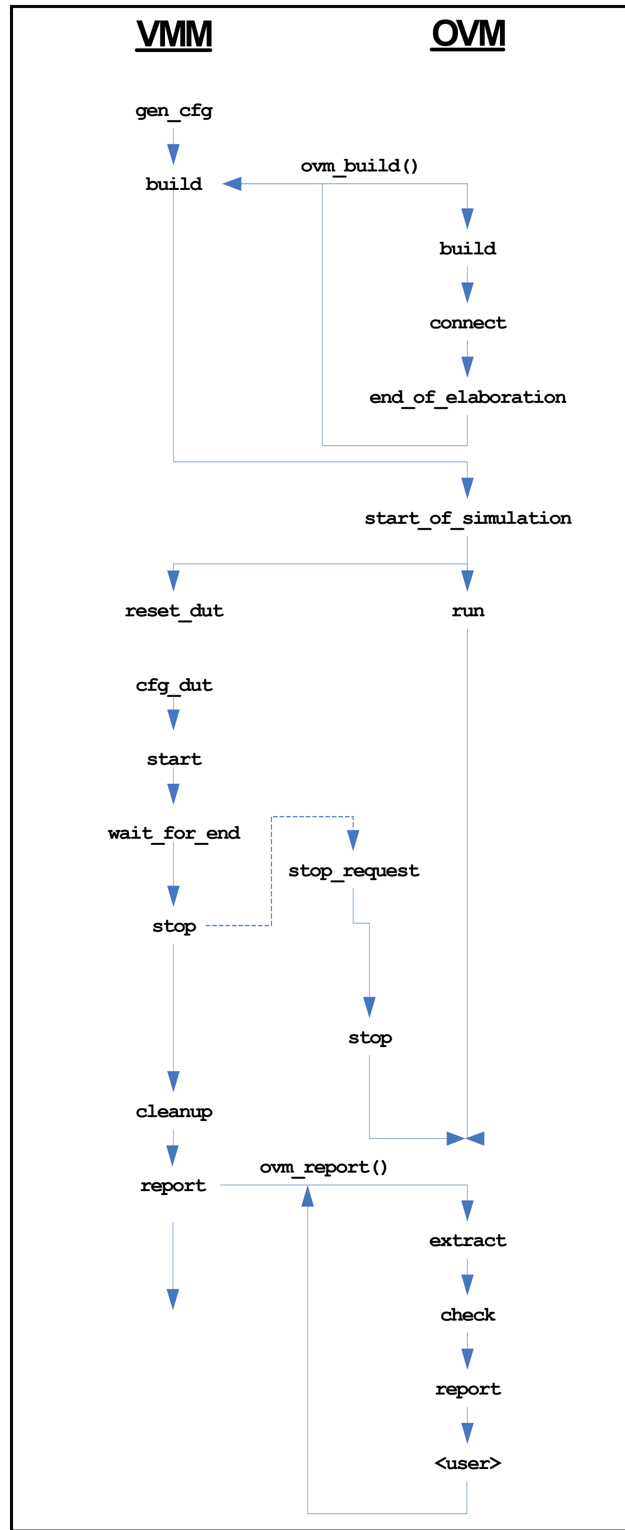

Figure 6—VMM-on-top phasing management

5.2.4.4 Participants

The integrator incorporates OVM IP into a VMM environment by instantiating the IP in the `avt_vmm_ovm_env::build()` method, or in a `vmm_subenv` or `vmm_xactor`, by instantiating the IP in the constructor. The instantiation may be done either directly by calling `new()` or by creating the OVM component via the OVM factory. Since the OVM separates the "build" operation across multiple phases, it is necessary for the VMM `build()` method to call `ovm_build()` (after the OVM IP is instantiated) to cause the OVM phase manager to execute the OVM `build()`, `connect()`, and `end_of_elaboration()` phases to completely build the OVM IP.

In either case, the VMM parent shall include the `` `ovm_build`` macro, which declares an instance-specific version of the `ovm_build()` method to ensure the OVM phases do not get called more than once.

To facilitate communication between VMM IP and OVM IP, existing interconnected OVM IP components shall be wrapped in a single OVM wrapper component that includes as constructor arguments any `vmm_channel`s necessary to connect to VMM transactors. Inside the wrapper, these channels are connected to the necessary communication adapters (see 5.6 and 5.7) that are then connected via OVM TLM guidelines to the underlying OVM IP. Thus, the wrapper contains additional constructor arguments beyond standard OVM components. This means the OVM wrapper may be created without using a factory, but the wrapper may itself use the factory to create its children.

5.2.5 Collaboration

The `vmm_env`'s phase methods are called from the VMM test either directly or via the `env.run()` method, which automatically calls the environment's phase methods.

— The `ovm_build()` method runs the OVM children through the `end_of_elaboration` phase, so `ovm_build()` shall be called before the top-level `avt_vmm_ovm_env` exits its `build()` method.

— Although the OVM IP may not be completely built, all communication with VMM IP is handled via channels, which shall be constructor arguments to the OVM wrapper.

— Since the `reset_dut()` method of the top-level VMM environment is the first method that entails communication with the DUT, the `avt_vmm_ovm_env::reset_dut()` method automatically spawns execution of the OVM `run()` method, allowing OVM components, which may be required to interact with the DUT, to be started.

— The `avt_vmm_ovm_env::stop()` method can call `ovm_top.global_stop_request()` to stop all OVM components' `run()` phase execution.

— In the `avt_vmm_ovm_env::report()` method, the OVM phase manager runs to completion, which executes the `extract()`, `check()`, and `report()` methods, along with any other user-defined phases that may have been added after the run phase. When a user-defined extension of `avt_vmm_ovm_env` calls `super.report()`, the OVM phases run to completion. When `super.report()` returns, all of the OVM phases are finished.

5.2.6 Implementation

To implement the practice, the integrator simply extends **avt_vmm_ovm_env** (see 6.5) to create the application-specific environment, according to VMM guidelines. In the `build()` method, the OVM IP wrapper is instantiated, with the necessary `vmm_channel`s connected in the constructor according to VMM guidelines and `ovm_build()` is called. Inside the wrapper, the `vmm_channel`s are connected to the necessary communication adapters, which are then connected to other OVM components in the wrapper's `connect()` method. The OVM IP may be configured via a configuration object constructor argument (as with other VMM IP) or via the OVM `set/get_config`* methods. In the former case, the

wrapper may extract information from the constructor argument and use `set_config*` to configure its OVM children.

5.2.7 Sample code

This illustrates how to implement this practice.

```
class vmm_env_with_ovm extends avt_vmm_ovm_env;
   ...
   function new (string name);
      super.new(name);
   endfunction

   `ovm_build

   ovm_comp_ext ovm_child,ovm_child2; // Instantiate OVM children

   virtual function void build();
      super.build();
      ovm_child = ovm_comp_ext::type_id::create({log.get_name(),
                                                ".ovm_child"}, null);
      ovm_child2 = new({log.get_name(),".ovm_child2"});
      ovm_build();
   endfunction

endclass

module example_03_vmm_on_top;

   vmm_env_with_ovm e = new("vmm_top");

   initial begin
      e.gen_cfg();
      // Manually modify the config object
      e.run();
   end
endmodule
```

The user can call individual phase methods of the `vmm_env` if desired, and/or may call the `run()` method to execute the remaining phases automatically.

5.3 Meta-composition

5.3.1 Practice name

Meta-composition

5.3.1.1 Also known as

Hierarchical wrapping.

5.3.1.2 Related practices

Used by *OVM-on-top phase synchronization* (see 5.1) and *VMM-on-top phase synchronization* (see 5.2). Also related to the *VIP configuration* practice (5.4).

5.3.2 Intent

This practice shows how to incorporate non-leaf-cell components from different methodologies into a single assembly.

5.3.2.1 Motivation

VIP seldom comes as a single monolithic component, but it often incorporates hierarchy. Therefore, successful interoperability requires that hierarchical components can be composed of other components which are themselves hierarchical. Any hierarchical component may include subcomponents that are implemented in either methodology or a combination thereof.

5.3.2.2 Consequences

None.

5.3.3 Applicability

This practice is used whenever two hierarchical components from different base classes or a mix of base classes need to be connected, with the following exceptions.

— When integrating `vmm_env`s into OVM components (including `ovm_env`), use the *OVM-on-top phase synchronization* practice (see 5.1).

— When integrating OVM components directly into a `vmm_env`, use the *VMM-on-top phase synchronization* practice (see 5.2).

5.3.4 Structure

The overall structure for this practice is based on the following diagrams and participants.

5.3.4.1 Declaration prototype

This practice does not require an explicit new class object to be declared.

5.3.4.2 Interaction diagrams

The interaction diagrams for this practice are shown in Figure 7 and Figure 8.

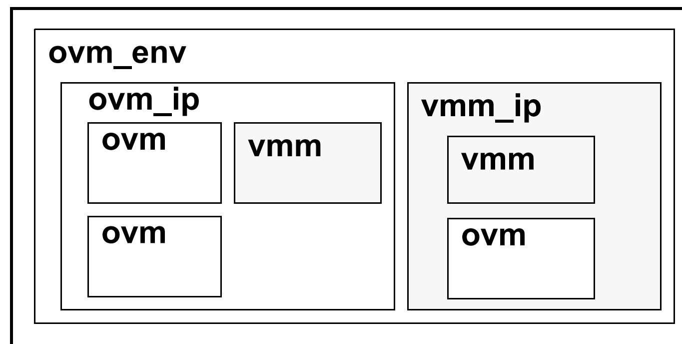

Figure 7—Hierarchical compositions in an OVM top-level environment

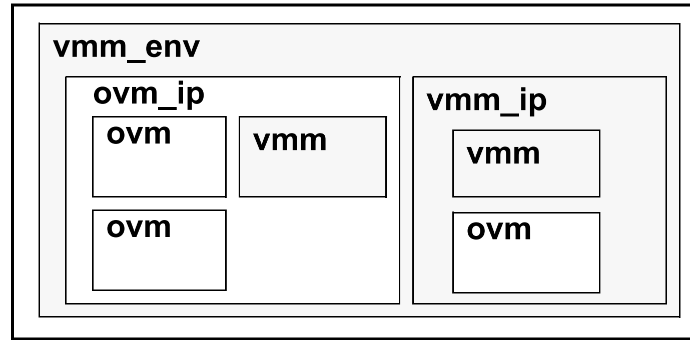

Figure 8—Hierarchical compositions in a VMM top-level environment

In Figure 7 and Figure 8, the *vmm_ip* represents a vmm_subenv or a vmm_xactor. The techniques used to encapsulate a mix of OVM and VMM subcomponents is independent of the enclosing VMM type, with the exception of standard VMM differences in how the container classes are themselves instantiated and configured. Since all containers in OVM are extensions of ovm_component, the specific type of component being used may be one of ovm_component, ovm_env, ovm_test, or a user-defined extension of these, or any of the other ovm_component extensions in OVM.

5.3.4.3 Participants

This practice includes the use of any OVM and VMM components used to model each composition, along with the top-level container.

5.3.5 Collaboration

The individual hierarchical assemblies may be connected arbitrarily within themselves according to the guidelines of the appropriate methodology. They may also be connected to each other, including hierarchically, using any of the adapters described in 5.6 and 5.7.

5.3.6 Implementation

OVM users can create an ovm_component extension, including ovm_env, to hold the assembly, while VMM users can create a vmm_xactor or vmm_subenv extension.

a) In an OVM container (ovm_ip in Figure 7 and Figure 8), the OVM and VMM subassemblies are instantiated as any other OVM component would be, in the container's build() method, and the connections between them are defined in the connect() method. The OVM container shall call the vmm_subenv or vmm_xactor methods directly during the appropriate OVM phase, as shown in Table 3.

b) In a VMM container (vmm_ip in the Figure 7 and Figure 8), the OVM and VMM subassemblies are instantiated and connected in the container's constructor, as any other VMM components would be.

In the VMM-on-top case, the top-level vmm_env calls the appropriate OVM methods via the ovm_build() call in vmm_env::build() [see 5.2.6]. Since VMM only allows a single vmm_env instance, an OVM container may only include instances of vmm_subenv or vmm_xactor, not vmm_env.

5.3.7 Sample code

The following code example demonstrates the composition of the hierarchical assembly shown in Figure 9.

Table 3—OVM phases calling VMM methods

VMM		OVM
Component	**Method**	**Phase method**
`vmm_subenv`	`configure()`	`build()` or `end_of_elaboration()`
	`start()`	`run()`
	`stop()`	`run()` or `stop()` [1]
	`cleanup()`	`stop()`
`vmm_xactor`	`start_xactor()`	`run()`
	`stop_xactor()` [2]	`run()` or `stop()` [3]

[1]If the OVM `enable_stop_interrupt` bit is set, `vmm_subenv::stop()` shall be called from the OVM parent's `stop()` method. Otherwise, `stop()` should be called from the parent's `run()` method.
[2]Calling `stop_xactor()` is optional and environment-dependent.
[3]If the OVM `enable_stop_interrupt` bit is set, `vmm_xactor::stop_xactor()` shall be called from the OVM parent's `stop()` method. Otherwise, `stop_xactor()` should be called from the parent's `run()` method.

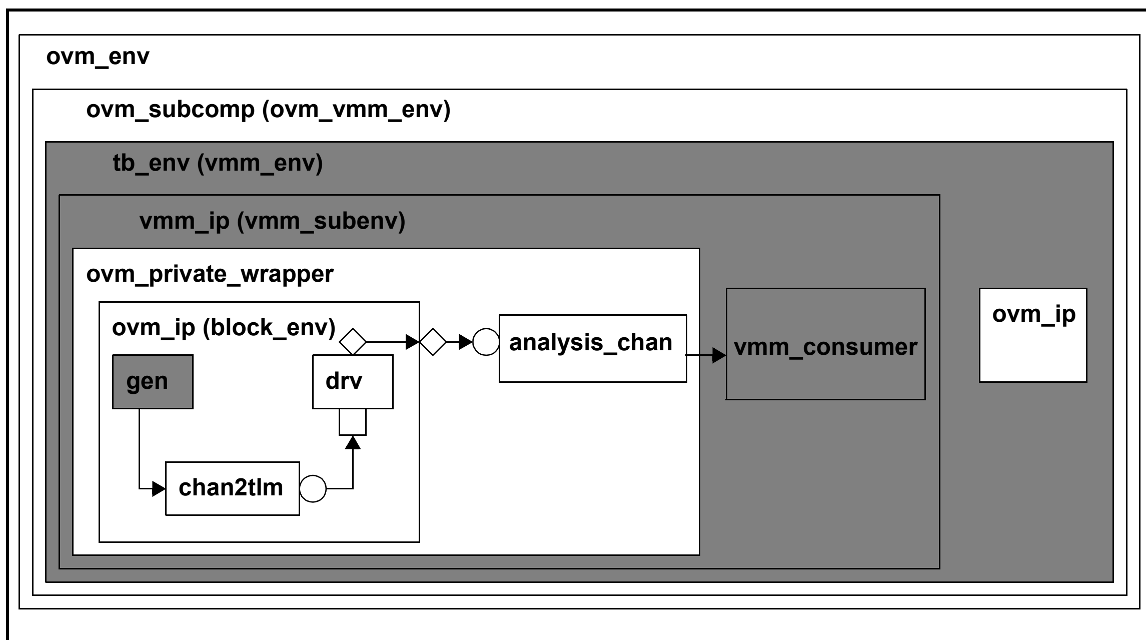

Figure 9—Hierarchical assembly mixing OVM and VMM

In the following example code, the `ovm_ip` block is extended from the block-level environment, which includes a mix of VMM and OVM components and an adapter (see 5.7). The `ovm_ip` extension adds an `analysis_port` that exposes the driver's `analysis_port` for connection to the `analysis_chan` adapter (see 5.8). The example shows that both OVM and VMM sub-components may be instantiated in a VMM environment, a VMM sub-environment, and an OVM component.

```
class block_env extends ovm_component;
   `ovm_component_utils(block_env)

   apb_rw_atomic_gen gen;
   apb_channel2tlm   adapt;
   ovm_driver_req    drv;

   function new (string name="block_env",ovm_component parent=null);
      super.new(name,parent);
   endfunction

   virtual function void build();
      gen   = new("gen", 0);
      adapt = new("adapt", this, gen.out_chan);
      drv   = new("drv", this);
      void'(get_config_int("max_trans",drv.max_trans));
   endfunction

   virtual function void connect();
      drv.seq_item_port.connect(adapt.seq_item_export);
   endfunction

   virtual task run();
      gen.start_xactor();
      gen.notify.wait_for(apb_rw_atomic_gen::DONE);
      ovm_top.stop_request();
   endtask
endclass

class ovm_ip extends block_env;

   `ovm_component_utils(ovm_ip)

   ovm_analysis_port #(ovm_apb_rw) ap;

   function new(string name, ovm_component parent=null);
      super.new(name,parent);
   endfunction

   function void build();
      super.build();
      ap = new ("ap",this);
   endfunction

   function void connect();
      super.connect();
      drv.ap.connect(ap);
   endfunction

   virtual task run();
      gen.start_xactor();
   endtask
endclass

class ovm_private_wrapper extends ovm_component;
   `ovm_component_utils(ovm_private_wrapper)
   ovm_ip               o_ip;
   apb_analysis_channel v_ap_adapter;
   vmm_channel_typed #(vmm_apb_rw) out_chan;
```

```
   function new(string name, ovm_component parent=null,
                vmm_channel_typed #(vmm_apb_rw) out_chan=null);
      super.new(name,parent);
      this.out_chan = out_chan;
   endfunction
   virtual function void build();
      o_ip = new("o_ip",this);
      v_ap_adapter = new("v_ap_adapter", this, out_chan);
      if (out_chan == null)
         out_chan = v_ap_adapter.chan;
   endfunction
   virtual function void connect();
      o_ip.ap.connect(v_ap_adapter.analysis_export);
   endfunction
endclass

class vmm_ip_cfg;
  int max_trans=0;
endclass

class vmm_ip extends vmm_subenv;

   ovm_private_wrapper    o_wrapper;
   vmm_consumer #(vmm_apb_rw) v_consumer;
   vmm_consensus end_vote;

   function new(string inst, vmm_ip_cfg cfg, vmm_consensus end_vote);
      super.new("vmm_ip", inst, end_vote);
      this.end_vote = end_vote;
      v_consumer = new({inst,".v_consumer"},0);
      o_wrapper  = new({inst,".o_wrapper"},,v_consumer.in_chan);
      v_consumer.stop_after_n_insts = cfg.max_trans;
      set_config_int({inst,".o_wrapper.o_ip"},"max_trans",cfg.max_trans);

      end_vote.register_notification(v_consumer.notify,v_consumer.DONE);
   endfunction

   task configure();
      super.configured();
   endtask

   virtual task start();
      super.start();
      v_consumer.start_xactor();
   endtask

   virtual task stop();
      super.stop();
   endtask

   virtual task cleanup();
      super.cleanup();
   endtask
endclass

class tb_env extends vmm_env;
   vmm_ip_cfg cfg;
   vmm_ip v_ip;
   ovm_ip o_ip;
```

```
    function new();
        super.new("mixed_tb_env");
    endfunction

    virtual function void gen_cfg();
        super.gen_cfg();
        cfg = new;
        // Could be randomized, but isn't
        cfg.max_trans = 10;
    endfunction

    virtual function void build();
        super.build();
        v_ip = new({log.get_name(),".v_ip"},cfg,end_vote);
        o_ip = new({log.get_name(),".env_o_ip"});
    endfunction

    virtual task cfg_dut();
        super.cfg_dut();
        v_ip.configure();
    endtask

    virtual task start();
        super.start();
        v_ip.start();
    endtask

    task wait_for_end();
        super.wait_for_end();
        end_vote.wait_for_consensus();
        global_stop_request();
    endtask
endclass

class ovm_subcomp extends avt_ovm_vmm_env #(tb_env);
    `ovm_component_utils(ovm_subcomp)

    function new (string name="ovm_subcomp", ovm_component parent=null);
        super.new(name,parent);
    endfunction
    virtual function void vmm_gen_cfg();
        ovm_object obj;
        super.vmm_gen_cfg();
        if (get_config_object("cfg",obj,0)) begin
            ovm_container #(vmm_ip_cfg) v_cfg;
            assert($cast(v_cfg,obj));
            env.cfg = v_cfg.obj;
            end
    endfunction
endclass

class ovm_env extends ovm_component;
    `ovm_component_utils(ovm_env)

    ovm_subcomp subcomp;

    function new (string name="ovm_env", ovm_component parent=null);
        super.new(name,parent);
    endfunction
```

```
    virtual function void build();
        super.build();
        subcomp = new("subcomp",this);
    endfunction
endclass
```

The example includes an OVM component (ovm_subcomp) instantiating a vmm_env (tb_env), as well as an OVM component (ovm_ip) instantiating a VMM component. The same template may be used for instantiating a vmm_subenv. It also shows a vmm_subenv (vmm_ip) instantiating an OVM component (ovm_private_wrapper). For an example of a vmm_env instantiating OVM components directly, see 5.2.

Notice the vmm_ip container instantiates the ovm_private_wrapper and the vmm_consumer in its build() method, passing the vmm_consumer's input channel to the ovm_private_wrapper's constructor, as if it were a standard VMM transactor. The build() method can also configure the vmm_consumer by setting its stop_after_n_insts parameter after the vmm_consumer has been constructed. See 5.4 for a more in-depth discussion of configuration.

To preserve the proper hierarchical naming of components, an OVM parent should set the instance name of its VMM children to be {get_full_name(),".",*child_name*}. This appends the *child_name* to the existing OVM hierarchical name of the OVM parent, thus giving the VMM child a consistent hierarchical name. Similarly, VMM parents that are derived from vmm_subenv or vmm_xactor, which themselves have a unique *instance* name, should set the instance name of OVM children to {*inst*, ".", *child_name*}. VMM parents that are derived from **avt_vmm_ovm_env** (see 6.5) should set the instance name of OVM children to {log.get_name(), ".", *child_name*}. As long as the children then follow the appropriate guidelines for naming their children, both OVM and VMM components have the proper hierarchical names.

The ovm_private_wrapper includes a vmm_channel as a constructor argument. Notice this restricts the component from being created via the factory, but this particular component does not need to be overridden. Other than that, ovm_private_wrapper is a standard ovm_component. It instantiates the OVM subcomponents in its build() method. Since the v_ap_adapter allocates its channel in its own constructor, the out_chan can similarly be assigned in build() after the adapter has been constructed. The integrated phasing mechanism then runs the connect() method of the wrapper, which makes the standard OVM port/export connections as necessary.

5.3.8 Printing the environment topology

For debugging purposes, the container print methods should be enhanced to also call the contained foreign component print method. For example an OVM component do_print() can be extended to call the psdisplay of a VMM transactor. This fits nicely with both the OVM and the VMM field automation macros.

a) OVM

```
function void do_print(ovm_printer printer);
    ...
    printer.print_generic("xactor", "simple_vmm_xactor", -1,
                          xactor.psdisplay());
  endfunction
```

b) VMM

```
function string psdisplay(string prefix="");
    ...
    psdisplay = {psdisplay, "\n", prefix, ovm_comp1.sprint()};
    ...
endfunction
```

```
//or using short-hand macros:

'vmm_data_member_begin(...);
   ...
   'vmm_data_member_user_defined(ovm_comp1, DO_ALL);
   ...
'vmm_data_member_end(...);
...
function bit do_ovm_comp1(...);
   do_ovm_comp1 = 1;
   case (do_what)
   ...
   DO_PRINT: begin
      image = {image . "\n", prefix, ovm_comp1.sprint()};
   end
   ...
   endcase
endfunction
```

5.4 VIP configuration

5.4.1 Practice name

Procedural VIP configuration or Randomized configuration

5.4.1.1 Also known as

N/A.

5.4.1.2 Related practices

Used with *OVM-on-top phase synchronization* (see 5.1), *VMM-on-top phase synchronization* (see 5.2), and *Meta-composition* (see 5.3).

5.4.2 Intent

A substantial part of the value of a reusable component is its ability to be reused in multiple contexts, based on information passed to it from the test and/or the container.

5.4.2.1 Motivation

Both OVM and VMM provide procedural mechanisms for passing such configuration information into subcomponents and these mechanisms need to be preserved in an interoperability environment.

5.4.2.2 Consequences

None.

5.4.3 Applicability

This practice is used to configure OVM components from a VMM container and vice-versa.

5.4.4 Structure

This practice uses methods that already exist in each base library. There are no new components or adapters required to implement this practice.

5.4.5 Collaboration

This practice uses OVM (global) `set_config_[int,string,object]()` methods for setting configuration information for OVM children instantiated by VMM. Once set, the `config` information is retrieved by the OVM child using the standard OVM `get_config_*()` methods. VMM components are configured via constructor arguments, so the OVM parent passes the configuration information when calling `new()`.

5.4.6 Implementation

An OVM container creates a configuration object of the appropriate type to pass to a particular VMM child in its constructor. Once created, the configuration object may be randomized using standard SystemVerilog randomization, including constraints, prior to being passed to the VMM child.

A VMM container calls `set_config_*()` before `build()` is called on its OVM children. If the VMM container is a `vmm_env`, this would occur in `build()`. If the VMM container is a `vmm_xactor` or `vmm_subenv`, it would be done in the constructor. The OVM child subsequently uses `get_config_*()` to retrieve the configuration information. In OVM, the configurable parameters may be declared using the `` `ovm_field* `` macros, in which case the get_config gets called automatically.

In either case, the child may randomize the desired configuration parameter if it is not set by the container. In VMM, this would mean the constructor argument is `NULL`. In OVM, it would mean the `get_config_*()` call returns `0`.

To support reconfiguration at run-time, it is up to the container component to apply the previously mentioned configuration methods at the appropriate time.

5.4.7 Sample code

This illustrates how to implement this practice.

a) OVM parent

```
class my_config extends ovm_object;
   rand int num_slaves = 2;
   rand other_obj obj;
   constraint c1 {num_slaves > 0; num_slaves < 10;}
endclass

class my_ovm_parent extends ovm_component;
   vmm_ip v_ip;
   my_config cfg;
   void function build();
      if(!cfg.randomize())
         ovm_error(get_full_name(), "Randomization failed");
      v_ip = new(...,cfg,...);
      v_ip.stop_after_n_insts = 5;
      ...
   endfunction
endclass;
```

b) VMM parent

```
class my_config;
   rand int num_slaves = 2;
   rand other_obj obj;
   constraint c1 {num_slaves > 0; num_slaves < 10;}
endclass

class my_vmm_parent extends vmm_xactor;
   ovm_ip o_ip;
   my_config cfg;
   void function new();
      if(!cfg.randomize())
         'vmm_error(log, "Randomization failed");
      set_config_object("o_ip", "cfgobj", cfg);
      set_config_int("o_ip", "stop_after_n_insts", 5);
      o_ip = ovm_ip::type_id::create("o_ip", this);
      ...
   endfunction
endclass
```

5.5 Data conversion

5.5.1 Practice name

Data conversion

5.5.1.1 Also known as

N/A.

5.5.1.2 Related practices

TLM to channel (5.6) and *Channel to TLM* (5.7).

5.5.2 Intent

Allows a transaction to be converted from VMM to OVM and vice-versa.

5.5.2.1 Motivation

VIPs implemented independently are likely to use different SystemVerilog types to represent the same transaction. When implemented in different methodologies, they further use distinct base types to model transaction descriptors. A transaction descriptor originating in one library shall thus be converted to the equivalent transaction descriptor in the destination library for to protocol-compatible VIPs to communicate with each other.

5.5.2.2 Consequences

It is necessary for the integrator to implement the type conversion through an explicit mapping between data fields contained by the two respective data objects. This does not allow declarative members, such as constraints or user-defined methods, to be converted: functionally-equivalent declarations need to already exists in the destination type.

The conversion process creates a separate transaction descriptor instance with identical content. Any subsequent changes to either transaction descriptor are not reflected in the other.

5.5.3 Applicability

This practice shall be used whenever a transaction descriptor is exchanged between two components, each implemented in different methodologies.

In some cases, the methods and properties of a transaction type may not have an equivalent translation in the other library. If those methods and properties are critical for the correct operation of the underlying VIP, interoperability between OVM and VMM components may not be feasible without modifying at least one of the transaction types to implement the necessary functionality.

5.5.4 Structure

The overall structure for this practice is based on the following diagrams, prototype, and participants.

5.5.4.1 Declaration prototype

The integrator declares OVM-to-VMM and VMM-to-OVM converter types (see 6.2) and passes these in as parameters to the appropriate adapter.

```
class avt_converter #(type IN=int, OUT=int);
  static function OUT convert(IN from, OUT to=null);
endclass
```

5.5.4.2 Interaction diagrams

See 5.6.4.3.

5.5.4.3 Participants

This practice defines the converter objects that get passed as parameters to the various adapter component(s) to convert input transaction descriptors from one methodology into output descriptors in another methodology.

It requires the existence of transaction descriptors in each library that can represent the same transaction.

5.5.5 Collaboration

None.

5.5.6 Implementation

To implement the practice, the integrator defines a converter class for each possible conversion operation. This class does not need to be an extension of any particular type, but shall follow the prototype exactly. The implementation of the static `convert()` method does the actual conversion.

The details of the conversion process are descriptor-specific. It may be implemented by explicitly assigning the relevant data members or by packing one descriptor into a byte stream and then unpacking the byte stream into the other descriptor. It may also be necessary to map the `rand_mode` state of data members and the `constraint_mode` state of constraint blocks from the original transaction descriptor to the equivalent data member or constraint block of the destination descriptor.

The `from` argument is a required input argument. The `to` argument is optional and, if not *null*, provides a reference to the destination descriptor and shall be returned by the function. If the `to` argument is *null*, the `convert()` method allocates a new `OUT`-type object and returns that.

In the OVM-to-VMM case, the `convert()` method takes an `ovm_transaction` extension as an argument and returns the corresponding `vmm_data` extension. In the VMM-to-OVM case, the `vmm_data` extension is the input argument and the method returns an `ovm_transaction` extension.

5.5.7 Sample code

This illustrates how to implement this practice.

```
class apb_rw_convert_ovm2vmm;
  static function vmm_apb_rw convert(ovm_apb_rw from,
                                     vmm_apb_rw to=null);
    if (to == null)
      convert = new;
    else
      convert = to;
    case (from.cmd)
      ovm_apb_rw::RD : convert.kind = vmm_apb_rw::READ;
      ovm_apb_rw::WR : convert.kind = vmm_apb_rw::WRITE;
    endcase
    convert.addr = from.addr;
    convert.data = from.data;
    convert.data_id = from.get_transaction_id();
    convert.scenario_id = from.get_sequence_id();
  endfunction
endclass
```

The `avt_analysis_channel` class (see 5.8) uses the converter class, as shown:

```
class avt_analysis_channel#(type OVM=int,
                           VMM=int,
                           OVM2VMM=avt_converter #(OVM,VMM),
                           VMM2OVM=avt_converter #(VMM,OVM))
                               extends ovm_component;
...
    function void write(OVM ovm_t);
        VMM vmm_t;
        if (ovm_t == null)
            return;
        vmm_t = OVM2VMM::convert(ovm_t);
        chan.sneak(vmm_t);
    endfunction
...
endclass
```

The implementation of `write()` in the analysis_channel adapter calls the `convert()` method of the type parameter passed in. To hide the details from the user, the `convert()` method is called in clone mode so a new VMM type object is created. The integrator specifies the one-to-one mapping from OVM to VMM data fields. A separate converter object is used for the reverse operation. This allows the same conversion mechanism to be used between arbitrary correlated types, regardless of the underlying library being used.

Using a static method allows `convert()` to be called via the `type` parameter without requiring the adapter to instantiate (and allocate) an instance of the converter.

Only the data fields are passed from one library to the other. The `convert()` method(s) simply map between fields. This does not allow, for example, constraints to be passed across the library boundary. Enacting a methodology in which no additional information needs to be communicated across the boundary eliminates dependencies between the components in one library and data in the other.

5.6 TLM to channel

5.6.1 Practice name

TLM to channel

5.6.1.1 Also known as

OVM producer to VMM consumer; Channel adapter.

5.6.1.2 Related practices

Meta-composition (5.3), *Data conversion* (5.5), and *Channel to TLM* (5.7).

5.6.2 Intent

This practice enables any OVM producer component communicating via TLM ports and exports to communicate at the transaction level with any VMM consumer communicating via a `vmm_channel`. Per OSCI TLM convention (see [B1]), such a component is referred to as a "bridge." In this document, it is referred to as an "adapter."

5.6.2.1 Motivation

When integrating OVM and VMM components, each component has established semantics that define the communication and the mechanics of establishing the communication path. This practice and the supporting adapter is designed to present the OVM interface on the "OVM side" and the VMM interface on the "VMM side," thus allowing the two components to interoperate.

For OVM components, the communication semantics of a particular connection are fully defined by the set of ports and/or exports used to establish the connection. In VMM, the `vmm_channel` API supports multiple semantics depending on the implementation of the consumer. The **avt_tlm2channel** adapter (see 5.6.4) allows connection to any established VMM consumer and properly converts the desired semantics of the `vmm_channel` connection(s) used by it into the corresponding TLM semantics required by the OVM producer, regardless of which TLM port/export it uses.

5.6.2.2 Consequences

Integrators need to provide an implementation of the *Data conversion* (5.5) practice for converting OVM requests to VMM requests and, when separate responses are involved, converting VMM responses to OVM responses.

Integrators of VMM-on-top environments need to apply the *Meta-composition* (5.3) practice to define a simple OVM component that contains (wraps) all OVM VIP and adapters used in any given VMM scope. This enables port connections between OVM components to be made in the connect phase of the OVM wrapper.

5.6.3 Applicability

This practice is useful for connecting an OVM producer to a VMM consumer that uses the `vmm_channel` to communicate.

5.6.4 Structure

The overall structure for this practice is based on the following diagrams, prototype, and participants.

5.6.4.1 Class diagram

The class diagram for this practice is shown in Figure 10.

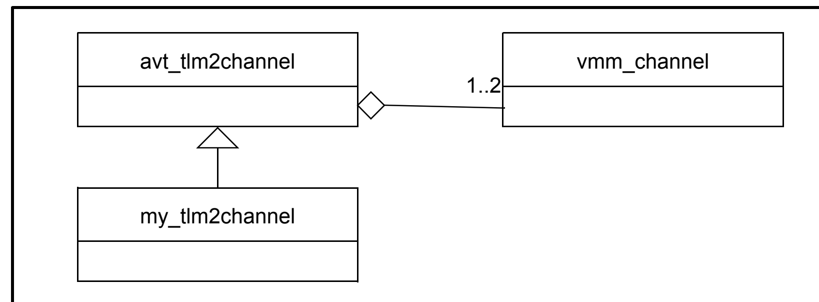

Figure 10—avt_tlm2channel class extension

5.6.4.2 Declaration prototype

This practice uses the following declaration prototype.

```
class avt_tlm2channel
             #(type OVM_REQ     = int,
                    VMM_REQ     = int,
                    OVM2VMM_REQ = int,
                    VMM_RSP     = VMM_REQ,
                    OVM_RSP     = OVM_REQ,
                    VMM2OVM_RSP = avt_converter #(VMM_RSP, OVM_RSP))
                                             extends ovm_component;
```

See 6.1 for a description of each `type` parameter. The default type values for the first three parameters are declared as `int` in the prototype to allow the compiler to report a type mismatch if these parameters are not explicitly set to a meaningful type. The user would declare a specialization of the adapter as

```
typedef avt_tlm2channel
           #(ovm_apb_rw, vmm_apb_rw,
             apb_rw_convert_ovm2vmm,
             vmm_apb_rw,ovm_apb_rw,
             apb_rw_convert_vmm2ovm) apb_tlm2channel;
```

5.6.4.3 Interaction diagrams

The interaction diagrams for this practice are shown in Figure 11. A single **avt_tlm2channel** adapter instance (see 6.6) enables connection of one of the OVM producer types on the left to one of the VMM consumer types on the right, subject to the collaboration requirements and limitations set forth in Table 4.

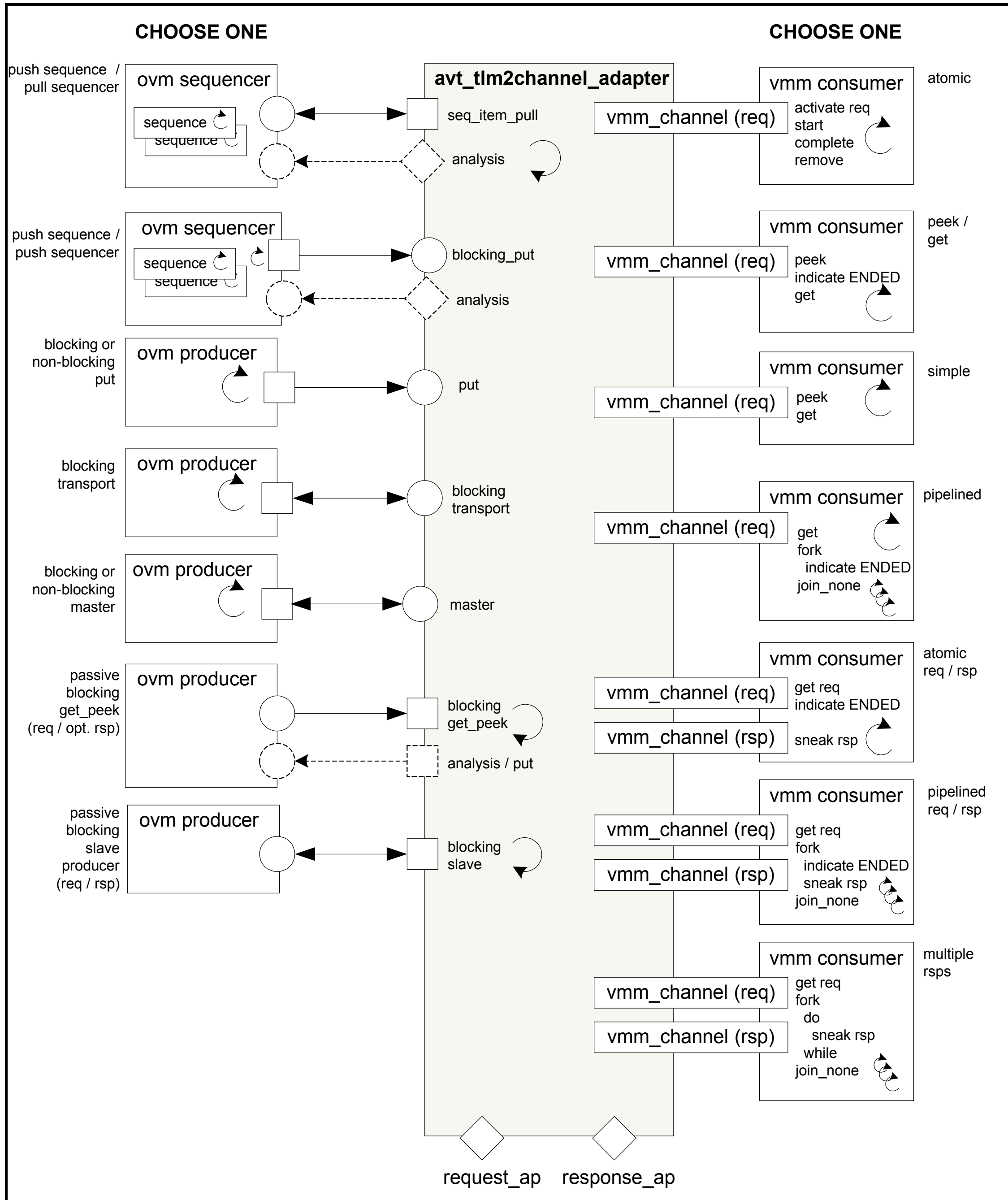

Figure 11—avt_tlm2channel interaction

5.6.4.4 Participants

Each application of this practice involves the following participants.

— An instance of one and only one of the `ovm producer` types, including `ovm_sequencer`, shown in Figure 11, subject to the collaboration requirements in Table 4.

— An instance of one and only one of the `vmm consumer` types shown in Figure 11, subject to the collaboration requirements in Table 4. If the OVM producer requires a return response, the VMM

consumer needs to also provide a response. In cases where the VMM consumer annotates the original request with the response rather than sending the response via a separate channel, this condition is met if the adapter's **rsp_is_req** bit is set (see 6.7.6.1). If the OVM producer does not take a response, it can still be paired with a VMM consumer that provides responses in a separate response channel provided that channel is sunk, e.g., `rsp_chan.sink()`.

- An instance of a request `vmm_channel`. Typically, the VMM consumer allocates its own channel, which is subsequently passed to the **avt_tlm2channel** adapter's constructor (see 6.6).
- An instance of a response `vmm_channel` if the selected VMM consumer uses a separate channel for responses. Typically, the VMM consumer allocates its own channel, which is subsequently passed to the **avt_tlm2channel** adapter's constructor.
- An instance of the **avt_tlm2channel** adapter whose `type` parameters are matched to the types used by the OVM producer and VMM consumer.

5.6.5 Collaboration

An active OVM producer injects transactions into the **avt_tlm2channel** (see 6.6) adapter via a TLM port, while a passive OVM producer responds to requests for new transactions via a TLM export. In the latter case, the adapter forks processes that make requests for new transactions and delivers their responses. The adapter automatically handles the completion of the transaction, regardless of how it is signalled by the VMM consumer.

When the adapter receives an OVM request transaction, it employs the *user-defined converter* to create an equivalent VMM transaction. This new VMM transaction is then `put` into the request `vmm_channel`.

The VMM consumer uses `peek`, `get`, or the active slot to retrieve transactions from the request `vmm_channel`. Depending on its completion model, it indicates transaction completion in one of several ways, each of which shall be accommodated by the adapter.

a) The VMM consumer using only the request channel may annotate the response in the original request, indicate the `ENDED` status in the original request, then remove the transaction from the channel using `get` or `remove`. The call to `get` or `remove` tells the adapter a response is ready to be sent back to the OVM producer in the form of a modified request.

1) Atomic

```
forever begin
   vmm_apb_rw tr;
   this.out_chan.activate(tr);
   // Pre-exec callbacks - could be tasks
   this.out_chan.start();
   // 'tr' executed and annotated with response
   this.out_chan.complete();
   // Post-exec callbacks - could be tasks
   this.out_chan.remove();
end
```

2) `peek/get`

```
forever begin
   vmm_apb_rw tr;
   this.out_chan.peek(tr);
   tr.notify.indicate(vmm_data::STARTED);
   // 'tr' executed and annotated with response
   tr.notify.indicate(vmm_data::ENDED);
   this.out_chan.get(tr);
end
```

3) Simple

```
forever begin
   vmm_apb_rw tr;
   this.out_chan.peek(tr);
   // 'tr' executed and annotated with response
   this.out_chan.get(tr);
end
```

b) A pipelined VMM consumer uses `get` to retrieve requests from the channel. Multiple requests may be executing in parallel, so `get` can not be used by the adapter to indicate availability of a response. After a request is executed, the consumer annotates the response and indicates the `ENDED` status in the original request. The adapter needs to watch for each pending requests's `ENDED` status to determine when to send a corresponding response back to the OVM producer.

```
forever begin
   vmm_apb_rw tr;
   // Wait for pipeline to be ready to accept transaction
   this.out_chan.get(tr);
   fork
      begin
         // 'tr' executed and annotated with response
         tr.notify.indicate(vmm_data::ENDED);
      begin
   join_none
end
```

c) A VMM consumer using a separate response channel `sneak`s responses to that channel upon completion of each request. The adapter has an active process that gets responses from the response channel and converts them to OVM responses, which are then sent back to the OVM producer.

1) Atomic request/response

```
forever begin
   apb_req req;
   apb_rsp rsp;
   this.req_chan.get(req);
   rsp = new(req);
   // 'req' executed and 'rsp' annotated with response
   req.notify.indicate(vmm_data::ENDED, rsp);
   this.rsp_chan.sneak(rsp); // better than using put()
end
```

2) Pipelined request/response

```
forever begin
   apb_req req;
   // Wait for pipeline to be ready to accept transaction
   this.req_chan.get(req);
   fork
      begin
         apb_rsp rsp;
         rsp = new(req);
         // 'req' executed and 'rsp' annotated with response
         req.notify.indicate(vmm_data::ENDED, rsp);
         this.rsp_chan.sneak(rsp); // Can also use put()
      end
   join_none
end
```

3) Multiple responses

```
forever begin
   apb_req req;
   this.req_chan.get(req);
```

```
      fork
         do begin
            apb_rsp rsp;
            rsp = new(req);
            // 'req' executed and 'rsp' annotated with response
            this.rsp_chan.sneak(rsp);
         end while …;
      join_none
   end
```

However the VMM consumer indicates the completion of a request or the availability of a response transaction, the adapter detects completion of each VMM request, matches it with the original OVM request (using their unique transaction ids), converts it to an OVM response, and deliver it back to the OVM producer.

Table 4 provides collaboration requirements and limitations for each of the possible OVM producer / VMM consumer pairings.

Table 4—avt_tlm2channel collaborations

VMM consumer / OVM producer	Atomic[1]	peek /get[1]	Simple[1]	Pipeline[1,2]	Atomic with rsp channel	Pipelined rsp channel	Multiple response rsp channel
Sequencer	yes	yes	yes	yes	yes	yes	yes
Push sequencer	yes	yes	yes	yes	yes	yes	yes
Blocking or non-blocking put	yes	yes	yes	yes	*yes*[3]	*yes*[3]	*yes*[3]
Blocking transport	yes	yes	yes	yes	yes	*yes*[4]	*no*
Blocking or non-blocking master	yes	yes	yes	yes	yes	yes	yes
Passive blocking slave	yes	yes	yes	yes	yes	yes	yes
Passive blocking get / peek	yes	yes	yes	yes	*yes*[3]	*yes*[3]	*yes*[3]
Passive blocking get / peek with rsp	yes	yes	yes	yes	yes	yes	yes

[1]Adaptor's **rsp_is_req** bit needs to be set. **VMM_REQ** and **VMM_RSP** types shall be the same.
[2]Requires the adapter wait for pending `vmm_data::ENDED` notifications.
[3]Needs to sink the adapter's response channel.
[4]May create idle cycles in the pipeline between requests.

5.6.6 Implementation

To implement the practice, the integrator instantiates an OVM producer, a VMM consumer, and an **avt_tlm2channel** adapter (see 5.6.4) whose parameter values correspond to the OVM and VMM data types used by the producer and consumer and the converter types used to translate in one or both directions. If the default `vmm_channel`s created by the VMM consumer or adapter are not used, the integrator shall also instantiate a request `vmm_channel` and a response `vmm_channel` if the VMM consumer uses one.

Integrators of VMM-on-top environments need to instantiate the OVM consumer and adapter via an OVM wrapper component. This wrapper component serves to provide the `connect` method needed to bind the OVM ports and exports.

5.6.7 Sample code

This section illustrates how to implement this practice within an OVM and VMM environment. The two examples share the following `typedef`.

```
typedef avt_tlm2channel #(ovm_apb_rw, vmm_apb_rw,
                          apb_rw_convert_ovm2vmm,
                          vmm_apb_rw,ovm_apb_rw,
                          apb_rw_convert_vmm2ovm) apb_tlm2channel;
```

5.6.7.1 OVM-on-top

```
class env extends ovm_component;

  `ovm_component_utils(env)

   ovm_producer #(ovm_apb_rw) o_prod;
   vmm_consumer #(vmm_apb_rw) v_cons;
   apb_tlm2channel            adapter;

   function new (string name="env",ovm_component parent=null);
      super.new(name,parent);
   endfunction

   virtual function void build();
      o_prod  = new("o_prod", this);
      v_cons  = new("v_cons",0);
      adapter = new("adapter",this,v_cons.in_chan);
endfunction

   virtual function void connect();
      o_prod.blocking_put_port.connect(adapter.put_export);
   endfunction

...
endclass
```

5.6.7.2 VMM-on-top

```
class env extends avt_vmm_ovm_env;

   ‘ovm_build
   ovm_producer #(ovm_apb_rw) o_prod;
   vmm_consumer #(vmm_apb_rw) v_cons;
   apb_tlm2channel            adapter;
```

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

      virtual function void build();
         o_prod  = new("o_prod", this);
         v_cons  = new("v_cons",0);
         adapter = new("adapter",this,v_cons.in_chan);
         o_prod.blocking_put_port.connect(adapter.put_export);
         ovm_build();
      endfunction

      ...
   endclass
```

5.7 Channel to TLM

5.7.1 Practice name

Channel to TLM

5.7.1.1 Also known as

VMM producer to OVM consumer; Channel adapter.

5.7.1.2 Related practices

Meta-composition (5.3), *Data conversion* (5.5), and *TLM to channel* (5.6).

5.7.2 Intent

This practice is intended to enable any VMM producer component communicating via a `vmm_channel` to communicate at the transaction level with any OVM consumer communicating via TLM ports and exports.

5.7.2.1 Motivation

When integrating OVM and VMM components, each component has established semantics that define the communication and the mechanics of establishing the communication path. This practice and its supporting adapter are designed to present an OVM-compliant interface on the "OVM side" and a VMM-compliant interface on the "VMM side," thus allowing the two components to interoperate.

For OVM components, the communication semantics of a particular connection are fully defined by the set of ports and/or exports used to establish the connection. In VMM, the `vmm_channel` API supports multiple semantics depending on the implementation of the consumer. The **avt_channel2tlm** adapter (see 5.7.4) allows connection to any established VMM producer and properly converts the desired semantics of the `vmm_channel` connection(s) used by it into the corresponding TLM semantics required by the OVM consumer, regardless of which TLM port/export it uses, subject to the clarifications specified in the footnotes to Table 5.

5.7.2.2 Consequences

Integrators need to provide an implementation of the *Data conversion* (5.5) practice for converting VMM requests to OVM requests and, when separate responses are involved, converting OVM responses to VMM requests.

Integrators of VMM-on-top environments need to apply the *Meta-composition* (5.3) practice to define a simple OVM component that contains (wraps) all OVM VIP and adapters used in any given VMM scope. This enables port connections between OVM components to be made in the connect phase of the OVM wrapper.

The VMM producer is more likely aware of channel size requirements for its own completion model than the generic **avt_channel2tlm** adapter (see 6.7). Thus, the practice recommends letting the VMM producer allocate its own channel, then passing that channel to the adapter via its constructor. A consequence of this recommendation is the VMM producer needs to be allocated prior to the adapter.

When using the `blocking_put_port` of the **avt_channel2tlm** adapter, a `tlm_fifo` should not be used as the consumer.

5.7.3 Applicability

This practice is useful for connecting VMM producer components that use `vmm_channels` to communicate to OVM consumers.

5.7.4 Structure

The overall structure for this practice is based on the following diagrams, prototype, and participants.

5.7.4.1 Class diagram

The class diagram for this practice is shown in Figure 12.

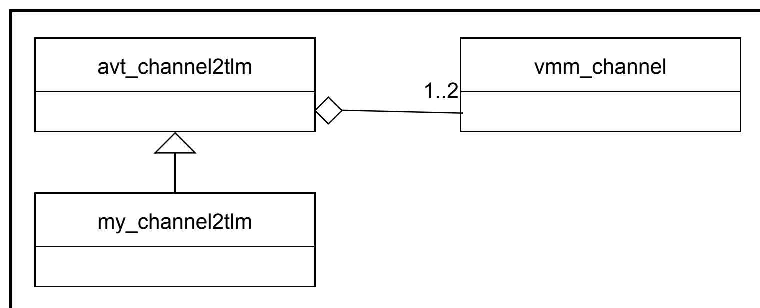

Figure 12—avt_channel2tlm derived class

5.7.4.2 Declaration prototype

This practice uses the following declaration prototype.

```
class avt_channel2tlm_adapter #(type VMM_REQ     = int,
                               OVM_REQ     = int,
                               VMM2OVM_REQ = int,
                               OVM_RSP     = OVM_REQ,
                               VMM_RSP     = VMM_REQ,
```

```
                                OVM2VMM_RSP = avt_converter #(OVM_RSP,VMM_RSP),
                                OVM_MATCH_REQ_RSP=avt_match_ovm_id)
                                          extends ovm_component;

function new( string name = "avt_channel2tlm",
              ovm_component parent = null,
              vmm_channel_typed #(VMM_REQ) req_chan = null,
              vmm_channel_typed #(VMM_RSP) rsp_chan = null,
              bit rsp_is_req = 1,
              int unsigned max_pending_req = 100);
   ...
endfunction
```

See 6.1 for a description of each `type` parameter.

5.7.4.3 Interaction diagram

The interaction diagram for this practice is shown in Figure 13. A single **avt_channel2tlm** adapter instance (see 6.7) enables connection of one of the VMM producer types on the left to one of the OVM consumer types on the right, subject to the conditions set forth in Table 5.

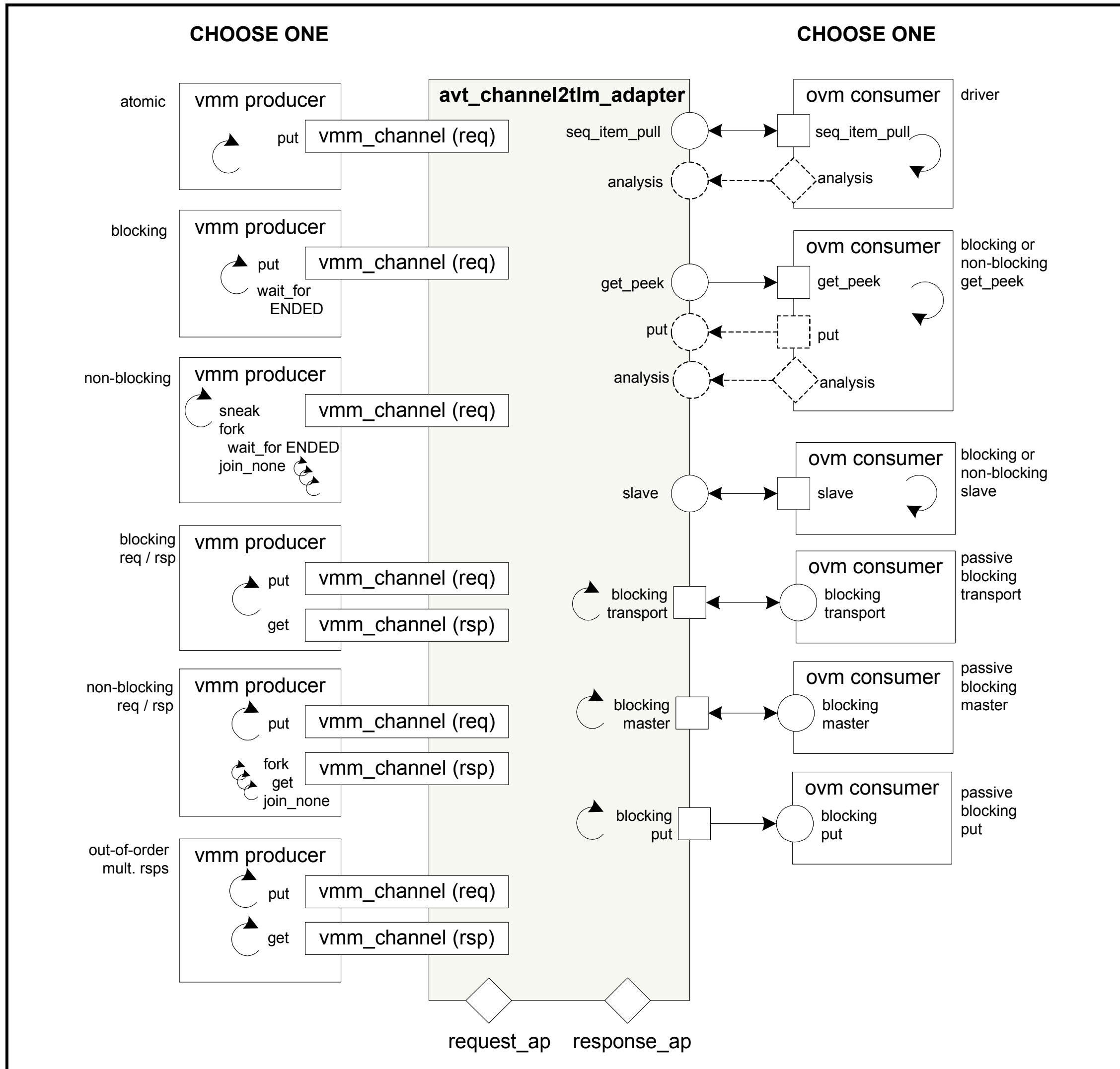

Figure 13—avt_channel2tlm interaction

5.7.4.4 Participants

Each application of this practice involves the following participants.

— An instance of one and only one of the `vmm producer` types shown in Figure 13, subject to the collaboration requirements in Table 5.

— An instance of one and only one of the `ovm consumer` types shown in Figure 13, subject to the collaboration requirements in Table 5.

— An instance of a request `vmm_channel`. Typically, the VMM producer allocates its own channel, which is subsequently passed to the **avt_channel2tlm** adapter's constructor (see 6.7).

— An instance of a response `vmm_channel` if the selected VMM producer uses a separate channel for responses. Typically, the VMM producer allocates its own channel, which is subsequently passed to the **avt_channel2tlm** adapter's constructor.

— An instance of the **avt_channel2tlm** adapter whose `type` parameters are matched to the types used by the VMM producer and OVM consumer.

5.7.5 Collaboration

The VMM producer uses `put` or `sneak` VMM-typed transactions into the request `vmm_channel`.

An active OVM consumer initiates requests for new transactions via its port, which can be of any interface type depicted in Figure 13. New request transactions are obtained by calling the port's `peek` or `get` method, which end up calling the **avt_channel2tlm** (see 6.7) adapter's `peek` or `get` via its corresponding export. In response, the adapter fetches a VMM transaction from the channel, converts it to OVM, and returns.

A passive OVM consumer implements a `put` method that is called via its export. If either of the adapter's blocking master or blocking put ports are connected, the adapter forks a process that continually fetches transactions from the channel, converts them to OVM, and sends them to the consumer via its port.

Depending on what type is being adapted, the VMM producer may expect responses to be annotated in the original requests and it may wait for request's `ENDED` status to determine when a valid response is available. The OVM consumer delivers responses explicitly. To accommodate this, the adapter stores handles to all outstanding requests so it can correlate incoming responses from the OVM consumer to the originating VMM requests. When a match is found, the adapter employs the *user-defined response converter* to annotate the response back into the original request and indicate the request's `ENDED` status. By default, the adapter holds a maximum of 100 pending requests. The depth may be set via the `max_pending_req` constructor argument, or by setting the `max_pending_req` value directly. A `FATAL` error is reported if the number of pending requests exceeds `max_pending_req`.

There are several types of VMM producer semantics that are supported by this adapter.

a) Atomic

```
forever begin
   vmm_apb_rw tr = new;
   // Fill in response part of 'tr'
   this.out_chan.put(tr);
   // 'tr' annotated with response
end
Requires: out_chan.full_level() == 1
```

b) Blocking

```
forever begin
   vmm_apb_rw tr = new;
   // Fill in response part of 'tr'
   this.out_chan.put(tr);
   tr.notify.wait_for(vmm_data::ENDED);
   // 'tr' annotated with response
   // or response in ENDED status
end
```

c) Non-blocking

```
forever begin
   vmm_apb_rw tr = new;
   // Fill in response part of 'tr'
   this.out_chan.sneak(tr);
   fork
      begin
         tr.notify.wait_for(vmm_data::ENDED);
         // 'tr' annotated with response
         // or response in ENDED status
      end
   join_none
```

```
   // Need some other blocking mechanism in this thread
end
//Required: The producer thread shall block using an external mechanism.
//This is not supported in non-atomic cases.
```

d) Blocking request/response

```
forever begin
   apb_req req = new;
   apb_rsp rsp;
   this.req_chan.put(req); // Can use sneak() too
   this.rsp_chan.get(rsp);
end
//Required: Consumer needs to provide one response per request,
//in the same order
```

e) Non-blocking request/response

```
forever begin
   apb_req req = new;
   this.req_chan.sneak(req);
   fork
      begin
         apb_rsp rsp;
         this.rsp_chan.get(rsp);
      end
   join_none
end
//Required:
// 1. The producer thread shall block using an external mechanism.
// 2. The consumer shall provide one response per request, in same order.
```

f) Out of order/multiple responses

```
fork
   forever begin
      apb_req req = new;
      this.pending_reqs.push_back(req);
      this.req_chan.put(req);
   end

   forever
      apb_rsp rsp;
      this.rsp_chan.get(rsp);
      foreach(this.pending_reqs[i]) begin
         if (rsp.is_response_to(this.pending_reqs[i])) begin
            // Handle response
            rsp = null;
            break;
         end
      end
      assert (rsp == null);
   end
join
//Required: Needs a user-defined mechanism to correlate the response
//with the request.
```

Table 5 provides collaboration requirements and limitations for each of the possible OVM producer / VMM consumer pairings pictured in Figure 13.

Table 5—avt_channel2tlm collaborations

OVM consumer / VMM producer	Sequence driver	get / peek	get / peek with rsp	Blocking or non-blocking slave	Passive blocking transport	Passive blocking master	Passive blocking put
Atomic[1]	*yes*[2 or 3]	*yes*[2]	yes	yes	yes	*yes*[4]	*yes*[5]
Blocking[1]	*yes*[2 or 3]	*yes*[2]	yes	yes	yes	*yes*[4]	*yes*[5]
Non-blocking[1]	*yes*[2 or 3]	*yes*[2]	yes	yes	yes	yes	*yes*[5]
Blocking rsp channel	*yes*[3]	*n/a*	yes	yes	yes	yes	*yes*[5]
Non-blocking rsp channel	*yes*[3]	*n/a*	yes	yes	yes	yes	*yes*[5]
Out-of-order multiple responses rsp channel	*yes*[3]	*n/a*	yes	yes	yes	yes	*yes*[5]

[1]Need **rsp_is_req** bit set, need **OVM2VMM_RSP** converter. **VMM_REQ** and **VMM_RSP** shall be the same.
[2]Needs to do `peek`, annotate response in request, then `get`.
[3]Needs to provide an explicit response.
[4]Needs to get responses in order. Do not get from channel until it has a response.
[5]Needs to annotate response in request before returning from `put`. Responses are delivered in-order.

5.7.6 Implementation

To implement the practice, the integrator instantiates a VMM producer, an OVM consumer, and an **avt_channel2tlm** adapter (see 5.7.4) whose parameter values correspond to the VMM and OVM data types used by the producer and consumer and the converter types used to translate in one or both directions. If the default `vmm_channels` created by the VMM producer or adapter are not used, the integrator shall also instantiate a request `vmm_channel` and, possibly, a response `vmm_channel` if the VMM producer uses one.

Integrators of VMM-on-top environments need to instantiate the OVM consumer and adapter via an OVM `container` or wrapper component. This wrapper component serves to provide the `connect` method needed to bind the OVM ports and exports.

5.7.7 Sample code

This illustrates how to implement this practice.

```
typedef avt_channel2tlm #(vmm_apb_rw,ovm_apb_rw,
                          apb_rw_convert_vmm2ovm,
                          ovm_apb_rw, vmm_apb_rw,
                          apb_rw_convert_ovm2vmm) apb_channel2tlm;
```

5.7.7.1 OVM-on-top

```
class env extends ovm_component;

   apb_rw_atomic_gen v_prod;
   ovm_consumer #(ovm_apb_rw)o_cons;
   apb_channel2tlm adapter;

   function new (string name="env",ovm_component parent=null);
      super.new(name,parent);
   endfunction

   virtual function void build();
      v_prod  = new("v_prod",1);
      o_cons  = new("o_cons", this);
      adapter = new("adapter",this,v_prod.out_chan);
   endfunction

  // Connect - Connect the OVM producer to the channel
  //           adapter using standard port connections.

   virtual function void connect();
      o_cons.blocking_get_port.connect(adapter.get_peek_export);
   endfunction
endclass
```

5.7.7.2 VMM-on-top

```
class env extends avt_vmm_ovm_env;

   'ovm_build

   apb_rw_atomic_gen v_prod;
   ovm_consumer #(ovm_apb_rw)o_cons;
   apb_channel2tlm adapter;

   function new (string name="env");
      super.new(name);
   endfunction

   virtual function void build();
      v_prod  = new("v_prod",1);
      o_cons  = new("o_cons", this);
      adapter = new("adapter",this,v_prod.out_chan);
      o_cons.blocking_get_port.connect(adapter.get_peek_export);
      ovm_build();
   endfunction

   ...
endclass
```

5.8 Analysis to channel / Channel to analysis

5.8.1 Practice name

Analysis channel

5.8.1.1 Also known as

OVM publisher to VMM channel; VMM channel to OVM subscriber; analysis port to channel; channel to analysis port.

5.8.1.2 Related practices

Data conversion (see 5.5), *TLM to channel* (5.6), and *Channel to TLM* (5.7).

5.8.2 Intent

Allows a component to emit a transaction object to multiple consumers, each of which gets a reference to (or optionally a copy of) the source transaction.

5.8.2.1 Motivation

It is common for one component, such as a monitor, to emit transactions to multiple other components, such as coverage collectors and/or scoreboards. Each recipient is unaware other recipients exist and the functionality of the source component needs to be independent of the number of possible recipients.

5.8.2.2 Consequences

For the VMM user, one-to-many communication may be performed as in a VMM-only environment, in which the source may be connected to a `vmm_broadcast` and/or use `vmm_notify` and/or callbacks to publish a transaction to multiple possible recipients. In the case of the `vmm_broadcast`, the originating `vmm_xactor` communicates to the broadcast via a single `vmm_channel`, and each target of the `vmm_broadcast` is similarly connected via a specific `vmm_channel`. For a discussion of the `vmm_notify` usage, see 5.9. For a discussion of the callback usage, see 5.10.

For the OVM user, one-to-many communication is accomplished using an `ovm_analysis_port`. To connect to VMM recipients, the **avt_analysis_channel**'s `analysis_export` is connected to the `analysis_port` (see 6.8). The VMM recipient may then get the transaction out of the adapter's underlying channel.

5.8.3 Applicability

The **avt_analysis_channel** (see 6.8) allows the VMM source to `put()` or `sneak()` the transaction into the adapter's underlying `vmm_channel`. One or more OVM subscribers may then connect to the adapter's `analysis_port` to receive the transaction via the `write()` method. The advantage of the analysis adapter over the broadcast approach is that the data conversion is only performed once for all OVM recipients (see Figure 15).

When the OVM source publishes the transaction to its `analysis_port`, the **avt_analysis_channel** places the converted transactions in the underlying channel via the `sneak()` method (which is non-blocking). If multiple VMM recipients are required, then a `vmm_broadcast`'s input channel may be connected to the analysis adapter's channel and multiple VMM transactors may then be connected via their own channels to the `vmm_broadcast`. This allows multiple VMM recipients while only requiring a single conversion.

5.8.4 Structure

The overall structure for this practice is based on the following diagrams, prototype, and participants.

5.8.4.1 Class diagram

The class diagram for this practice is shown in Figure 14.

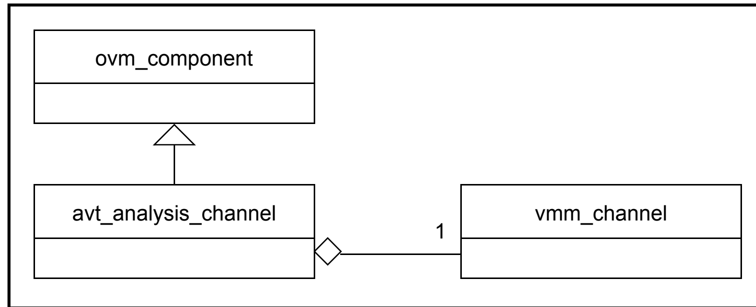

Figure 14—avt_analysis_channel class

5.8.4.2 Declaration prototype

The **avt_analysis_channel** (see 6.8) is extended from `ovm_component` as follows.

```
class avt_analysis_channel #(type OVM=int, VMM=int,
                             OVM2VMM=avt_converter #(OVM,VMM),
                             VMM2OVM=avt_converter #(VMM,OVM))
                          extends ovm_component;
```

5.8.4.3 Interaction diagrams

Figure 15 shows the structure with a `vmm_xactor` connected to an **avt_analysis_channel** object (see 6.8). Each of the `ovm_subscriber` targets gets its transaction from the `analysis_port` of the **avt_analysis_channel**.

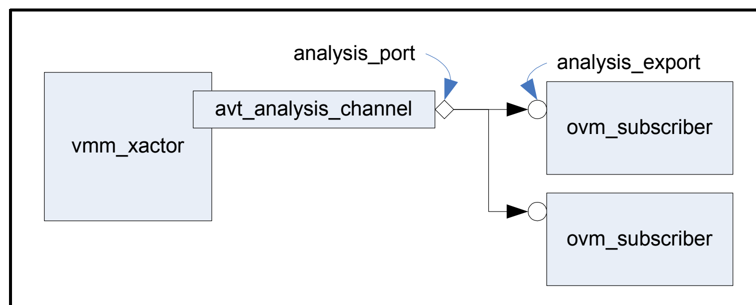

Figure 15—vmm_xactor to multiple ovm_subscriber

Figure 16 shows the structure with an `ovm_analysis_port` connected to multiple analysis adapters. The target `vmm_xactors` are connected via shared reference to the underlying `vmm_channel` in each **avt_analysis_channel**. These `vmm_xactors` would simply get the transactions from the appropriate channel and process them in some application-specific way.

NOTE—The **avt_analysis_channel** could also be connected on the VMM side to a `vmm_broadcast` which could, in turn, connect to multiple `vmm_xactors`. In this case, the OVM-to-VMM data conversion would only need to be done once.

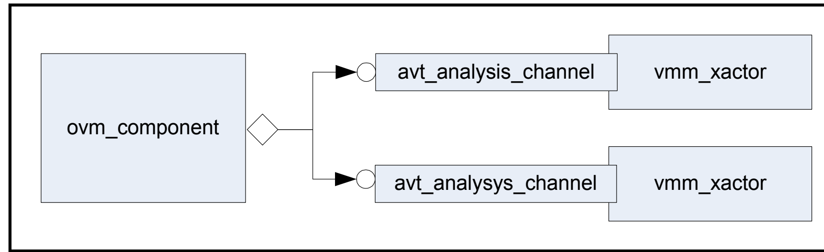

Figure 16—ovm_analysis_port to vmm_channel_adapters

5.8.4.4 Participants

On the OVM side, the `vmm_analysis_adapter` is used to connect to the originating `ovm_analysis_port`. On the VMM side, the `vmm_channel` inside the adapter is used as any `vmm_channel` would be.

5.8.5 Collaboration

In the OVM-to-VMM case, the `write()` method of the adapter (see the `write()` function declaration in 5.8.4.2) takes an `ovm_transaction` extension as an argument, converts it to the corresponding `vmm_data` extension, and inserts the transaction into the underlying `vmm_channel` via the `sneak()` method. In the VMM-to-OVM case, when a `vmm_data` extension object is put into the adapter's `vmm_channel`, the adapter automatically gets the transaction from the channel, converts it to the OVM transaction type, and publishes it to the adapter's `analysis_port`. The OVM subscriber(s) receives the transaction via the adapter's `analysis_port` (see 6.8).

5.8.6 Implementation

To implement the practice, the integrator connects the `avt_analysis_channel` to the `ovm_component` on the OVM side, and its underlying `vmm_channel` is connected to the `vmm_xactor`.

5.8.7 Sample code

This illustrates how to implement this practice.

a) To write an OVM transaction to multiple VMM recipients in an OVM container, and vice-versa, use:

```
class example extends ovm_component;

  // OVM source -> VMM sink
  ovm_publish #(ovm_apb_rw) o_prod;
  vmm_consumer #(apb_rw)    v_cons;
  apb_analysis_channel      o_to_v;

  // VMM source -> OVM sink
  apb_rw_atomic_gen           v_prod;
  ovm_subscribe #(ovm_apb_rw) o_cons;
  apb_analysis_channel        v_to_o;

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

```
  virtual function void build();

    o_prod = new("o_prod",this);
    v_cons = new("v_cons");
    o_to_v = new("o_to_v ",this, v_cons.in_chan);

    v_prod = new("v_prod");
    o_cons = new("o_cons",this);
    v_to_o = new("v_to_o",this, v_prod.out_chan);

    v_prod.stop_after_n_insts = 1;

  endfunction

  virtual function void connect();
    o_prod.out.connect(o_to_v.analysis_export);
    v_to_o.analysis_port.connect(o_cons.analysis_export);
  endfunction

  virtual task run();
    v_cons.start_xactor();
    v_prod.start_xactor();
  endtask

endclass
```

b) To write an VMM transaction to multiple OVM recipients in an VMM container, and vice-versa, use:

```
class example extends vmm_xactor;

  // OVM source -> VMM sink
  ovm_publish #(ovm_apb_rw) o_prod;
  vmm_consumer #(apb_rw)    v_cons;
  apb_analysis_channel      o_to_v;

  // VMM source -> OVM sink
  apb_rw_atomic_gen           v_prod;
  ovm_subscribe #(ovm_apb_rw) o_cons;
  apb_analysis_channel        v_to_o;

  function new(...);
    super.new(...);

    o_prod = new("o_prod",);
    v_cons = new("v_cons");
    o_to_v = new("o_to_v ", , v_cons.in_chan);

    v_prod = new("v_prod");
    o_cons = new("o_cons",);
    v_to_o = new("v_to_o", , v_prod.out_chan);

    v_prod.stop_after_n_insts = 1;

   o_prod.out.connect(o_to_v.analysis_export);
    v_to_o.analysis_port.connect(o_cons.analysis_export);
  endfunction

  virtual task start();
```

```
    v_cons.start_xactor();
    v_prod.start_xactor();
  endtask

endclass
```

5.9 Notify to analysis / Analysis to notify

5.9.1 Practice name

Notification service adapter (analysis_port)

5.9.1.1 Also known as

Analysis port to VMM indicate; VMM indicate to analysis export; OVM publisher to VMM notification; VMM notification to OVM subscriber.

5.9.1.2 Related practices

None.

5.9.2 Intent

Allows a VMM component with a vmm_notify instance to be connected to one or more OVM components with an ovm_analysis_export. Allows an OVM component with an ovm_analysis_port to be connected to one or more VMM components with a vmm_notify instance.

5.9.2.1 Motivation

In addition to communicating via transaction descriptors, components may communicate transaction-asynchronous or non-transactional information. The "source" indicates that information is available and one or more "listeners" react to that information. In VMM, this may be accomplished using the notification service provided by the vmm_notify class. In OVM, this may be accomplished using analysis ports.

5.9.2.2 Consequences

In addition to using different notification mechanisms, the attached status descriptor uses a different SystemVerilog type to describe the same information. The OVM status descriptor would be based on the ovm_sequence_item class, whereas the VMM status descriptor would be based on the vmm_data class. In addition to providing an adapter component with an OVM-compliant analysis interface and a vmm_notify, this practice needs to perform the necessary translation of the transaction descriptor.

Theoretically, it is possible for the type of the status descriptor associated with a single VMM notification to be of different types, but that does not happen in practice, and it is a reasonable limitation to require that a VMM notification carries a consistent status object type to be adapted to an ovm_analysis_port.

5.9.3 Applicability

This practice can connect a notification in a vmm_notify to an ovm_analysis_port and an ovm_analysis_port to a notification in a vmm_notify instance.

5.9.4 Structure

The overall structure for this practice is based on the following diagrams, prototype, and participants.

5.9.4.1 Class diagram

The class diagram for this practice is shown in Figure 17.

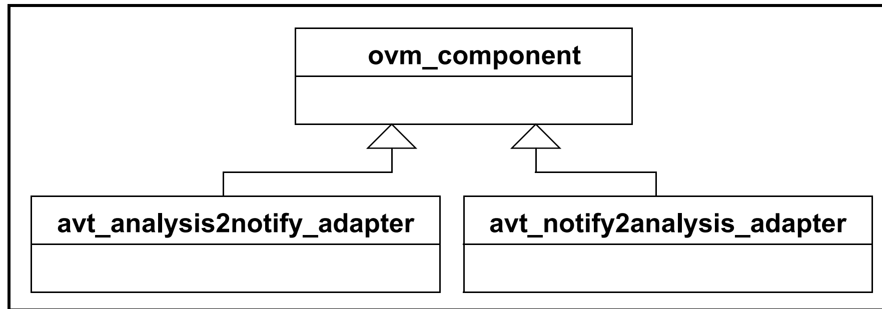

Figure 17—analysis_port class

5.9.4.2 Declaration prototype

This practice uses the following declaration prototype.

```
class avt_analysis2notify #(type OVM = int, VMM = int, OVM2VMM = int)
                           extends ovm_component;

class avt_notify2analysis #(type VMM = int, OVM = int, VMM2OVM = int)
                           extends ovm_component;
```

5.9.4.3 Interaction diagrams

The interaction diagram for this practice is shown in Figure 18.

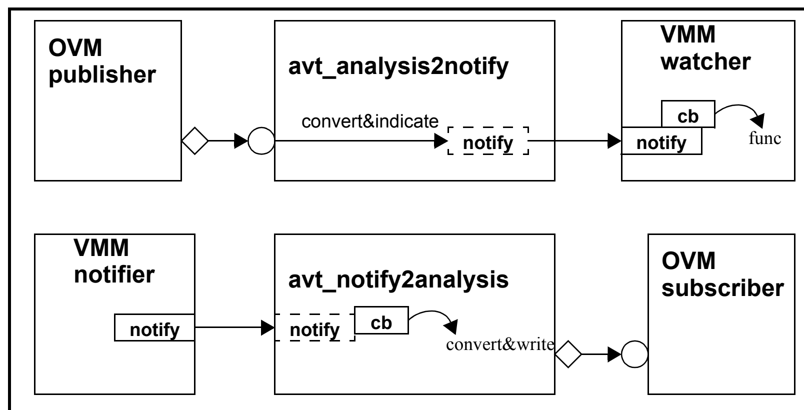

Figure 18—Using avt_analysis2notify and avt_notify2analysis for sideband communication

5.9.4.4 Participants

In addition to the OVM and VMM source/sink components, the adapters are instantiated to covert any status objects. An OVM component with an analysis port writes objects on that port whenever relevant. A VMM component, listening to the corresponding VMM notification, receives a VMM version of that object. A

VMM component indicates a notification with a status object whenever relevant. An OVM component, listening to the corresponding analysis export, receives an OVM version of that object.

5.9.5 Collaboration

A VMM source indicates a notification using the `vmm_notify::indicate()` method. The adapter converts the notification status data into an OVM status transaction. The adapter then publishes the status transaction to its `analysis_port`, to which multiple OVM listeners may connect.

An OVM source publishes a status transaction on an analysis port using the `write()` method. The adapter converts the published status transaction into a VMM status descriptor. The adapter indicates the corresponding notification in its `vmm_notify` object, to which multiple VMM listeners may listen.

5.9.6 Implementation

To implement the practice, the integrator instantiates the appropriate adapter, parameterized according to the OVM and VMM data types, and the corresponding converter classes. The analysis port or export is connected to the desired OVM component(s). The desired `vmm_notify` object may be passed into the adapter as a constructor argument or assigned directly.

5.9.7 Sample code

This illustrates how to implement this practice.

5.9.7.1 VMM-on-top

```
typedef avt_analysis2notify
            #(ovm_apb_rw,vmm_apb_rw,
              apb_rw_convert_ovm2vmm) apb_analysis2notify_adapter;

typedef avt_notify2analysis
            #(vmm_apb_rw,ovm_vmm_apb_rw,
              apb_rw_convert_vmm2ovm) apb_notify2analysis_adapter;

class tb_env extends vmm_env;
   vmm_sink vmm_snk;
   ovm_src  ovm0;
   ovm_src  ovm1;
   ovm_src  ovm2;
   apb_analysis2notify_adapter o2v_adapter0, o2v_adapter1,o2v_adapter2;

   ovm_sink ovm_snk;
   vmm_src  vmm0;
   vmm_src  vmm1;
   vmm_src  vmm2;
   apb_notify2analysis_adapter v2o_adapter0, v2o_adapter1, v2o_adapter2;

   virtual function void build();
      super.build();
      this.vmm_snk = new("vmm_snk");

      this.o2v_adapter0 = new(“o2v0”, ,vmm_snk.notify,0);
      this.o2v_adapter1 = new(“o2v1”, ,vmm_snk.notify,1);
      this.o2v_adapter2 = new(“o2v2”, ,vmm_snk.notify,2);
```

```
        this.ovm0 = ovm_src::type_id::create("tb.ovm0", null);
        this.ovm1 = ovm_src::type_id::create("tb.ovm1", null);
        this.ovm2 = ovm_src::type_id::create("tb.ovm2", null);

        ovm0.ap.connect(o2v_adapter0.analysis_export);
        ovm1.ap.connect(o2v_adapter1.analysis_export);
        ovm2.ap.connect(o2v_adapter2.analysis_export);

        this.ovm_snk = ovm_sink::type_id::create("ovm_snk", ovm_top);

        this.vmm0 = new("vmm0");
        this.vmm1 = new("vmm1");
        this.vmm2 = new("vmm2");

        this.v2o_adapter0 = new("v2o_0", ,this.vmm0.notify,0);
        this.v2o_adapter1 = new("v2o_1", ,this.vmm1.notify,0);
        this.v2o_adapter2 = new("v2o_2", ,this.vmm2.notify,0);
        v2o_adapter0.analysis_port.connect(ovm_snk.ap0.exp);
        v2o_adapter1.analysis_port.connect(ovm_snk.ap1.exp);
        v2o_adapter2.analysis_port.connect(ovm_snk.ap2.exp);

    endfunction: build
endclass
```

Notice this example allows the adapters, `ovm_sink`, and `ovm_src` components to be connected in the `build()` method after they have been constructed. This is only possible because these particular components instantiate their relevant ports/exports in their own constructors, which is not typical. See 5.3 for a more detailed discussion of connecting OVM components in a VMM environment.

5.9.7.2 OVM-on-top

```
typedef avt_analysis2notify
            #(ovm_apb_rw,vmm_apb_rw,
              apb_rw_convert_ovm2vmm) apb_analysis2notify_adapter;

typedef avt_notify2analysis
            #(vmm_apb_rw,ovm_apb_rw,
              apb_rw_convert_vmm2ovm) apb_notify2analysis_adapter;
class ovm_on_top_env extends ovm_env;

    vmm_sink vmm_snk;
    ovm_src  ovm0;
    ovm_src  ovm1;
    ovm_src  ovm2;
    apb_analysis2notify_adapter o2v_adapter0, o2v_adapter1,o2v_adapter2;

    ovm_sink ovm_snk;
    vmm_src  vmm0;
    vmm_src  vmm1;
    vmm_src  vmm2;
    apb_notify2analysis_adapter v2o_adapter0, v2o_adapter1, v2o_adapter2;

    virtual function void build();
        this.vmm_snk = new("vmm_snk");

        this.ovm0 = ovm_src::type_id::create("ovm0", this);
        this.ovm1 = ovm_src::type_id::create("ovm1", this);
        this.ovm2 = ovm_src::type_id::create("ovm2", this);
```

```
      this.v2o_adapter0 = new(“v2o_0”, , this.vmm0.notify, 0);
      this.v2o_adapter1 = new(“v2o_1”, , this.vmm0.notify, 0);
      this.v2o_adapter2 = new(“v2o_2”, , this.vmm0.notify, 0);

      this.ovm_snk = ovm_sink::type_id::create("ovm_snk", ovm_top);

      this.vmm0 = new("vmm0");
      this.vmm1 = new("vmm1");
      this.vmm2 = new("vmm2");
   endfunction

   virtual function void connect();
      ovm0.ap.connect(o2v_adapter0.analysis_export);
      ovm1.ap.connect(o2v_adapter1.analysis_export);
      ovm2.ap.connect(o2v_adapter2.analysis_export);

      v2o_adapter0.analysis_port.connect(ovm_snk.ap0.exp);
      v2o_adapter1.analysis_port.connect(ovm_snk.ap1.exp);
      v2o_adapter2.analysis_port.connect(ovm_snk.ap2.exp);
   endfunction
endclass
```

5.10 Callback adapter

5.10.1 Practice name

Callback adapter

5.10.1.1 Also known as

One-to-many interconnect; `analysis_port/vmm_callback` adaptor.

5.10.1.2 Related practices

Data conversion (see 5.5), *TLM to channel* (see 5.6), *Channel to TLM* (see 5.7), *Analysis to channel / Channel to analysis* (see 5.8), and *Notify to analysis / Analysis to notify* (see 5.9).

5.10.2 Intent

Allows a VMM component with a callback method to be connected to one or more OVM components with an analysis export.

5.10.2.1 Motivation

When integrating OVM and VMM components, each component has established semantics that define the communication of transaction-level information and the mechanics of establishing the communication path. An adapter can present the OVM semantics on the “OVM side” and the VMM semantics on the “VMM side.”

5.10.2.2 Consequences

In addition to using different transaction-level communication mechanisms, it is very likely each methodology would use a different SystemVerilog type to describe the same the transaction. The OVM transaction descriptor would be based on the `ovm_sequence_item` class, whereas the VMM transaction descriptor would be based on the `vmm_data` class. In addition to providing a callback facade extension

with an OVM-compliant analysis interface, this practice needs to perform the necessary translation of the transaction descriptor.

5.10.3 Applicability

This practice is useful for connecting an observed transaction or event via a callback method in a `vmm_xactor` to an `ovm_analysis_port`.

5.10.4 Structure

The overall structure for this practice is based on the following class diagrams, prototype, and participants.

5.10.4.1 Class diagram

The class diagram for this practice is shown in Figure 19.

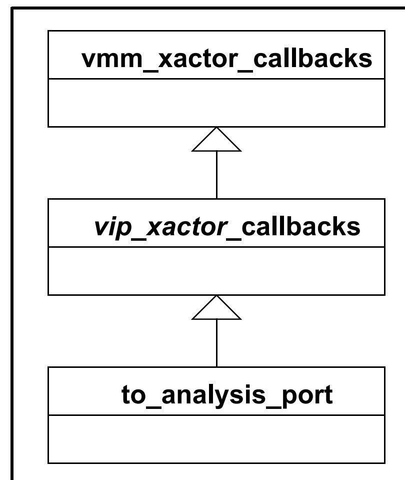

Figure 19—Callback class

5.10.4.2 Declaration prototype

This practice assumes the existence of an appropriate callback method and callback facade class in the VMM component, e.g.,

```
class apb_master_cbs extends vmm_xactor_callbacks;
    virtual task pre_tr(apb_master xactor,
                        apw_rw     tr,
                        ref bit    drop);
    endtask
    virtual task post_tr(apb_master xactor,
                         apw_rw     tr);
    endtask
endclass
```

5.10.4.3 Interaction diagrams

The interaction diagram for this practice is shown in Figure 20.

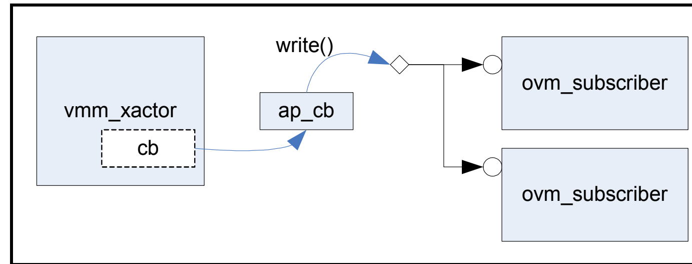

Figure 20—VMM callback to OVM analysis_port

5.10.4.4 Participants

This practice requires a VMM transactor with an observation callback method and a OVM subscriber with an `ovm_analysis_export`.

Between the producer and subscriber, this practice instantiates a callback object to transfer the observed data to the analysis port via its `write()` method. The callback extension create an instance of the appropriate OVM type. This is typically done by using an appropriate data conversion class (see 5.5).

5.10.5 Collaboration

A VMM transactor calls a callback method, providing a transaction descriptor as one or many of its argument. An instance of the analysis port adapter callback extension is registered with the transactor and gets invoked. The callback information is converted to the appropriate OVM type and published on the analysis port by calling its `write()` method. The OVM subscriber at the other end obtains it through its `write()` method.

5.10.6 Implementation

To implement the practice, the integrator creates a new extension of the VMM transactor callback facade class, instantiates an analysis port of the appropriate type and implements the relevant virtual method.

The implementation of the callback method creates an instance of the appropriate OVM type, either through explicit construction or via a data conversion method, then publishes the OVM data onto the analysis port via its `write()` method.

The analysis port of the callback extension is then connected to the appropriate OVM analysis export of the desired component(s).

5.10.7 Sample code

This illustrates how to implement this practice.

```
typedef avt_converter #(vmm_apb_tr, ovm_apb_tr) apb_rw_vmm2ovm;

class to_analysis_port extends apb_master_cbs;
    analysis_port #(ovm_apb_rw) ap;

    function new(ovm_component parent);
       ap = new("ap", parent);
    endfunction
```

```
    virtual task post_tr(apb_master xactor,
                         vmm_apb_rw     tr);
       ovm_apb_rw o_tr;
       o_tr = apb_rw_vmm2ovm::convert(tr);
       ap.write(o_tr);
    endtask
endclass
```

a) To connect a VMM callback to multiple OVM recipients in an OVM container, use:

```
apb_master        vmm_bfm;
my_apb_scoreboard ovm_sbd1, ovm_sbd2;
to_analysis_port  v2o;
...
function void build();
   vmm_bfm  = new("APB BFM");
   ovm_sbd1 = new("sbd1", this);
   ovm_sbd2 = new("sbd2", this);
   v2o      = new(this);
   vmm_bfm.append_callback(v2o);
endfunction

function void connect();
      v2o.ap.connect(ovm_sbd1.analysis_export);
      v2o.ap.connect(ovm_sbd2.analysis_export);
endfunction
```

b) To connect a VMM callback to multiple OVM recipients in a VMM environment container, use:

```
apb_master        vmm_bfm;
my_apb_scoreboard ovm_sbd1, ovm_sbd2;
to_analysis_port    v2o;

function new(string inst, virtual apb_if.passive sigs);
   super.new("VMM/OVM SubEnv", inst);
   vmm_bfm = new("APB BFM", sigs);
   ovm_sbd1 = new("sbd1", ovm_root);
   ovm_sbd2 = new("sbd2", ovm_root);
   v2o      = new(ovm_root);

   v2o.ap.connect(ovm_sbd1.analysis_export);
   v2o.ap.connect(ovm_sbd2.analysis_export);
endfunction
```

5.11 Sequence and scenario composition

5.11.1 Practice name

Sequential stimulus or Multi-stream sequences

5.11.1.1 Also known as

N/A.

5.11.1.2 Related practices

OVM-on-top phase synchronization (see 5.1) and *VMM-on-top phase synchronization* (see 5.2).

5.11.2 Intent

This practice shows how to execute VMM scenarios from OVM sequences and vice-versa.

5.11.2.1 Motivation

Stimulus often involves coordinating the operation of multiple transaction streams, often on multiple interfaces. It is necessary to allow both OVM sequences to control VMM (multi-stream) scenarios and VMM multi-stream scenarios to control OVM (virtual) sequences.

5.11.2.2 Consequences

None.

5.11.3 Applicability

This practice is used to implement a hierarchical OVM (virtual) sequence or VMM multi-stream scenario that coordinates the execution of OVM sequences and/or VMM (multi-stream) scenarios.

NOTE—As mentioned in Chapter 4, it may be easier to add additional (OVM) sequences or (VMM) scenarios to an existing VIP component using its native methodology rather than trying to emulate the behavior in the parent methodology.

5.11.4 Structure

This practice uses methods that already exist in each base library. There are no new components or adapters required to implement this practice.

5.11.5 Collaboration

The collaboration for this practice varies depending on which methodology is on top.

5.11.5.1 OVM-on-top

In the OVM-on-top case, proper coordination of VMM scenarios, which are instantiated in VMM scenario generators, requires the integrator to ensure that the generator is not otherwise executing the scenario. This may require an OVM sequence or the OVM environment to stop the VMM scenario generator by calling its `stop_xactor()` method.

a) An OVM sequence can start a VMM single-stream scenario by calling the VMM scenario's `apply()` method. The OVM `` `ovm_do* `` macros may not be used to execute VMM scenarios.
b) An OVM sequence can start a VMM multi-stream scenario by calling its `execute()` method. The following steps shall be completed, by the existing VMM VIP or by the OVM container during the build phase.
    1) A VMM multi-stream scenario generator has to be instantiated.
    2) The multi-stream scenarios to be started from an OVM sequence have to be instantiated and registered with the (multi-stream scenario) generator.
    3) The channels used by the multi-stream scenarios have to be registered with the generator.

The verification environment integrator shall ensure the required class handles for the respective VMM components (e.g., VMM generators and possibly VMM scenarios, VMM multi-stream scenarios and VMM channels) are accessible from the OVM sequences. These class handles should be initialized in the `connect()` phase of the environment (**avt_ovm_vmm_env** [see 6.4]) to point to the VMM components that need to be controlled by the OVM sequences. 5.11.5.2 illustrates how to accomplish this.

5.11.5.2 VMM-on-top

In the VMM-on-top case, the guidelines specified in 5.2 should be followed to ensure the OVM components are built and connected correctly. A VMM multi-stream scenario may execute an OVM sequence by calling its `start()` method. The sequencer that runs this sequence is specified as an argument of the call to `start()`. The VMM multi-stream scenario may not call the OVM `` `ovm_do* `` macros to invoke OVM sequences. The verification environment integrator needs to ensure the sequencer handles are initialized to point to the correct OVM sequencer(s); this initialization should be done during the `build()` phase, after the multi-stream scenario has been created and the OVM component has been built. 5.11.7.2 shows the sample code for this case.

In the VMM-on-top case, the following steps shall be completed by the existing OVM VIP or by the VMM container during the build phase.

a) An OVM sequencer has to be instantiated.

b) The OVM sequencer shall be connected to the appropriate driver.

The verification environment integrator shall ensure the required class handles for the respective OVM components (e.g., OVM sequencer(s) and possibly OVM sequences) are accessible from the VMM multi-stream scenario generator. These class handles should be initialized in the `build()` phase of the environment (**avt_vmm_ovm_env** [see 6.5]) to point to the OVM components that need to be controlled by the VMM multi-stream scenarios. 5.11.7.2 illustrates how to accomplish this.

5.11.6 Implementation

See 5.11.7.

5.11.7 Sample code

This illustrates how to implement this practice.

5.11.7.1 OVM-on-top

This example shows how to perform the following operations from an OVM sequence.

— Grab/ungrab VMM channels to control the execution of VMM-generated transactions,

— Start the VMM scenarios, including multi-stream scenarios, and send the generated transactions to the VMM drivers (via the VMM channels).

OVM (virtual) sequences are used to coordinate the operation of multiple transaction streams on multiple interfaces. It is important that OVM sequences can coordinate the traffic generation of OVM components, as well as VMM components.

The following example illustrates the OVM-on-top case. In this example, the OVM environment `ovm_vmm_tb` consists of many OVM components and the following VMM components:

— Two APB drivers: `vmmdriver0` and `vmmdriver1`

— Two VMM channels: `apb_vmmch0` and `apb_vmmch1`

— A multi-stream scenario generator: `vmmmss_gen0`

— Some multi-stream scenarios: `vmmmss1` and `vmmmss2`

— One APB scenario: `apb_vmmscen1`

To keep the sample code readable, only the details necessary to understand the integration of the VMM components into an OVM verification environment are shown here. It is presumed the readers are familiar

with both the OVM and VMM libraries, and the recommended OVM verification component (OVC) architecture described in the *OVM User Guide* ([B2]).

```
class ovm_vmm_seq extends ovm_sequence;
…
   vmm_ms_scenario_gen vmmmss_gen
   apb_scenario1 apb_vmmscen1;
   apb_ovm_init_seq ovm_init_seq;
   int unsigned num_items;
   int mss_no_items;

   virtual task body();
      vmm_channel apb_vmmch0 = vmmmss_gen.get_channel("ch0");
      vmm_ms_scenario vmmmss1 = vmmmss_gen.get_scenario("vmmmss1");

      fork
         begin

            apb_vmm_ch0.grab(vmmmss1);
            ovm_init_seq.start(sqr1);
            …
            apb_vmm_ch0.ungrab(vmmmss1);
         end
         begin
      // start a VMM scenario
            assert(apb_vmmscen1.randomzize() with {…});
            apb_vmmscen1.apply(apb_vmmch0, num_items);
            …
            //Now, start a VMM multi-stream scenario
            vmmmss1.execute(mss_no_items);
         end
      join
      …
   endtask : body
endclass

class ovm_vmm_tb extends ovm_env;
// OVM sequence
   ovm_vmm_seq my_ovm_vmm_seq;
   apb_ovm_init_seq ovm_init_seq;

//VMM components
   apb_driver  vmmdriver0, vmmdriver1;
   apb_channel apb_vmmch0, apb_vmmch1;
   apb_scenario1 apb_vmmscen1;
   vmm_ms_scenario_gen vmmmss_gen0;
   ms_scenario1 vmmmss1;
   …
   extern virtual function void build();
   extern virtual function void connect();
endclass

function void ovm_vmm_tb::build();
   //Allocate the VMM components
   apb_vmmch0 = new("apb_vmmch0", apb_vmmch0);
   vmmdriver0 = new("vmmdriver0", 0, top.apb_if0, apb_vmmch0);
   …
   vmmmss1 = new();
```

```
    apb_vmmscen1 = new();
    // Assign handles in OVM sequence
    ovm_init_seq = apb_ovm_init_seq::type_id::create("init_seq");
    my_ovm_vmm_seq.ovm_init_seq = ovm_init_seq;
    my_ovm_vmm_seq.apb_vmmscen1 = apb_vmmscen1;
    //Multi-stream scenarios and channels registrations
    vmmmss_gen0.register_channel("ch0", this.apb_vmmch0);
    vmmmss_gen0.register_channel("ch1", this.apb_vmmch1);
    vmmmss_gen0.register_ms_scenario("vmmmss1", vmmmss1);
    vmmmss_gen0.register_ms_scenario("vmmmss2", vmmmss2);
            …
    my_ovm_vmm_seq.vmmmss_gen = vmmmss_gen0;
    vmmmss_gen0.stop_after_n_scenarios = 5;
      …
endfunction : build

function void ovm_vmm_tb::connect();
    //Connections for remaining OVM components
    …
endfunction : connect
```

5.11.7.2 VMM-on-top

In a VMM-on-top verification environment, VMM multi-stream scenarios are used to coordinate the operation of multiple transaction streams on multiple interfaces. It is important that VMM multi-stream scenarios can coordinate the traffic generation of VMM components, as well as OVM components, including starting the OVM sequences and virtual sequences.

The following example illustrates the VMM-on-top case. In this example, the VMM environment `avt_ovm_vmm_env` consists of many VMM components and an Ethernet OVC, which consists of OVM subcomponents, such as: Ethernet agents, Ethernet sequencers, Ethernet drivers, etc.

To keep the sample code readable, only the details necessary to understand the integration of the Ethernet OVC into the VMM verification environment and the code to start the OVM sequence from a multi-stream scenario are shown here. It is presumed the readers are familiar with both the OVM and the VMM.

```
class my_vmm_ovm_env extends avt_vmm_ovm_env;
    `ovm_build
    apb_driver  vmmdriver0, vmmdriver1;
        apb_channel apb_vmmch0, apb_vmmch1;
        vmm_ms_scenario_gen vmmmss_gen0;
        mss_ovm_sequence mss_ovm_sequence_1;// multi-stream scenario that
                                             // calls OVM sequence

        ethernet_ovc ovm_ethernet_ovc;

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

        extern function void build();
        ...
endclass

    function void avt_ovm_vmm_env::build();
        super.build();
        …
```

```
        //Multi-stream scenario allocation & mss registration
        mss_gen0.register_channel("ch0",this.apb_vmmch0);
        mss_gen0.register_channel("ch1",this.apb_vmmch1);
        mss_ovm_sequence_1= new();
        mss_gen0.register_ms_scenario("mss_ovm_sequence_1",
                                      mss_ovm_sequence_1);
        mss_gen0.stop_after_n_scenarios = 3;
        // Instantiation other VMM components
        ….
        //Build the OVM component
        ovm_ethernet_ovc = new("ovm_ethernet_ovc");
        ovm_build();
        //Initialize the sequencer handle in multi-stream scenario
        mss_ovm_sequence_1.ovm_seqr = ovm_ethernet_ovc.agent.sequencer;
        // disable ovm_ethernet_ovc.agent.sequencer
    endfunction : build

class mss_ovm_seq extends vmm_ms_scenario;
    rand apb_txn txn;
    ovm_sequencer#(ethernet_packet) ovm_seqr;//OVM sequencer handle,
                                            //initialized in avt_ovm_vmm_env
    ethernet_error_sequence ethernet_sequence_i; //ovm sequence to be executed

    function new(vmm_scenario parent = null);
        super.new(parent);
        ethernet_sequence_i = new("ethernet_sequence_i");
    endfunction : new

    virtual task execute(ref int n);
    …
        fork
            begin
            //Start OVM sequence
                ethernet_sequence_i.grab(ovm_seqr);
                ethernet_sequence_i.start(ovm_seqr);
                ethernet_sequence_i.ungrab(ovm_seqr);
                …
            end
            begin
                //start traffic on other VMM drivers
                ch1.put(this.txn.copy());
                …
                ch0.put(this.txn.copy());
                …
            end
        join
    endtask : execute
endclass
```

5.12 Messaging

5.12.1 Practice name

Messaging

5.12.1.1 Also known as

N/A.

5.12.1.2 Related practices

Used by *OVM-on-top phase synchronization* (see 5.1) and *VMM-on-top phase synchronization* (see 5.2).

5.12.2 Intent

All messages issued by OVM and VMM components should have a similar appearance and integrated accounting. The maximum verbosity level to be displayed should be set globally and affect any messages issued from OVM and VMM components.

5.12.2.1 Motivation

It is easier for users visually scanning messages or parsing message log files if all messages have a consistent format.

Error messages issued from OVM or VMM components shall be combined for accounting purposes to optionally stop the simulation after a user-defined number of errors has been reached. Similarly, a fatal error message issued from an OVM or VMM component shall be globally accounted to abort the simulation.

5.12.2.2 Consequences

It is necessary to map the message types, severities, and verbosities between VMM and OVM. VMM uses eleven message types and seven orthogonal severity levels. OVM uses four message severities and an orthogonal 32-bit integer numeric verbosity value. Typically, only a subset of all possible combinations of message types, severities, and verbosities is used.

Table 6 shows how typical VMM messages are mapped when they are converted into OVM messages.

Table 6—OVM-on-top message mapping

VMM		OVM	
Severity	**Type**	**Severity**	**Verbosity**
`FATAL_SEV`	`FAILURE_TYP`	`OVM_FATAL`	`OVM_NONE`
	other	`OVM_INFO`	
`ERROR_SEV`	`FAILURE_TYP`	`OVM_ERROR`	`OVM_LOW`
	other	`OVM_INFO`	
`WARNING_SEV`	`FAILURE_TYP`	`OVM_WARNING`	`OVM_MEDIUM`
	other	`OVM_INFO`	
`NORMAL_SEV`	any	`OVM_INFO`	`OVM_MEDIUM`
`TRACE_SEV`	any		`OVM_HIGH`
`DEBUG_SEV`	any		`OVM_FULL`
`VERBOSE_SEV`	any		`OVM_DEBUG`

Table 7 shows how typical OVM messages are mapped when they are converted into VMM messages.

Table 7—VMM-on-top message mapping

OVM		VMM	
Severity	**Verbosity**	**Type**	**Severity**
OVM_WARNING	any	FAILURE_TYP	WARNING_SEV
OVM_ERROR		FAILURE_TYP	ERROR_SEV
OVM_FATAL		FAILURE_TYP	FATAL_SEV
OVM_INFO	<= OVM_NONE	NOTE_TYP	ERROR_SEV
	<= OVM_LOW		WARNING_SEV
	<= OVM_MEDIUM		NORMAL_SEV
	<= OVM_HIGH		TRACE_SEV
	<= OVM_FULL		DEBUG_SEV
	<= OVM_DEBUG		VERBOSE_SEV

Because messages are still issued using their respective native message interface and are funneled to the top-level message reporting service behind-the-scenes, any filtering that occurs in the source methodology is still applied first. Therefore, messages issued in the foreign methodology are subject to the sum of the filters in both methodologies. To avoid over-filtering foreign messages, the message adapter in the host methodology does not perform (by default) any filtering on foreign messages.

5.12.3 Applicability

This practice is used whenever mixed environments are created. In a VMM-on-top situation, messages are formatted and counted by the *VMM Message Service*. In an OVM-on-top situation, messages are formatted and counted by the *OVM Global Report Server*.

5.12.4 Structure

The messages issued in any of the foreign message report interfaces are captured by a single message adapter in the host methodology, as illustrated in Figure 21.

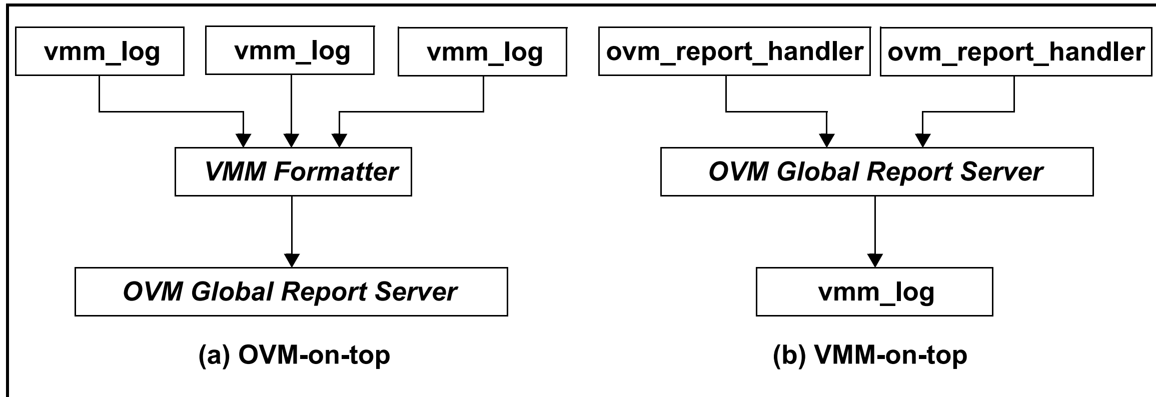

Figure 21—Message interoperability dataflow

5.12.4.1 Class diagram

There are no user-visible classes specific to messaging interoperability.

5.12.4.2 Declaration prototype

There are no user-visible APIs specific to messaging interoperability.

5.12.4.3 Participants

None.

5.12.5 Collaboration

None.

5.12.6 Implementation

To ensure messages are integrated before being issued by the foreign methodology, the user shall specify which methodology to use for messaging by setting the `OVM_ON_TOP` or `VMM_ON_TOP` compiler directive, via the command line. Only one of these directives shall be defined.

5.12.7 Sample code

These examples shows how messages issued from OVM and VMM components in an arbitrary hierarchy can be globally or individually controlled. It is necessary to incorporate these code examples in larger self-running examples to see how the messages are formatted and accounted by the host methodology.

5.12.7.1 OVM-on-top

```
ovm_top.set_report_max_quit_count(15);

// Increase verbosity on the VMM leaf on the 2nd set
#10;
env.ovm.leaf.log.set_verbosity(vmm_log::VERBOSE_SEV);

// Increase verbosity on the OVM leaf on the 3rd set
// And turn down the VMM leaf and middle to a trickle
```

```
#10;
env.vmm.leaf.set_report_verbosity_level(OVM_DEBUG);
env.ovm.leaf.log.set_verbosity(vmm_log::ERROR_SEV);

// Globally turn off OVM messages and crank up the VMM ones
// (which should turn off everything because VMM messages
// are routed through OVM).
#10;
env.vmm.log.set_verbosity(vmm_log::VERBOSE_SEV, "/./", "/./");
ovm_top.set_report_verbosity_level_hier(OVM_LOW);
```

Simulation output

```
OVM_INFO ovm_leaf.sv(40) @ 26: vmm.leaf [ovm_leaf] High-Info message from
   vmm.leaf
OVM_INFO ovm_leaf.sv(42) @ 26: vmm.leaf [ovm_leaf] Info message from vmm.leaf
OVM_WARNING ovm_leaf.sv(44) @ 26: vmm.leaf [ovm_leaf] Warning message from
   vmm.leaf
OVM_ERROR ovm_leaf.sv(46) @ 26: vmm.leaf [ovm_leaf] Error message from vmm.leaf
==============================================


OVM_ERROR ovm_mid.sv(58) @ 31: ovm [ovm_mid] Error message from ovm
OVM_ERROR @ 32: VMM Leaf [ovm.leaf] Error message from ovm.leaf
OVM_ERROR @ 35: VMM Mid [vmm] Error message from vmm

--- OVM Report Summary ---

Quit count reached!
Quit count :          15 of          15
** Report counts by severity

OVM_INFO :   92
OVM_WARNING :   11
OVM_ERROR :   15
OVM_FATAL :    0
** Report counts by id
[COMPPH]    40
[ENDPH]     9
[RNTST]     1
[STARTPH]    10
[ovm.leaf]    11
[ovm_leaf]    18
[ovm_mid]    19
[vmm]    10
```

5.12.7.2 VMM-on-top

```
env.log.stop_after_n_errors(15);

// Increase verbosity on the OVM leaf on the 2nd set
#10;
env.vmm.leaf.set_report_verbosity_level(OVM_FULL);

// Increase verbosity on the VMM leaf on the 3rd set
// And turn down the OVM leaf and middle to a trickle
#10;
env.ovm.leaf.log.set_verbosity(vmm_log::VERBOSE_SEV);
```

```
env.vmm.leaf.set_report_verbosity_level(OVM_LOW);
env.ovm.set_report_verbosity_level(OVM_LOW);

// Globally turn off VMM messages and crank up the OVM ones
// (which should turn off everything because OVM messages
// are routed through VMM).
#10;
ovm_top.set_report_verbosity_level(OVM_DEBUG);
env.log.set_verbosity(vmm_log::ERROR_SEV, "/./", "/./");
```

Simulation output

```
Normal[NOTE] on VMM Mid(vmm) at                   25:
    Note message from vmm
WARNING[FAILURE] on VMM Mid(vmm) at                   25:
    Warning message from vmm
!ERROR![FAILURE] on VMM Mid(vmm) at                   25:
    Error message from vmm
!ERROR![FAILURE] on OVM(reporter) at                   26:
    vmm.leaf(ovm_leaf): Error message from vmm.leaf
==============================================

!ERROR![FAILURE] on OVM(reporter) at                   31:
    ovm(ovm_mid): Error message from ovm
!ERROR![FAILURE] on VMM Leaf(ovm.leaf) at                   32:
    Error message from ovm.leaf
!ERROR![FAILURE] on VMM Mid(vmm) at                   35:
    Error message from vmm
Maximum number of error messages exceeded. Aborting
Use method stop_after_n_errors() of vmm_log to increase threshold.
Simulation *FAILED* on /./ (/./) at                 35: 15 errors, 10 warnings
```

6. Application programming interface (API)

This chapter defines the application programming interface (API) for each of the VIP classes. Each API is based on the SystemVerilog syntax of IEEE Std 1800™.

6.1 Common parameters

Many of the VIP classes contain some or all of the following parameters as part of their APIs.

6.1.1 OVM

The OVM transaction type; this needs to be an extension of `ovm_transaction` or `ovm_sequence_item`.

6.1.2 OVM_REQ

The OVM request transaction type for a bidirectional request/response communication component, such as **avt_tlm2channel** (see 6.6); this needs to be an extension of `ovm_transaction` or `ovm_sequence_item`.

6.1.3 OVM_RSP

The OVM response transaction type for a bidirectional request/response communication component, such as **avt_tlm2channel** (see 6.6); this needs to be an extension of `ovm_transaction` or `ovm_sequence_item`.

6.1.4 VMM

The VMM transaction type; this needs to be an extension of `vmm_data`.

6.1.5 VMM_REQ

The VMM request transaction type for a bidirectional request/response communication component, such as **avt_tlm2channel** (see 6.6); this needs to be an extension of `vmm_data`.

6.1.6 VMM_RSP

The VMM response transaction type for a bidirectional request/response communication component, such as **avt_tlm2channel** (see 6.6); this needs to be an extension of `vmm_data`.

6.1.7 OVM2VMM

The converter class to go from OVM to VMM. The converter class shall implement a single static method having the following prototype:

```
static function VMM convert(OVM in, VMM to=null);
```

If the `to` argument is provided, the OVM transaction contents are copied into the existing `to` VMM transaction. Otherwise, a new VMM transaction is allocated, copied into, and returned. See also 6.2.

6.1.8 OVM2VMM_REQ

The converter class (see 6.1.7) used to convert *request* transaction descriptors from OVM to VMM.

6.1.9 OVM2VMM_RSP

The converter class (see 6.1.7) used to convert *response* transaction descriptors from OVM to VMM.

6.1.10 VMM2OVM

The converter class to go from VMM to OVM. The converter class shall implement a single static method having the following prototype:

```
static function OVM convert(VMM in, OVM to=null);
```

If the `to` argument is provided, the VMM transaction contents are copied into the existing `to` OVM transaction. Otherwise, a new OVM transaction is allocated, copied into, and returned. See also 6.2.

6.1.11 VMM2OVM_REQ

The converter class (see 6.1.10) used to convert *request* transaction descriptors from VMM to OVM.

6.1.12 VMM2OVM_RSP

The converter class (see 6.1.10) used to convert *response* transaction descriptors from VMM to OVM.

6.2 avt_converter #(IN,OUT)

This pass-though converter simply returns the input transaction of type **IN** as a return value of type **OUT**. **avt_converter** has the following declaration, parameters, and methods.

NOTE—Typically users create their own converter (see 5.5).

6.2.1 Declaration

This class is declared as follows.

```
class avt_converter #( type  IN  =  int,
                                     OUT  =  IN );
```

6.2.2 Parameters

This class contains the following parameters.

6.2.2.1 IN

This is the input type, which can be any user-defined class.

6.2.2.2 OUT

This the output type, which needs to be assignment-compatible with the input type (see 6.2.2.1).

6.2.3 Methods

This class contains the following method.

```
static function OUT convert(  IN  in,
                              OUT  to  =  null );
```

This returns the input argument of type **IN** as type **OUT**, which shall be assignment-compatible with type **IN** (see 6.2.2.1). For class types, this means **OUT** shall be the same type as **IN** or a super-class of **IN**.

The `to` argument allows the conversion to copy into an existing object and avoid the expense of allocation.

If `to` is *null* (the default), the convert method shall create a new instance of **OUT**, copy the fields of `in` to it, and return it. If the `to` argument is non-null, the convert method should copy the fields of `in` to the corresponding fields of `to` and then return `to`.

6.3 avt_match_ovm_id

This simple comparator class provides a default implementation of the static `match()` method used by the **avt_channel2tlm** (see 6.7) adapter class.

6.3.1 Declaration

This class is declared as follows.

```
class avt_match_ovm_id;
```

6.3.2 Parameters

This class is not parameterized.

6.3.3 Methods

This class contains the following method.

```
static function bit match(ovm_sequence_item req,
                          ovm_sequence_item rsp);
   return req.get_sequence_id() == rsp.get_sequence_id() &&
          req.get_transaction_id() == rsp.get_transaction_id();
endfunction
```

This method returns a `1` if both the `sequence_id` and `transaction_id` members of the `req` transaction descriptor match the specified `rsp` transaction descriptor. Otherwise, it returns `0`.

6.4 avt_ovm_vmm_env

This class is an **ovm_component** that automatically creates the **vmm_env** class during construction. `avt_ovm_vmm_env` has the following hierarchy, declaration, methods, and variables.

6.4.1 Hierarchy

This class has the hierarchy shown in Figure 22.

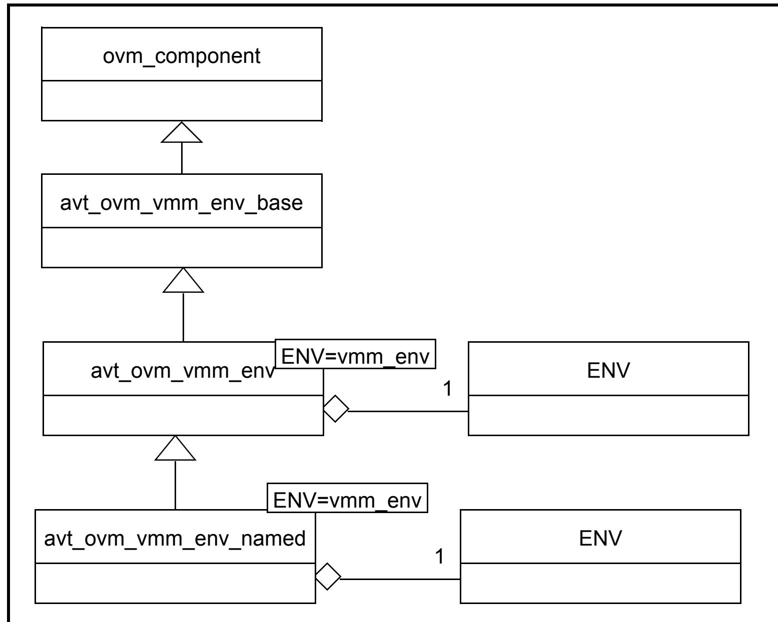

Figure 22—avt_ovm_vmm_env

6.4.2 Declaration

This class is declared as follows.

```
class avt_ovm_vmm_env #(type ENV=vmm_env) extends avt_ovm_vmm_env_base;

class avt_ovm_vmm_env_named #(type ENV=vmm_env) extends avt_ovm_vmm_env_base;
```

The `avt_ovm_vmm_env_named` derived class is used when the instantiated `vmm_env` uses a `name` argument in its constructor.

6.4.3 Methods

This class contains the following methods.

6.4.3.1 new

```
function new ( string name,
                          ovm_component parent = null,
                          vmm_env env = null )
```

This creates a `vmm_env` container component with the given `name` and `parent`. A new instance of a `vmm_env` of type `ENV` is created if one is not provided in the `env` argument. The environment is not named. In the `avt_ovm_vmm_env_named` derived class, the arguments are the same, but if the environment is provided, its name value is set to `{parent.get_full_name(),"."},name}`.

6.4.3.2 vmm_gen_cfg

```
virtual function void gen_cfg()
```

Calls the underlying VMM environment's gen_cfg() method. The user may extend **vmm_gen_cfg** to add additional functionality. In this case, the user calls super.vmm_gen_cfg() to execute the underlying environment's gen_cfg() method.

6.4.3.3 build

```
virtual function void build()
```

Calls the underlying VMM environment's build method. The user may extend **build** to add additional functionality. In this case, the user calls super.build() to execute the underlying environment's build() method.

6.4.3.4 vmm_reset_dut

```
virtual task reset_dut()
```

Calls the underlying VMM environment's reset_dut method, followed by the ovm_top.stop_request() method. The user may extend **reset_dut** to add additional functionality. In this case, the user calls super.reset_dut() to execute the underlying environment's reset_dut() method.

6.4.3.5 vmm_cfg_dut

```
virtual task vmm_cfg_dut()
```

Calls the underlying VMM environment's cfg_dut method followed by the ovm_top.stop_request() method. The user may extend **cfg_dut** to add additional functionality. In this case, the user calls super.cfg_dut() to execute the underlying environment's cfg_dut() method.

6.4.3.6 run

```
task avt_ovm_vmm_env::run()
```

Calls the underlying VMM environment's reset_dut, cfg_dut, start, and wait_for_end methods, returning when the environment's end-of-test condition has been reached. Extensions of this method may augment or remove certain end-of-test conditions from the underlying environment's consensus object before calling super.run(). When super.run() returns, extensions may choose to call ovm_top.stop_request() if the underlying environment is the only governor of end-of-test. Extensions may completely override this base implementation by not calling super.run. In such cases, all four VMM phases shall still be executed explicitly by the user in the prescribed order.

If **auto_stop_request** is set (see 6.4.4.2), OVM's stop_request() is called to end the run phase.

6.4.3.7 stop

```
virtual task stop (string ph_name)
```

When called during the OVM run phase, this task waits for the underlying environment's wait_for_end phase to return, then calls the VMM environment's stop and cleanup tasks. When the **ok_to_stop** variable (see 6.4.4.1) is set at the time stop is called, stop does not wait for wait_for_end to return. This allows OVM components to control when the VMM environment and its embedded xactors are stopped.

6.4.3.8 vmm_report

```
virtual task vmm_report()
```

Calls the underlying VMM environment's report method, then stops the `report_vmm` phase. This phase is called after OVM's `report` phase has completed. The user may extend **vmm_report** to add additional functionality. In this case, the user calls `super.vmm_report()` to execute the underlying environment's `vmm_report()` method.

6.4.4 Variables

This class contains the following variables.

6.4.4.1 ok_to_stop

```
bit ok_to_stop = 0
```

When **ok_to_stop** is clear (default), the **avt_ovm_vmm_env**'s `stop` task waits for the VMM environment's `wait_for_end` task to return before continuing. This bit is automatically set with the underlying VMM environment's `wait_for_end` task returns, which allows the stop task to call the VMM environment's `stop` and `cleanup` phases.

If **ok_to_stop** is set manually, other OVM components may terminate the `run` phase before the VMM environment has returned from `wait_for_end`.

6.4.4.2 auto_stop_request

```
bit auto_stop_request = 0
```

When set, this bit enables calling an OVM `stop_request` after the VMM environment's `wait_for_end` task returns, thus ending OVM's `run` phase coincident with VMM's `wait_for_end`. The default is `0`.

Now, a wrapped VMM environment is a subcomponent of a larger-scale OVM environment (that may incorporate multiple wrapped VMM environments). A VMM environment's end-of-test condition is no longer sufficient for determining the overall end-of-test condition. Thus, the default value for **auto_stop_request** is `0`. Parent components of the VMM environment wrapper may choose to wait on the posedge of **ok_to_stop** (see 6.4.4.1) to indicate the VMM environment has reached its end-of-test condition.

6.5 avt_vmm_ovm_env

This class is used to automatically integrate OVM phasing with VMM phasing in a VMM-on-top environment. **avt_vmm_ovm_env** has the following hierarchy, declaration, methods, and macros.

6.5.1 Hierarchy

This class has the hierarchy shown in Figure 23.

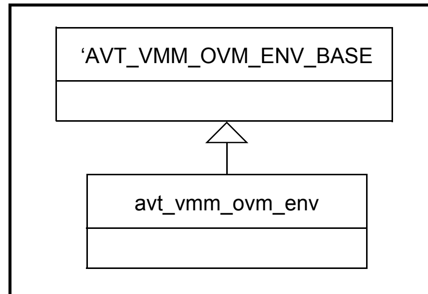

Figure 23—avt_vmm_ovm_env

By default, the **AVT_VMM_OVM_ENV_BASE** compiler variable is set to `vmm_env`.

6.5.2 Declaration

This class is declared as follows.

```
class avt_vmm_ovm_env extends `AVT_VMM_OVM_ENV_BASE;
```

6.5.3 Methods

This class contains the following methods.

6.5.3.1 new

```
function new(string name = "Verif Env"
             `VMM_ENV_BASE_NEW_EXTERN_ARGS);
```

This creates a new instance of **avt_vmm_ovm_env**.

6.5.3.2 ovm_build

```
virtual function void ovm_build()
```

This calls into the OVM's phasing mechanism to complete OVM's build, connect, and any other user-defined phases up to `end_of_elaboration`.

6.5.3.3 reset_dut

```
virtual task reset_dut()
```

This synchronizes the start of VMM `reset_dut` with the start of OVM `run` phase, then forks the OVM `run` phase to run in parallel with `reset_dut`, `config_dut`, `start`, and `wait_for_end`.

6.5.3.4 stop

```
virtual task stop ()
```

This requests the OVM `run` phase to stop if it is still running, then waits for the OVM `run` phase to finish.

6.5.3.5 report

```
virtual task report()
```

This calls into the OVM's phasing mechanism to execute user-defined OVM phases inserted after `report_ph`, if any.

6.5.4 Macros

This class contains the following macro.

`ovm_build

This declares the `ovm_build()` method, which calls into the OVM's phasing mechanism to complete OVM's `build`, `connect`, and any other user-defined phases up to `end_of_elaboration`. This macro must be specified in every extension of **avt_vmm_ovm_env** to ensure that `ovm_build()` is only executed once.

6.6 avt_tlm2channel

Use this class to connect an OVM producer to a VMM consumer via a `vmm_channel`. Consumers can implement many different response-delivery models. See also 5.6.

6.6.1 Hierarchy

The inheritance hierarchy of the **avt_tlm2channel** class is shown in Figure 24.

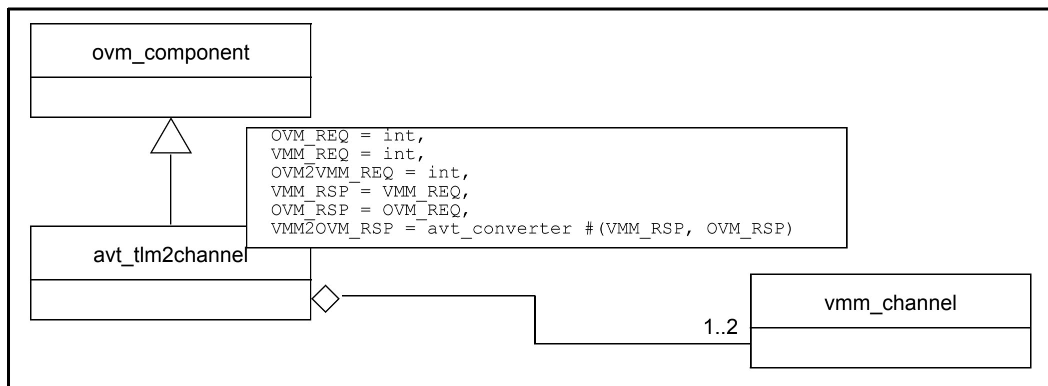

Figure 24—avt_tlm2channel

6.6.2 Declaration

This class is declared as follows.

```
class avt_tlm2channel #(type OVM_REQ     = int,
                             VMM_REQ     = int,
                             OVM2VMM_REQ = int,
                             VMM_RSP     = VMM_REQ,
                             OVM_RSP     = OVM_REQ,
                             VMM2OVM_RSP = avt_converter #(VMM_RSP,OVM_RSP))
                                 extends ovm_component;
```

6.6.3 Parameters

This class contains the following parameters (see 6.1).

- **OVM_REQ**
- **VMM_REQ**
- **OVM2VMM_REQ**
- **VMM_RSP**
- **OVM_RSP**
- **VMM2OVM_RSP**

6.6.4 Communication interfaces

The **avt_tlm2channel** communication adapter includes both OVM and VMM communication interfaces.

The OVM interfaces are implemented as TLM `port`s or `export`s.

a) `seq_item_port` This bidirectional port is used to connect to an `ovm_sequencer` or any other component providing an `ovm_seq_item_export`.

b) `put_export` This export is used to receive transactions from an OVM producer that utilizes a blocking or non-blocking put interface.

c) `master_export` This bidirectional export is used to receive requests from and deliver responses to an OVM producer that utilizes a blocking or non-blocking master interface.

d) `blocking_transport_export` This bidirectional export is used to receive requests from and deliver responses to an OVM producer that utilizes a blocking transport interface.

e) `blocking_get_peek_port` This unidirectional port is used to retrieve responses from a passive OVM producer with a blocking `get_peek` export.

f) `blocking_put_port` This port is used to deliver responses to an OVM producer that expects responses from a blocking put interface.

g) `blocking_slave_port` This bidirectional port is used to request transactions from and deliver responses to a passive OVM producer utilizing a blocking slave interface.

h) `request_ap` All transaction requests received from any of the interface ports and exports in this adapter are broadcast out this analysis port to any OVM subscribers.

i) `response_ap` All transaction responses received from any of the interface ports and exports in this adapter are broadcast out this analysis port to any OVM subscribers.

The VMM interfaces are implemented via built-in `vmm_channel` objects.

j) `req_chan` Handle to the request `vmm_channel_typed #(VMM_REQ)` instance being adapted.

k) `rsp_chan` Handle to the optional response `vmm_channel_typed #(VMM_RSP)` instance being adapted.

6.6.5 Methods

The only user-accessible method of the **avt_tlm2channel** component is the `constructor`. All TLM and `vmm_channel` communication methods shall be called via the appropriate TLM port or `vmm_channel` reference, respectively. This class contains the following method.

```
function new(string name="avt_tlm2channel",
             ovm_component parent=null,
             vmm_channel_typed #(VMM_REQ) req_chan=null,
             bit wait_for_req_ended=0);
```

This creates a new instance of **avt_tlm2channel**. If the `vmm_channel` arguments are not supplied, they are created internally.

6.6.6 Variables

This class contains the following variables.

6.6.6.1 wait_for_req_ended

```
protected bit wait_for_req_ended = 0;
```

When the VMM consumer does not use a separate response channel, this bit specifies whether the response, which is annotated into the original request, is available after a get from the request channel (`wait_for_req_ended=0`) or after the original request's `ENDED` status is indicated (`wait_for_req_ended=1`).

This bit may be specified via a constructor argument and/or by using a `set_config_int()` call targeting the desired component, using **wait_for_req_ended** as the second argument to specify the variable to be set.

6.6.6.2 request_timeout

```
time request_timeout = 100us;
```

When **wait_for_req_ended** is set (see 6.6.6.1), this specifies the time-out value to wait for the response before a warning is issued.

6.7 avt_channel2tlm

Use this class to connect a VMM producer to an OVM consumer. Consumers can implement many different response-delivery models. See also 5.7.

6.7.1 Hierarchy

The inheritance hierarchy of the **avt_channel2tlm** class is shown in Figure 25.

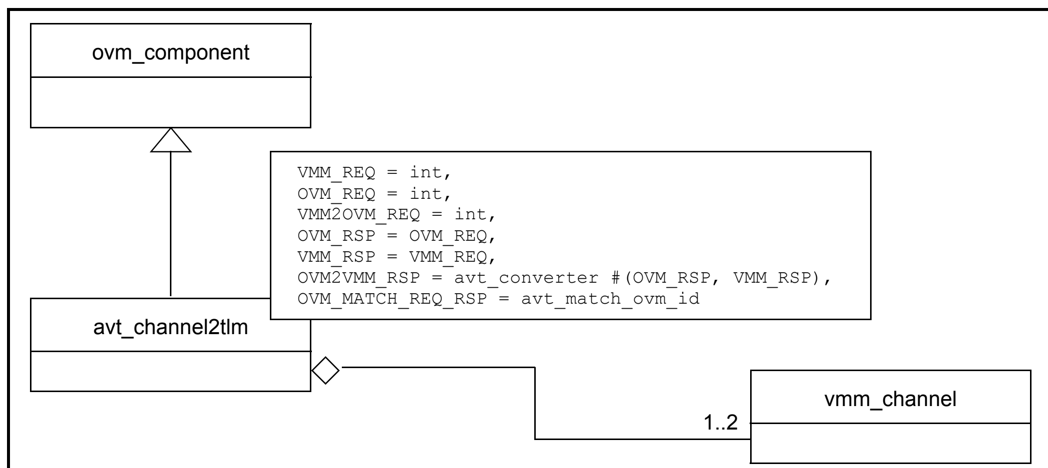

Figure 25—avt_channel2tlm

6.7.2 Declaration

This class is declared as follows.

```
class avt_channel2tlm #(type VMM_REQ     = int,
                             OVM_REQ     = int,
                             VMM2OVM_REQ = int,
                             OVM_RSP     = OVM_REQ,
                             VMM_RSP     = VMM_REQ,
                             OVM2VMM_RSP = avt_converter #(OVM_RSP,VMM_RSP),
                             OVM_MATCH_REQ_RSP=avt_match_ovm_id)
                                          extends ovm_component;
```

6.7.3 Parameters

This class contains the following parameters (see 6.1).

- **VMM_REQ**
- **OVM_REQ**
- **VMM2OVM_REQ**
- **OVM_RSP**
- **VMM_RSP**
- **OVM2VMM_RSP**
- **OVM_MATCH_REQ_RSP**

The comparator class to compare the OVM `req` transaction descriptor to the OVM `rsp`. The comparator class shall implement a single static method having the following prototype:

```
static function bit match(ovm_sequence_item req,
                          ovm_sequence_item rsp);
```

The method shall return `1` if the two transaction descriptors are deemed to match, otherwise it shall return `0`. See also 6.3.

6.7.4 Communication interfaces

The **avt_channel2tlm** communication adapter includes both OVM and VMM communication interfaces.

The OVM interfaces are implemented as TLM `port`s and `export`s.

a) `seq_item_export` Used by OVM driver consumers using the sequencer interface to process transactions.
b) `get_peek_export` For OVM consumers getting requests via peek/get.
c) `response_export` For OVM consumers returning responses via analysis write.
d) `put_export` For OVM consumers returning responses via blocking put.
e) `slave_export` For sending requests to passive OVM consumers via blocking put.
f) `blocking_put_port` For sending requests to passive OVM consumers via blocking put.
g) `blocking_transport_port` For atomic execution with passive OVM consumers via blocking transport.
h) `blocking_slave_port` For driving passive OVM consumers via blocking slave interface.

i) request_ap All requests are broadcast out this analysis port after successful extraction from the request vmm_channel.

j) response_ap All responses sent to the response channel are broadcast out this analysis port.

The VMM interfaces are implemented via built-in vmm_channel objects.

k) req_chan Handle to the request vmm_channel_typed #(VMM_REQ) instance being adapted.

l) rsp_chan Handle to the optional response vmm_channel_typed #(VMM_RSP) instance being adapted.

6.7.5 Methods

The only user-accessible method of the **avt_channel2tlm** component is the constructor. All TLM and vmm_channel communication methods shall be called via the appropriate TLM port or vmm_channel reference, respectively.

This class also contains the following method.

```
function new (string name="avt_channel2tlm",
              ovm_component parent=null,
              vmm_channel_typed #(VMM_REQ) req_chan=null,
              vmm_channel_typed #(VMM_RSP) rsp_chan=null,
              bit rsp_is_req=1,
              int unsigned max_pending_req=100);
```

This creates a new instance of **avt_channel2tlm**. If either of the req_chan or rsp_chan arguments are not supplied, they are created internally. The rsp_is_req and max_pending_req arguments are used optionally to set values for important user-visible variables (see 6.7.6.1 and 6.7.6.2, respectively).

6.7.6 Variables

This class contains the following variables.

6.7.6.1 rsp_is_req

```
protected bit rsp_is_req = 1;
```

This indicates whether a response is the same object as the request with the status and/or read data filled in. When set, and the rsp_chan is *null*, the request process, after returning from a put to the request channel, copies the VMM request into the original OVM request object and sends it as the OVM response to the seq_item_port's put method.

In certain vmm_channel/driver completion models, the channel's full level is 1 and the connected driver does not consume the transaction until it has been fully executed. In this mode, the driver peeks the transaction from the channel, executes it, fills in the response in fields of the same request object, then finally pops (gets) the transaction off the channel. This then frees the put process, which was waiting for the transaction to leave the channel.

This variable can be specified in a constructor argument (see 6.7.5) and/or by using a set_config_int() call targeting the desired component, using **rsp_is_req** as the second argument to specify the variable to be set.

6.7.6.2 max_pending_requests

```
int unsigned max_pending_req = 100;
```

This specifies the maximum number of requests that can be outstanding. The adapter holds all outgoing requests in a queue for later matching with incoming responses. A maximum exists to prevent this queue from growing too large.

This variable can be specified in a constructor argument (see 6.7.5) and/or by using a `set_config_int()` call targeting the desired component, using **max_pending_requests** as the second argument to specify the variable to be set.

6.8 avt_analysis_channel

Use this class to connect any OVM component with an analysis port or export to any VMM component via a `vmm_channel`.

6.8.1 Hierarchy

This class has the hierarchy shown in Figure 26.

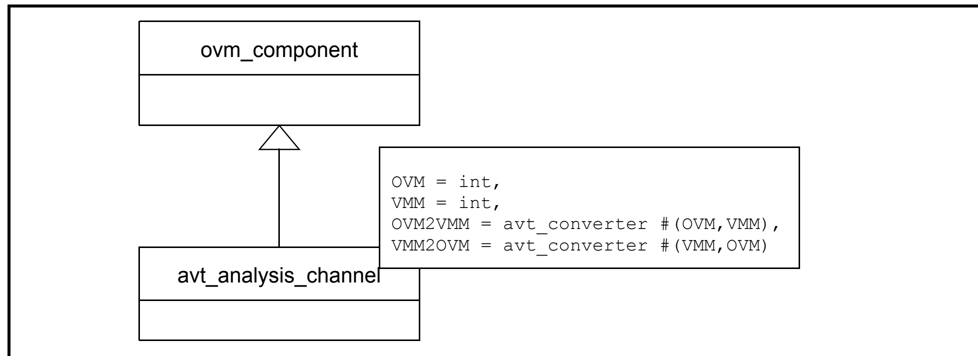

Figure 26—avt_analysis_channel

6.8.2 Declaration

This class is declared as follows.

```
class avt_analysis_channel #(type OVM = int, VMM = int,
                              OVM2VMM = avt_converter #(OVM,VMM),
                              VMM2OVM = avt_converter #(VMM,OVM))
                                 extends ovm_component;
```

6.8.3 Parameters

This class contains the following parameters (see 6.1).

— **OVM**

— **VMM**

— **OVM2VMM**

— **VMM2OVM**

6.8.4 Communication interfaces

The **avt_analysis_channel** communication adapter includes both OVM and VMM communication interfaces.

The OVM interfaces are implemented as TLM `ports` and `exports`.

a) `analysis_port` VMM transactions received from the channel are converted to OVM transactions and broadcast out this analysis port.

b) `analysis_export` The adapter may receive OVM transactions via this analysis export.

The VMM interfaces are implemented via built-in `vmm_channel` objects.

c) `chan` Handle to the request `vmm_channel_typed #(VMM)` instance being adapted.

6.8.5 Methods

The only user-accessible method of the **avt_analysis_channel** component is the `constructor`.

This class also contains the following method.

```
function new (string name, ovm_component parent=null,
              vmm_channel_typed #(VMM) chan=null);
```

This creates a new **avt_analysis_channel** with the given `name` and optional `parent`; the optional `chan` argument provides the handle to the `vmm_channel` being adapted. If no channel is given, the adapter creates one.

6.9 avt_analysis2notify

The **avt_analysis2notify** adapter receives OVM data from its `analysis_export`, converts it to VMM, then indicates the configured event notification, passing the converted data as `vmm_data`-based status. VMM components that have registered a callback for the notification receive the converted data.

6.9.1 Hierarchy

The inheritance hierarchy of the **avt_analysis2notify** class is shown in Figure 27.

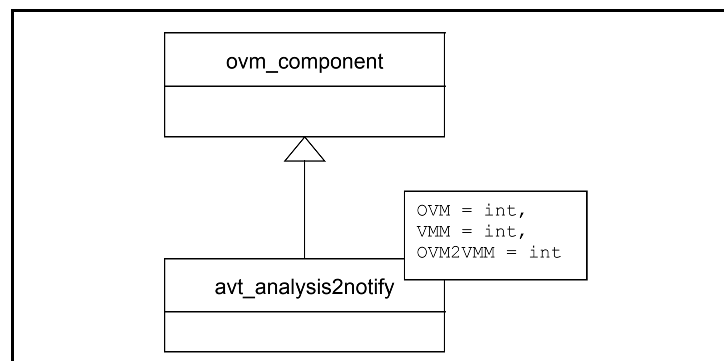

Figure 27—avt_analysis2notify

6.9.2 Declaration

This class is declared as follows.

```
class avt_analysis2notify #(type OVM=int,
                            VMM=int,
                            OVM2VMM=int) extends ovm_component;
```

6.9.3 Parameters

This class contains the following parameters (see 6.1).

— **OVM**

— **VMM**

— **OVM2VMM**

6.9.4 Communication interfaces

The **avt_analysis2notify** adapter includes both OVM and VMM communication interfaces.

The OVM interface is implemented as a TLM `analysis_export`. OVM transactions written to this export is converted to VMM and embedded in the built-in `notify` object that gets *indicated* by the adapter.

6.9.5 Methods

The only user-accessible method of the **avt_analysis2notify** component is the `constructor`.

This class also contains the following method.

```
function new(string name,
             ovm_component parent=null,
             vmm_notify notify=null,
             int notification_id=-1);
```

This creates a new analysis-to-notify adapter with the given `name` and optional `parent`; the `notify` and `notification_id` together specify the notification event this adapter notes upon receipt of a transaction on its `analysis_export`.

If the `notify` handle is not supplied or *null*, the adapter creates one and assigns it to the `notify` property. If the `notification_id` is not provided, the adapter configures a `ONE_SHOT` notification and assigns it to the `RECEIVED` property.

6.9.6 Variables

This class contains the following variables.

6.9.6.1 notify

```
vmm_notify notify;
```

This is the `notify` object this adapter uses to indicate the **RECEIVED** event notification (see 6.9.6.2).

6.9.6.2 RECEIVED

```
int RECEIVED;
```

This is the notification id this adapter notes upon receipt of OVM data from its `analysis_export`.

6.10 avt_notify2analysis

The **avt_notify2analysis** adapter receives an indication from VMM, converts the data to OVM, then publishes the data to its `analysis_port`.

6.10.1 Hierarchy

The inheritance hierarchy of the **avt_notify2analysis** class is shown in Figure 28.

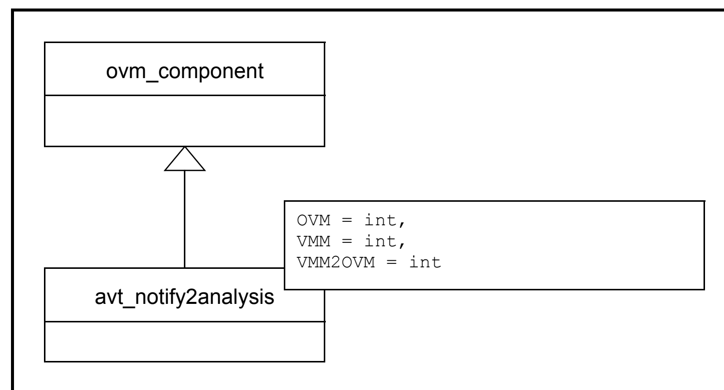

Figure 28—avt_notify2analysis

6.10.2 Declaration

This class is declared as follows.

```
class avt_notify2analysis #(type VMM=int,
                             OVM=int,
                             VMM2OVM=int)
                                     extends ovm_component;
```

6.10.3 Parameters

This class contains the following parameters (see 6.1).

— **OVM**

— **VMM**

— **VMM2OVM**

6.10.4 Communication interfaces

The **avt_notify2analysis** adapter includes both OVM and VMM communication interfaces.

The OVM interface is implemented as a TLM `analysis_port`. When the notify is indicated, the adapter converts the indicated VMM transaction to the appropriate OVM transaction and then publishes the OVM transaction via the `analysis_port`'s `write()` method.

6.10.5 Methods

The only user-accessible method of the **avt_notify2analysis** component is the `constructor`.

This class also contains the following method.

```
function new(string name,
             ovm_component parent=null,
             vmm_notify notify=null,
             int notification_id=-1);
```

This creates a new notify-to-analysis adapter with the given `name` and optional `parent`; the `notify` and `notification_id` together specify the notification event this adapter notes upon receipt of a transaction on its `analysis_export`.

If the `notify` handle is not supplied or *null*, the adapter creates one and assigns it to the `notify` property. If the `notification_id` is not provided, the adapter configures a `ONE_SHOT` notification and assigns it to the `RECEIVED` property.

6.10.6 Variables

This class contains the following variables.

6.10.6.1 notify

```
vmm_notify notify;
```

This is the `notify` object this adapter uses to listen to the **RECEIVED** event notification (see 6.10.6.2).

6.10.6.2 RECEIVED

```
int RECEIVED;
```

This is the notification id this adapter listens to for receipt of VMM data from its `status` of the notification.

Appendix A

(informative)

Bibliography

[B1] Open SystemC Initiative (OSCI), Transaction Level Modeling (TLM) Library, Release 1.0.

[B2] *OVM User Guide* (part of the following Internet location: http://www.ovmworld.org).

[B3] For a summary of OVM, see the following Internet location: http://www.ovmworld.org.

[B4] For a summary of VMM, see the following Internet location: http://www.vmmcentral.org.