



CUSTOMER EDUCATION SERVICES

SystemVerilog Verification UVM Workshop

Student Guide

40-I-055-SSG-007 2019.06

Synopsys Customer Education Services
690 E. Middlefield Road
Mountain View, California 94043

Workshop Registration: <https://training.synopsys.com>

Copyright Notice and Proprietary Information

© 2019 Synopsys, Inc. All rights reserved. This software and documentation contain confidential and proprietary information that is the property of Synopsys, Inc. The software and documentation are furnished under a license agreement and may be used or copied only in accordance with the terms of the license agreement. No part of the software and documentation may be reproduced, transmitted, or translated, in any form or by any means, electronic, mechanical, manual, optical, or otherwise, without prior written permission of Synopsys, Inc., or as expressly provided by the license agreement.

Destination Control Statement

All technical data contained in this publication is subject to the export control laws of the United States of America. Disclosure to nationals of other countries contrary to United States law is prohibited. It is the reader's responsibility to determine the applicable regulations and to comply with them.

Disclaimer

SYNOPSYS INC., AND ITS LICENSORS MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Trademarks

Synopsys and certain Synopsys product names are trademarks of Synopsys, as set forth at <https://www.synopsys.com/company/legal/trademarks-brands.html>. All other product or company names may be trademarks of their respective owners.

Third-Party Links

Any links to third-party websites included in this document are for your convenience only. Synopsys does not endorse and is not responsible for such websites and their practices, including privacy practices, availability, and content.

Synopsys, Inc.
690 E. Middlefield Road
Mountain View, CA 94043
www.synopsys.com

Document Order Number: 40-I-055-SSG-007
SystemVerilog Verification UVM Workshop Student Guide

Table of Contents

Unit i: Introduction

Introductions	i-2
Facilities.....	i-3
Course Materials	i-4
Workshop Overview	i-5
Workshop Goals.....	i-6
Workshop Target Audience	i-7
Workshop Prerequisite Knowledge	i-8
Curriculum Flow	i-9
Agenda	i-10
Agenda	i-11
Agenda	i-12
What Is the Device Under Test?	i-13
A Functional Perspective	i-14
Input Packet Structure.....	i-15
Input Packet Structure.....	i-16
Reset Signal	i-17
Icons Used in this Workshop	i-18

Unit 1: OOP Inheritance Review

Agenda	1-1
Unit Objectives	1-2
Object Oriented Programming (OOP): Class	1-3
OOP: Inheritance	1-4
OOP: Inheritance	1-5
OOP: Polymorphism.....	1-6
OOP: Polymorphism.....	1-7
OOP: Polymorphism.....	1-8
OOP: Polymorphism.....	1-9
Unit Objectives Review	1-10
Appendix.....	1-11
Parameterized Classes.....	1-12
typedef.....	1-13
Methods Outside of the Class	1-14
Static Property.....	1-15
Static Method.....	1-16
Singleton Objects	1-17
Simple Singleton Proxy Object Example	1-18
Applying Proxy Class in Factory (1/5)	1-19
Applying Proxy Class in Factory (2/5)	1-20
Applying Proxy Class in Factory (3/5)	1-21

Table of Contents

Applying Proxy Class in Factory (4/5)	1-22
Applying Proxy Class in Factory (5/5)	1-23
Singleton Object Core Service (1/2).....	1-24
Singleton Object Core Service (2/2).....	1-25

Unit 2: UVM Structural Overview

Agenda	2-1
Unit Objectives	2-2
UVM - Universal Verification Methodology	2-3
Origin of UVM	2-4
Verification Goal	2-5
Coverage-Driven Verification	2-6
Phases of Verification	2-7
Run More Tests, Write Less Code.....	2-8
UVM Class Tree (Partial)	2-9
Typical Testbench Architecture	2-10
UVM Testbench Architecture.....	2-11
UVM Structure is Scalable	2-12
Structural Class Support in UVM	2-13
Structural Functional Support - Phasing	2-14
UVM Hello World Example	2-15
Compile and Simulate.....	2-16
Inner Workings of UVM Simulation	2-17
User Report Messages.....	2-18
UVM Simple Structure Example	2-19
Starting UVM Execution	2-20
Structural Classes - Test.....	2-21
Structural Classes - Environment.....	2-22
Structural Classes - Agent.....	2-23
Structural Classes - Driver.....	2-24
Structural Classes - Debug.....	2-25
Print Format Can Be Specified	2-26
General Debugging with Report Messages.....	2-27
Default Simulation Handling	2-28
Command Line Control of Report Messages.....	2-29
User Filterable Code Block.....	2-30
Test For Understanding.....	2-31
Lab 1: Managing UVM Report Messages	2-32
Key Structural Concept: Parent-Child	2-33
Key Component Concepts: Logical Hierarchy	2-34
Key Component Concepts: Phase	2-35
Key Component Concepts: Override	2-36
Unit Objectives Review	2-37

Table of Contents

Appendix.....	2-38
Migration from UVM1.1 to UVM1.2	2-39
Global Handles.....	2-40
Sequences.....	2-41
Phasing.....	2-42
uvm_config_db	2-43
Reporting.....	2-44
Objection Performance	2-45
Appendix.....	2-46
Calling UVM Messaging From Modules (1/2).....	2-47
Calling UVM Messaging From Modules (2/2).....	2-48
Appendix.....	2-49
Catch and Throw UVM Report Messages.....	2-50
Appendix.....	2-51
Embed UVM Reporter Control in Test.....	2-52
Appendix.....	2-53
UVM Command Line Processor.....	2-54
Appendix.....	2-55
Compiling UVM with VCS	2-56

Unit 3: UVM Transaction

Agenda	3-1
Unit Objectives	3-2
UVM Transaction Base Classes.....	3-3
UVM Transaction Flow	3-4
Modeling Transactions.....	3-5
Other Properties to be Considered (1/2)	3-6
Other Properties to be Considered (2/2)	3-7
Transactions: Must-Obey Constraints.....	3-8
Transactions: Should-Obey Constraints	3-9
Transactions: Constraint Considerations	3-10
Transaction Class Methods	3-11
Customization of Field Processing Methods	3-12
Field Automation Enabled Field Processing.....	3-13
'uvm_field_* Field Automation Macros	3-14
Print Radix Specified by FLAG.....	3-15
Examples of Using Transaction Methods	3-16
Modify Constraint in Transactions by Type	3-17
Transaction Replacement Results.....	3-18
Modify Constraint in Transaction by Instance.....	3-19
Command-line Override	3-20
Parameterized Transaction Class (1/3)	3-21
Parameterized Transaction Class (2/3)	3-22

Table of Contents

Parameterized Transaction Class (3/3)	3-23
Simplifying Report Messages	3-24
Applying convert2string().....	3-25
Unit Objectives Review	3-26
Appendix.....	3-27
uvm_object_utils Macro	3-28
Appendix.....	3-29
uvm_object Class Common Members	3-30

Unit 4: UVM Sequence

Agenda	4-1
Unit Objectives	4-2
UVM Transaction Flow - Continued	4-3
Stimulus Generation Classes.....	4-4
Sequence Class.....	4-5
Generate Transactions in Sequence Class.....	4-6
User Can Manually Create and Send Item.....	4-7
'uvm_do Macro Interaction Detailed.....	4-8
Sequence Execution: Starting a Sequence	4-9
start() Method in Sequence Class	4-10
Sequence Execution Methodologies	4-11
Explicit Sequence Execution	4-12
Implicit Sequence Execution (1/2).....	4-13
Implicit Sequence Execution (2/2).....	4-14
Implicit Sequence Execution and Objection.....	4-15
UVM-1.1 Sequence Phase Objection	4-16
UVM-1.2 & IEEE UVM Sequence Phase Objection	4-17
Code That Can Work in UVM-1.1 and UVM-1.2	4-18
Lab 2: Generate Stimulus.....	4-19
Sequence with randc Transaction Property.....	4-20
Creating a Sequence of Related Items	4-21
Nested Sequences	4-22
Implicit Sequence Execution Overrides.....	4-23
Implicit Sequence Execution at Phases.....	4-24
Unit Objectives Review	4-25
Appendix.....	4-26
Sequence Priority/Weight	4-27
Appendix.....	4-28
Sequencer-Driver Response Port	4-29
Appendix.....	4-30
Sequence with Out-Of-Order Response (1/2).....	4-31
Sequence with Out-Of-Order Response (2/2).....	4-32
Out-Of-Order Driver (1/2)	4-33
Out-Of-Order Driver (2/2)	4-34

Table of Contents

Unit 5: UVM Configuration & Factory

Agenda	5-1
Unit Objectives	5-2
UVM Component Base Class Structure	5-3
Component Parent-Child Relationships.....	5-4
Display & Querying	5-5
Query Hierarchy Relationship	5-6
Use Logical Hierarchy in Configuration.....	5-7
Component Configuration Example	5-8
UVM Resource	5-9
Manage DUT Interface Configuration.....	5-10
Test Configures Agents with Interfaces.....	5-11
Configuring Component's Interface (1/2)	5-12
Configuring Component's Interface (2/2)	5-13
Additional Needs: Manage Test Variations	5-14
Test Requirements: Transaction	5-15
Test Requirements: Components	5-16
Factories in UVM	5-17
Transaction Factory	5-18
UVM Factory Transaction Creation	5-19
Component Factory.....	5-20
UVM Factory Component Creation.....	5-21
Override in Test	5-22
Command-Line Override	5-23
Visually Inspect Factory Overrides (1/2).....	5-24
Visually Inspect Factory Overrides (2/2).....	5-25
Parameterized Component Class	5-26
Unit Objectives Review	5-27
Appendix.....	5-28
Run-time Configuration Switch.....	5-29
Retrieve int Configuration Field via User Macro	5-30
Appendix.....	5-31
Configuring Sequences (Instance-Based)	5-32
Configuring Sequences (Class-Based).....	5-33
Configuring Sequences (Sequencer-Based).....	5-34
Configuring Sequences (Agent-Based).....	5-35
Appendix.....	5-36
Configuring Array Members.....	5-37
Configuring Array.....	5-38
Appendix.....	5-39
Configure enum Field via string	5-40
Appendix.....	5-41
UVM DVE Configuration Debugging.....	5-42
UVM Verdi Configuration Debugging.....	5-43

Table of Contents

Appendix.....	5-44
UVM Field Macro Auto Configuration Issues	5-45
Disabling Auto Configuration Retrieval: UVM-1.1 & 1.2	5-46
Disabling Auto Configuration Retrieval: IEEE UVM (1/2).....	5-47
Disabling Auto Configuration Retrieval: IEEE UVM (2/2).....	5-48

Unit 6: UVM Component Communication

Agenda	6-1
Unit Objectives	6-2
Component Communication: Overview	6-3
Component Communication: Method Based.....	6-4
Component Communication: TLM.....	6-5
Component Communication: TLM.....	6-6
Communication in UVM: TLM 1.0, 2.0.....	6-7
UVM TLM 1.0.....	6-8
Push Mode	6-9
Pull Mode.....	6-10
FIFO Mode.....	6-11
Analysis Port.....	6-12
Port Pass-Through.....	6-13
UVM TLM 2.0.....	6-14
Blocking Transport Initiator	6-15
Blocking Transport Target.....	6-16
Non-Blocking Transport Initiator	6-17
Non-Blocking Transport Target.....	6-18
Unit Objectives Review	6-19
Lab 3: Driving the DUT.....	6-20
Appendix.....	6-21
TLM 2.0 Generic Payload (1/4).....	6-22
TLM 2.0 Generic Payload (2/4).....	6-23
TLM 2.0 Generic Payload (3/4).....	6-24
TLM 2.0 Generic Payload (4/4).....	6-25
Appendix.....	6-26
Verdi UVM Debug Switches	6-27
UVM-Aware Features in Verdi	6-28
UVM Transaction and Log Debug	6-29
Object Hierarchy Browser	6-30
UVM Factory Debug	6-31
UVM Resource Debug.....	6-32
UVM Phase-Based Breakpoint.....	6-33
UVM Phase Objection Debug	6-34
UVM Sequence Debug	6-35
UVM Register Debug	6-36

Table of Contents

Unit 7: UVM Scoreboard & Coverage

Agenda	7-1
Unit Objectives	7-2
Scoreboard - Introduction	7-3
Scoreboard – Data Streams	7-4
Scoreboard Implementation	7-5
Scoreboard Transaction Source: Monitor	7-6
Embed Monitor in Agent	7-7
UVM Agent Example	7-8
Using UVM Agent in Environment	7-9
Scoreboard Can Be Parameterized.....	7-10
Scoreboard: User Implementation (1/2).....	7-11
Scoreboard: User Implementation (2/2).....	7-12
Functional Coverage	7-13
Connecting Coverage to Testbench	7-14
Component Configuration Coverage (1/2)	7-15
Component Configuration Coverage (2/2)	7-16
Stimulus Coverage	7-17
Correctness Coverage	7-18
Unit Objectives Review	7-19
Appendix.....	7-20
Scoreboard: Multi-Stream.....	7-21
Scoreboard: Multi-Stream.....	7-22

Unit 8: UVM Callback

Agenda	8-1
Unit Objectives	8-2
Changing Behavior of Components.....	8-3
Implementing Simple Callback Operations	8-4
Implementing UVM Callbacks	8-5
Step 1: Embed Callback Methods.....	8-6
Step 2: Declare the façade Class.....	8-7
Step 3: Implement Callback: Error	8-8
Step 4: Create and Register Callback Objects	8-9
Driver Coverage Example.....	8-10
Implement Coverage via Callback.....	8-11
Create and Register Callback Objects.....	8-12
Sequence Simple Callback Methods.....	8-13
Unit Objectives Review	8-14
Lab 4: Full Testbench with Monitors and Scoreboard.....	8-15

Table of Contents

Unit 9: UVM Advanced Sequence/Sequencer

Agenda	9-1
Unit Objectives	9-2
Managing Sequence Execution.....	9-3
Managing Sequence Execution.....	9-4
Top Sequence.....	9-5
Top Sequencer	9-6
Executing Top Sequence.....	9-7
Set Top Sequencer Content.....	9-8
Sequence Execution Management	9-9
Synchronization Mechanism: uvm_event.....	9-10
Synchronization Mechanism: uvm_event.....	9-11
Specialized Pools for Synchronization	9-12
Trigger Global Reset Event (Level Sensitive)	9-13
uvm_event_pool Example (Level Sensitive).....	9-14
Exclusive Sequencer Access (lock/unlock)	9-15
Exclusive Sequencer Access (grab/ungrab).....	9-16
Unit Objectives Review	9-17
Appendix.....	9-18
Sequence Arbitration and Priority (1/2).....	9-19
Sequence Arbitration and Priority (2/2).....	9-20
Code Example.....	9-21
Appendix.....	9-22
Reactive Sequences.....	9-23
Reactive TLM Port Setup	9-24
Reactive Sequence Request	9-25
Reactive Sequence Response	9-26
Appendix.....	9-27
Interrupt Sequences.....	9-28
Detect Interrupt	9-29
React to Interrupt	9-30
Appendix.....	9-31
Component Synchronization: uvm_barrier.....	9-32

Unit 10: UVM Phasing and Objections

Agenda	10-1
Unit Objectives	10-2
UVM Component Phasing.....	10-3
Common Phases.....	10-4
Run-Time Task Phases	10-5
Task Phase Synchronization	10-6
Phase Objection – Device Drivers (1/6)	10-7

Table of Contents

Phase Objection – Sequence (2/6)	10-8
Phase Objection – Drain Time (3/6)	10-9
Phase Objection – Scoreboard (4/6)	10-10
Phase Objection – Test (5/6).....	10-11
Phase Objection - Debug (6/6).....	10-12
UVM Timeout.....	10-13
Advanced Features.....	10-14
Phase Domains (1/2).....	10-15
Phase Domains (2/2).....	10-16
User Defined Phase.....	10-17
Phase Jump: Backward	10-18
Phase Jump: Forward	10-19
Phase Jumping Cleanup	10-20
Get Phase Execution Count	10-21
Unit Objectives Review	10-22
Lab 5: Top Sequencer and Sequence	10-23
Appendix.....	10-24
uvm_phase Class Key Methods	10-25
uvm_phase Class States	10-26
Appendix.....	10-27
Debugging Objections with Callbacks.....	10-28
Appendix.....	10-29
Driver Guideline	10-30
Monitor Guideline.....	10-31
Agent Guideline	10-32
Scoreboard Guideline.....	10-33
Environment Phase Guideline.....	10-34
Test Phase Guideline.....	10-35
Appendix.....	10-36
Driver Reset Guideline (1/3).....	10-37
Driver Reset Guideline (2/3).....	10-38
Driver Reset Guideline (3/3).....	10-39
Transaction Monitor Reset Guideline (1/2).....	10-40
Transaction Monitor Reset Guideline (2/2)	10-41
Reset Monitor Guideline.....	10-42
Agent Reset Guideline	10-43
Scoreboard Reset Guideline.....	10-44
Environment Reset Guideline	10-45
Sequence Reset Guideline (1/2).....	10-46
Sequence Reset Guideline (2/2).....	10-47
Sequencer Reset Guideline	10-48

Table of Contents

Unit 11: UVM Register Abstraction Layer (RAL)

Agenda	11-1
Unit Objectives	11-2
Register & Memories	11-3
Testbench without UVM Register Abstraction.....	11-4
Testbench with UVM Register Abstraction.....	11-5
UVM Register Abstraction	11-6
Implement UVM Register Abstraction.....	11-7
Example Specification	11-8
Step 1: Create Host Data & Driver	11-9
Step 1: Create Host Sequence	11-10
Verify Frontdoor Host is Working.....	11-11
Step 2: Create .ralf File Based on Spec.....	11-12
Step 2: Create .ralf File Based on Spec.....	11-13
UVM Register Abstraction File (.ralf) Syntax.....	11-14
UVM Register Abstraction: Field.....	11-15
Field Access Types	11-16
UVM Register Abstraction File (.ralf) Syntax.....	11-17
UVM Register Abstraction: Register.....	11-18
UVM Register Abstraction File (.ralf) Syntax.....	11-19
UVM Register Abstraction File (.ralf) Syntax.....	11-20
UVM Register Abstraction File (.ralf) Syntax.....	11-21
UVM Register Abstraction: Block.....	11-22
UVM Register Abstraction File (.ralf) Syntax.....	11-23
UVM Register Abstraction: System	11-24
Step 3: Create UVM Register Abstraction Model	11-25
Step 4: Create UVM Register Adapter	11-26
Sequencer Adapter Class (1/2).....	11-27
Sequencer Adapter Class (2/2).....	11-28
Optional Feature in Adapter Class	11-29
Step 5: Instantiating UVM Register Model	11-30
Tie Sequencer Adapter to Register Map.....	11-31
UVM Register Abstraction Sequence	11-32
Run RAL Sequence Implicitly or Explicitly.....	11-33
UVM Register Test Sequences	11-34
Execute RAL Test Sequence	11-35
Enabling Auto Mirror Prediction	11-36
Explicit (Manual) Mirror Prediction	11-37
Connecting Explicit Mirror Predictor	11-38
Unit Objectives Review	11-39
Appendix.....	11-40
UVM Register Modes	11-41
Register Frontdoor Write	11-42
Register Frontdoor Read.....	11-43

Table of Contents

Register Backdoor Write.....	11-44
Register Backdoor Read	11-45
Register Backdoor Poke.....	11-46
Register Backdoor Peek.....	11-47
Mirrored & Desired Property Update	11-48
UVM Register Desired Property Write.....	11-49
Randomize UVM Register Desired Property	11-50
UVM Register Desired Property Read	11-51
Mirrored & DUT Value Update.....	11-52
Writing to uvm_reg Mirrored Property.....	11-53
Reading uvm_reg Mirrored Property.....	11-54
Appendix.....	11-55
Memory Frontdoor Write.....	11-56
Memory Frontdoor Read.....	11-57
Memory Backdoor Write	11-58
Memory Backdoor Read.....	11-59
Memory Backdoor Poke	11-60
Memory Backdoor Peek	11-61
Appendix.....	11-62
ralgen Options: Common.....	11-63
ralgen Options: Advanced.....	11-64
ralgen Address Granularity	11-65
ralgen Functional Coverage	11-66
Appendix.....	11-67
Optional: Backdoor Access.....	11-68
Modified ralgen XMR Backdoor Access (1/3).....	11-69
ralgen XMR Backdoor Access (2/3).....	11-70
ralgen XMR Backdoor Access (3/3).....	11-71
ralgen DPI Backdoor Access (1/2)	11-72
ralgen DPI Backdoor Access (2/2)	11-73
Appendix.....	11-74
UVM Register Base Class Commonly Used Content.....	11-75
UVM Register Classes: uvm_reg_bus_op & uvm_reg_item.....	11-76
UVM Register Classes: uvm_reg_field	11-77
UVM Register Classes: uvm_reg.....	11-78
UVM Register Classes: uvm_reg_map	11-79
UVM Register Classes: uvm_reg_block	11-80
Appendix.....	11-81
RAL Class Key Callback Members	11-82
RAL Class Key Callback Members	11-83
Appendix.....	11-84
Changing the Address offsets of a Domain	11-85
Appendix.....	11-86
Direct DUT Signal Access.....	11-87

Table of Contents

Unit 12: Summary

Agenda	12-1
Key Elements of UVM	12-2
UVM Methodology Guiding Principles.....	12-3
Scalable Architecture	12-4
Standardized Component Communication	12-5
Customizable Component Phase Execution	12-6
Flexible Components Configuration (1/2).....	12-7
Flexible Components Configuration (2/2).....	12-8
Flexible Component Search & Replace	12-9
Standardized Register Abstraction.....	12-10
UVM Command Line Options.....	12-11
Getting Help.....	12-12
Lab 6: Implement RAL.....	12-13
That's all Folks!	12-14

Unit CS: Customer Support

Synopsys Support Resources	CS-2
SolvNet Online Support.....	CS-3
SolvNet Registration.....	CS-4
Support Center	CS-5
Other Technical Sources	CS-6
Summary: Getting Support	CS-7

SystemVerilog Verification with UVM

VCS 2019.06

Verdi 2019.06

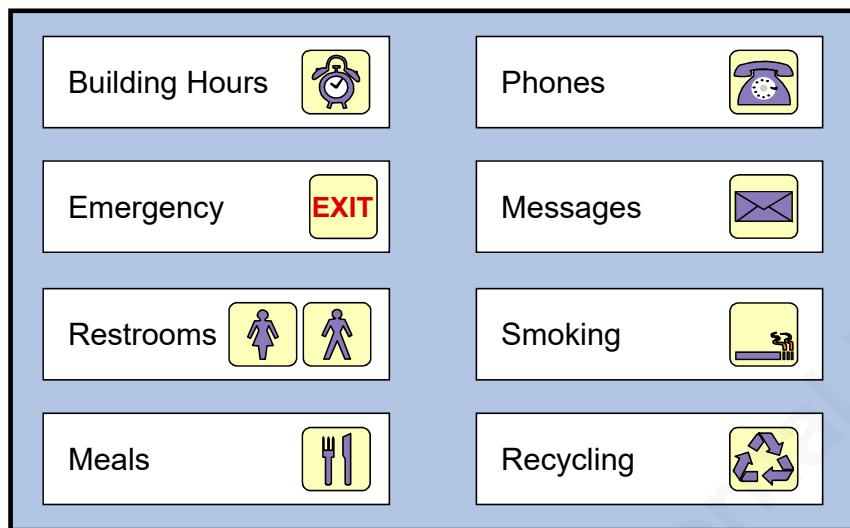


Introductions

- Name
- Company
- Job Responsibilities
- Relevant Experience
- Main Goal(s) and Expectations for this Course

i-2

Facilities



Please turn off cell phones and pagers

i-3

Course Materials

- Student Workbook
- Lab Book
- Reference Materials
- Course Evaluations

i-4

Workshop Overview

■ This workshop covers UVM-1.1, UVM-1.2 and IEEE UVM

- A lot of UVM testbenches are still running on UVM-1.1
 - ◆ There are compatibility issues (will be covered)
- For the labs:
 - ◆ "make" uses UVM-1.2 source code in vcs
 - ◆ "make uvm_ver=uvm-1.1" uses UVM-1.1d source code in vcs

■ IEEE UVM

- Officially approved
- Accellera IEEE UVM Rev 1.0
 - ◆ Can be found in lab directory: ces_uvm-1.2_2018.09/1800.2-2017-1.0
 - ◆ Please read ces_uvm-1.2_2018.09/labs/README file for instruction in executing IEEE UVM

■ Major differences between UVM-1.1, 1.2 and IEEE are described in the lecture and labs

i-5

Workshop Goals



- Develop a scalable and reusable UVM test environment
- Manage test environment with `uvm_config_db` class
- Create test stimulus with UVM sequence class
- Execute UVM sequences in UVM tests
- Modify tests with UVM factory class
- Manage debugging messages with UVM reporter class

i- 6

Workshop Target Audience

**Design or Verification engineers
writing SystemVerilog testbenches
to verify Verilog or VHDL designs**



i-7

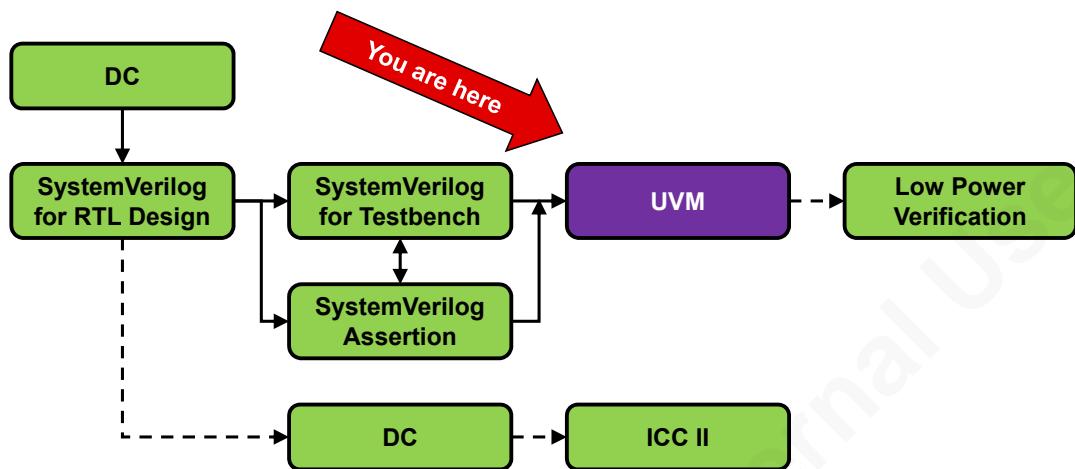
Workshop Prerequisite Knowledge

- You should have experience in the following areas:

- Familiarity with a UNIX text editor
- Programming skills in SystemVerilog
- Debugging experience with SystemVerilog

i-8

Curriculum Flow



i-9

The entire Synopsys Customer Education Services course offering can be found at:
<https://www.synopsys.com/support/training.html>

Agenda

DAY

1

i Introduction

1 OOP Inheritance Review

2 UVM Structural Overview



3 UVM Transaction

4 UVM Sequence



i-10

Agenda

DAY

2

5 UVM Configuration & Factory

6 UVM Component Communication



7 UVM Scoreboard & Coverage

8 UVM Callback



i-11

Agenda

DAY

3

9 UVM Advanced Sequence/Sequencer



10 UVM Phasing and Objections

11 UVM Register Abstraction Layer (RAL)



12 Summary

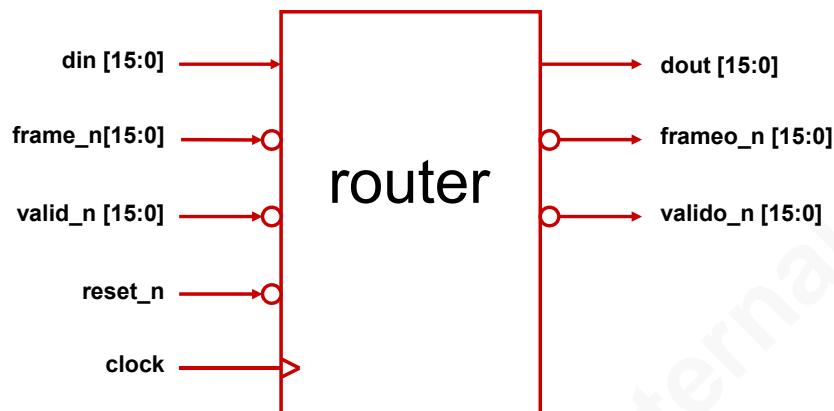
CS Customer Support

i-12

What Is the Device Under Test?

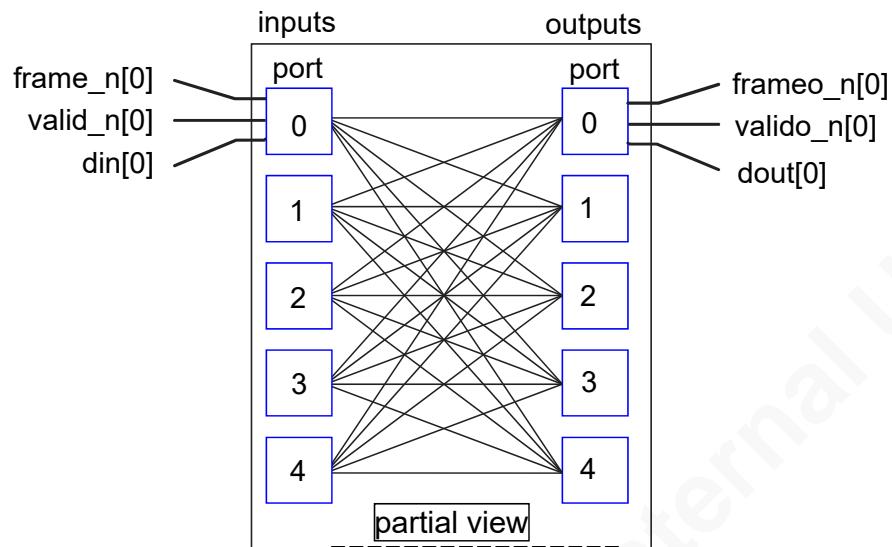
A router:

16 x 16 crosspoint switch



i-13

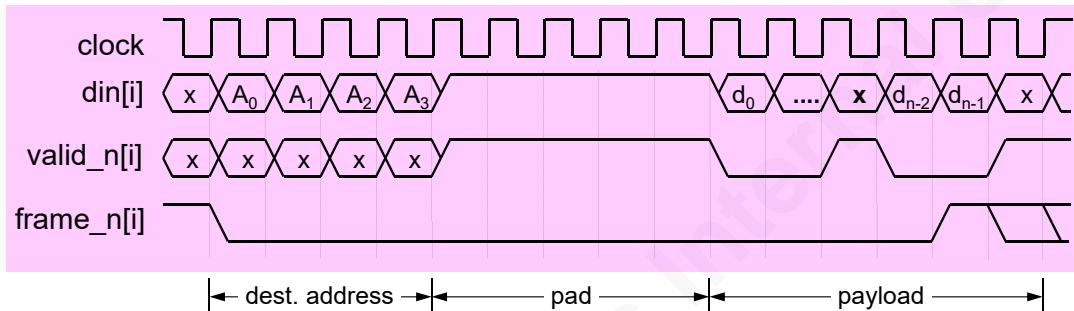
A Functional Perspective



i-14

Input Packet Structure

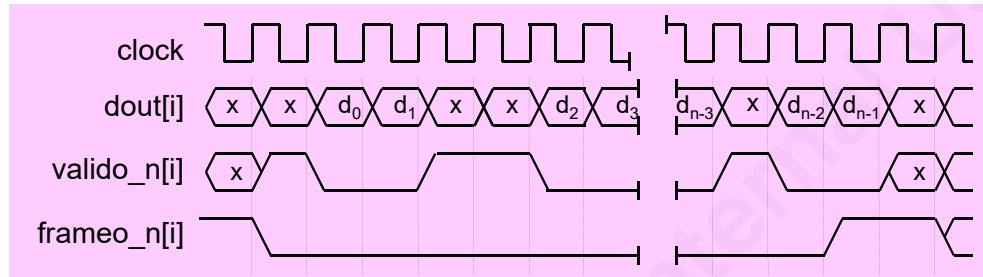
- **frame_n:**
 - Falling edge indicates first bit of packet
 - Rising edge indicates last bit of packet
- **din:**
 - Header (destination address & padding bits) and payload
- **valid_n:**
 - Is low if payload bit is valid, high otherwise



i-15

Input Packet Structure

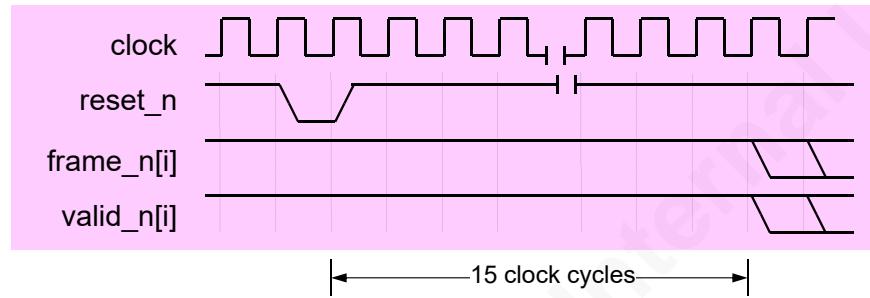
- Output activity is indicated by: `frameo_n`, `valido_n`, and `dout`
- Data is valid only when:
 - `frameo_n` output is low (except for last bit)
 - `valido_n` output is low
- Header field is stripped



i-16

Reset Signal

- While asserting `reset_n`, `frame_n` and `valid_n` must be de-asserted
- `reset_n` is asserted for at least one clock cycle
- After de-asserting `reset_n`, wait for 15 clocks before sending a packet through the router



i-17

Icons Used in this Workshop



Lab Exercise



Caution



Recommendation



Question



For Further Reference



Exercise

i-18

Lab Exercise: A lab is associated with this unit, module, or concept.

Recommendation: Recommendations, tips, performance boost, etc.

For Further Reference: Identifies pointer or URL to other references or resources.

Caution: Warnings of common mistakes, unexpected behavior, etc.

Question: Marks questions asked on the slide.

Exercise: Test for Understanding (TFU), which may require you to work in groups.

Agenda

DAY

1

1 OOP Inheritance Review

2 UVM Structural Overview



3 UVM Transaction



4 UVM Sequence



Synopsys 40-I-055-SSG-007

© 2019 Synopsys, Inc. All Rights Reserved

1-1

Unit Objectives



After completing this unit, you should be able to:

- Use OOP inheritance to create new OOP classes
- Use Inheritance to add new properties and functionalities
- Override methods in existing classes with inherited methods using virtual methods and polymorphism

1-2

Object Oriented Programming (OOP): Class

- Similar to a module, an OOP **class** encapsulates:

- Variables (**properties**) used to model a system
- Subroutines (**methods**) to manipulate the data
- Properties & methods are called **members** of class

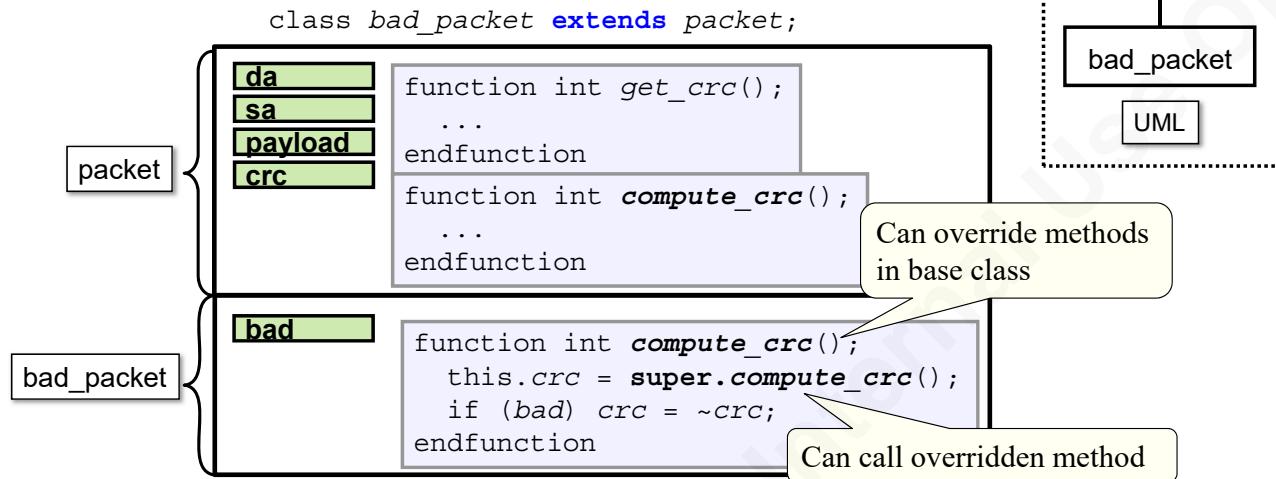
```
class packet;
    bit[3:0]  sa, da;           // packet class properties
    bit[7:0]  payload[$];      // packet class property
    int       crc;              // packet class property
    function int get_crc();    // packet class method
        ...
    endfunction
    function int compute_crc(); // packet class method
        ...
    endfunction
endclass
```

1-3

OOP: Inheritance

■ OOP inheritance

- New classes extends from original (base) class
- Inherits all contents of base class



1-4

Suppose you have defined a **packet** class and want to now make a packet that can be corrupted.

You could create a new class, but then any changes to **packet** would have to be manually added to the new class.

Instead, extend the **packet** class, and add new methods and properties.

A **packet** object has the properties **da**, **sa**, **data**, and **crc**, plus **display** and **compute_crc** methods.

A **bad_packet** object has all these and an additional **bad** property.

So every **bad_packet** object is also a **packet** object.

Terms: **packet** is the **base** class, and **bad_packet** is the **derived** class.

packet is the **base** class of **bad_packet**, and **bad_packet** is **derived** from **packet**.

A class can refer to its base class with the **super** prefix, as shown in **bad_packet**'s **compute_crc()**.

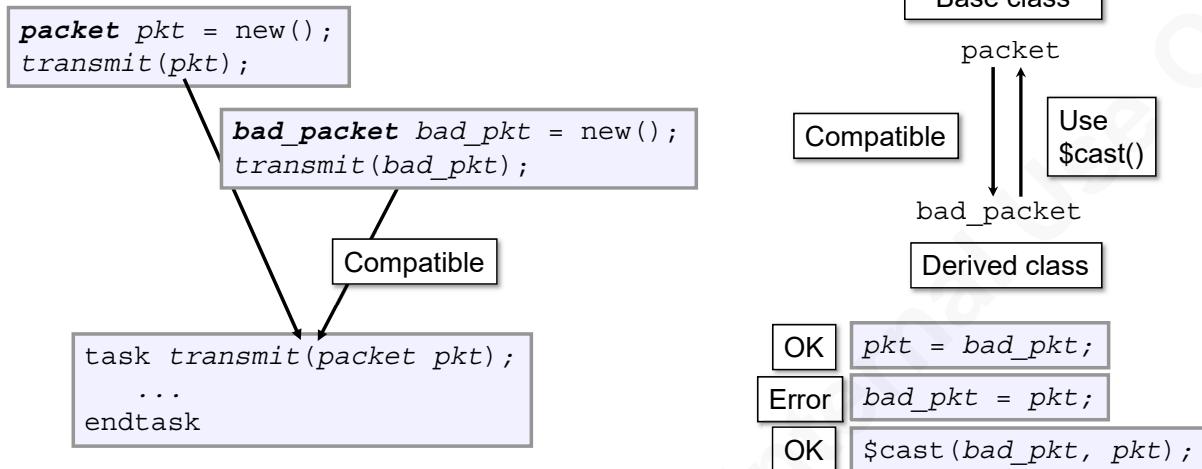
A class can't go up more than one level – **super.super.name** is not legal.

Lastly, you can create a method in the derived class with the same name as one in the base class. You will see shortly how this can be used to inject new behavior. But don't create a property in the derived class with the same name as an existing property. While this is legal, it is a bad programming practice in SystemVerilog and leave you confused as to which definition is being used.

OOP: Inheritance

■ Derived classes compatible with base class

- Can reuse code



1-5

On the left is a diagram showing how a task can pass a *packet* object handle to the *transmit* method. This method can access the object members such as *sa*, and *compute_crc()* with *pkt.sa*, and *pkt.compute_crc()*. One can also pass a *bad_packet* object handle to *transmit()* as every *bad_packet* object is also a *packet*, with *sa*, and *compute_crc()*.

On the lower right, this idea is shown by the handle assignment ***pkt = bad_pkt***; After this assignment, the handle *pkt* can access object members such as *pkt.sa* and *pkt.compute_crc()*.

But, the SystemVerilog compiler does not allow a base handle to be assigned to an extended handle. If it allowed ***bad_pkt = pkt***; then potentially the handle *bad_pkt* could point to a *packet* object. If this happened, ***bad_pkt.bad*** would refer to a variable that does not exist in the object. So, this statement causes a compile-time error.

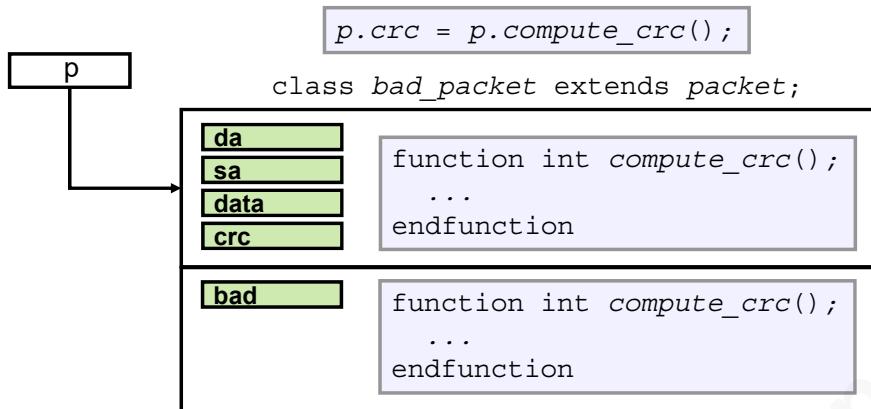
However, if *pkt* did point to a *bad_packet* object, then it should be legal to make the handle *bad_pkt* point to the same object. The only way to know is to check the type of the object pointed to by *pkt*. At **run-time**, the statement ***\$cast(bad_pkt, pkt)*** checks the type of the object that *pkt* points to. If the object is of type *bad_packet* or is extended from *bad_packet*, the handle *bad_pkt* is assigned the value of the *pkt* handle. If *pkt* does not point to a compatible object, VCS will terminate simulation.

When you call *\$cast* as a task, it will cause termination of simulation if the source object is not type compatible with the destination handle. However, if you call *\$cast* as a function, it silently returns a 1 for success, and 0 for failure.

```
if (!$cast(bad_pkt, pkt))
$display("pkt type is not compatible with bad_pkt handle");
```

OOP: Polymorphism

■ Which method gets called?



■ Depends on

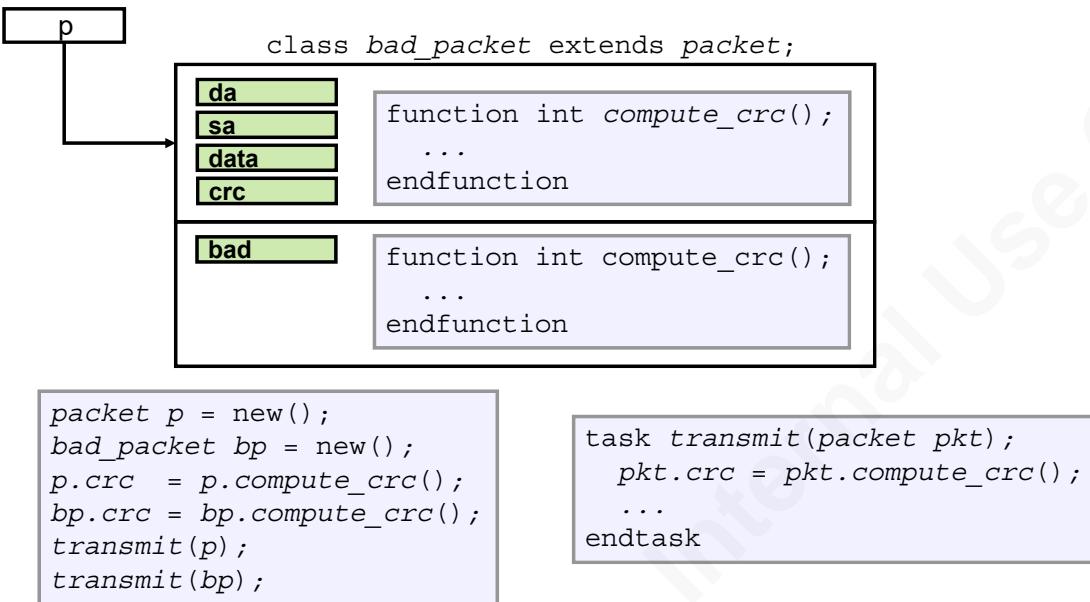
- Type of handle *p* (e.g. “*packet*” or “*bad_packet*” ?)
- Whether *compute_crc()* is *virtual* or not

1-6

There are two *compute_crc* methods. Which one is called when you call *p.compute_crc()*? It depends on the type of the *p* handle, *packet* or *bad_packet*, and if *compute_crc* is virtual or not.

OOP: Polymorphism

- If `compute_crc()` is not virtual



1-7

In the example above,

`p.compute_crc()` executes the `compute_crc()` method in `packet`

`bp.compute_crc()` executes the `compute_crc()` method in `bad_packet`

And,

`transmit(p)` executes the `compute_crc()` method in `packet`

But,

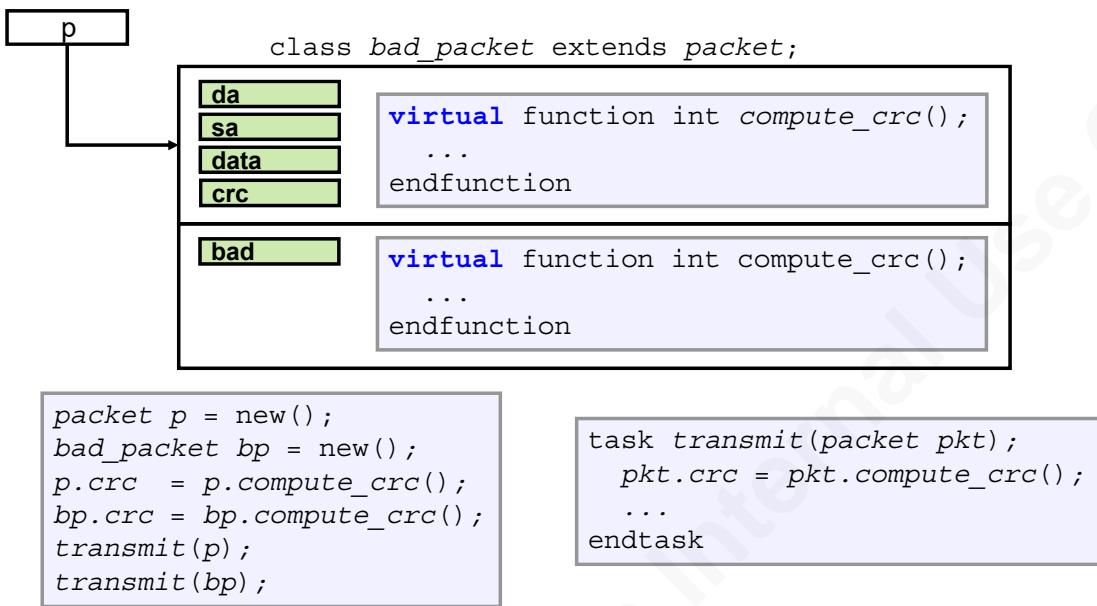
`transmit(bp)` also executes the `compute_crc()` method in `packet` method despite the fact that `bp` is a handle to a `bad_packet` object.

The reason is that the handle used to execute `compute_crc()`, `pkt`, is a `packet` handle. Therefore, its scope is `packet`. If a method within a scope is not declared as `virtual`, then the local scope version of that method will be executed.

If `compute_crc()` is not `virtual`, then there is no polymorphic method execution.

OOP: Polymorphism

- If `compute_crc()` is **virtual**



1-8

When a method within the scope of the object is declared to be **virtual**, then the last definition of the method in the object's memory will be executed.

Bottom line: polymorphism requires methods to be declared as **virtual**.

Caution: once a method is declared to be **virtual**, the signature of that method must remain identical throughout all derived classes.

Example, the following will not compile:

```
class A;
    virtual task t(A a=null);
        $display("A");
    endtask
endclass

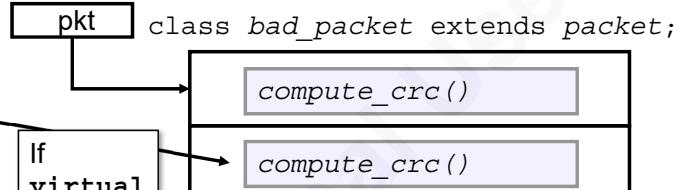
class B extends A;
    virtual task t(B a=null);
        $display("B");
    endtask
endclass
```

OOP: Polymorphism

■ Trying to inject errors

```
class protocol;
  ...
  virtual task transmit(packet pkt);
    pkt.crc = pkt.compute_crc();
  ...
endtask
endclass
```

```
protocol bfm = new(...);
repeat (100) begin
  bad_packet bp = new();
  bfm.transmit(bp);
end
```



Can inject CRC errors
without modifying original code



Guideline: methods should be **virtual**

1-9

User class should declare methods as virtual methods unless one can justify why the method cannot be virtual (very few user can successfully make this justification).

For base class developers (very few fall into this category), however, whether or not the method should be declared as virtual is made with a different consideration. The base class methods intended for the users to override are declared as virtual methods. But, public access methods are typically declared as non-virtual methods to ensure that the user who extends from the base class does not corrupt the intent usage model of these public access methods.

Unit Objectives Review

You should now be able to:

- Use OOP inheritance to create new OOP classes
- Use Inheritance to add new properties and functionalities
- Override methods in existing classes with inherited methods using virtual methods and polymorphism



1-10

Appendix

Parameterized Class
Typedef Class
External Definition
Static Property and Method
Singleton Classes
Singleton Objects
Proxy Classes
Factory Class
Singleton Core Service Class

1-11

Parameterized Classes

■ Written for a generic type

- Parameter type specified at instantiation, just like parameterized modules
- Allows reuse of common code

```
class stack #(type T = int);
local T items[$];
function void push( T a );
...
function T pop();
function int size(); ...
endclass
```

```
program automatic test;
stack #(bit[31:0]) addr_stack;
stack #(packet) pkt_stack;
initial begin
...
repeat(10) begin
packet pkt = new();
if(!pkt.randomize())
$finish;
pkt.addr = addr_stack.pop();
pkt_stack.push(pkt);
end
end
endprogram: test
```

1-12

Parameterized classes allow you to write many types of generic components. e.g. Generator

```
class Generic_Generator #(type T = Packet);
T randomized_obj = new();
...
while(pkt_cnt < run_for_n_pkts) begin
T pkt;
if(!randomized_obj.randomize()) $finish;
$cast(pkt, randomized_obj.clone());
port.put(pkt);
end
...
endclass
```

Parameterized classes can be extended just like other classes.

typedef

■ Can be used to make a forward declaration of a class

- e.g. circular reference

```
typedef class S2;
class S1;
  S2 inside_S1;
  ...
endclass: S1
class S2;
  S1 i_am_inside;
  ...
endclass: S2
```

This is a compile error if typedef is missing

■ Can simplify parameterized class usage

```
typedef stack #(bit[31:0]) stack32;
typedef stack #(packet)   stack_pkt;
program automatic test;
  stack32      addr_stack;
  stack_pkt    data_stack;
```

1-13

The top example shows two classes that refer to each other. Without the typedef, class *S1* can't declare a handle of type *S2*. If you reverse the class compilation order, then class *S2* can't declare a handle of *S1*. Use typedef to tell the compiler that a class will be defined later on.

Another use for typedef is to create new data types. Rather than typing the fully parameterized class name, you can use typedef to define a new data type based on the parameter.

Methods Outside of the Class

- **The body of the class should fit on one “screen”**
 - Show the properties, and method headers
- **Method bodies can go later in the file**
 - Scope resolution operator :: separates class and method name

```
class packet;
    bit[3:0] da, sa; bit[7:0] payload[$]; int crc;
    extern virtual function int get_crc();
    extern virtual function int compute_crc();
endclass

function int packet::get_crc();
    ...
endfunction

function int packet::compute_crc();
    ...
endfunction
```

1-14

The method header inside the class must match the external definition. The virtual label is not required and not allowed for the external source code declaration.

If you declared default values, they must be specified in both the class method declaration and the actual method implementation.

Static Property

- How do I create a variable shared by all objects of a class, but not make a global?
- A static property is associated with the class, not the object
 - Can store meta-data, such as number of instances constructed
 - Single memory allocated all objects of that class
 - ◆ All objects share access to same memory

```
class packet;
    static int count = 0;
    int id;
    function new();
        id = count++;
    endfunction
endclass
```

Using a id field can help keep track of transactions as they flow through test

1-15

Non-static variables declared within a function disappear at the end of the function execution. So when we call the function again, storage for these variables will be re-created and content reinitialized.

Static variables inside a function or class are local in scope in which they are defined, but its lifetime is throughout the program execution. So if we want the value of a variable in function or class to persist throughout the life of a program, we can define these function/class variable as "static."

Initialization for these static variables in function/class is done only once at the beginning of simulation.

Static Method

- A static method allows access via class name

- Can only access static properties in the class
- Method memory is allocated per call
 - ◆ Each static method call creates its own memory

- Cannot be declared as virtual

```
class packet;
    static int count = 0;
    int id;
    static function void print_count();
        $display("Created %0d packets", count);
    endfunction
    ...
endclass
```

```
function void test::end_of_test();
    packet::print_count();
    ...
endfunction
```

1-16

A static method can be called with the class name::method(), no object/handle needed.

Singleton Objects

- A singleton object is a globally accessible static object providing customizable service methods

- One and only one object in existence
 - ◆ Created at compile-time
- Globally accessible at run-time
- Can have static and non-static members
- See slides 24 & 25 for customization

```
class service_class;
    protected static service_class me = get();
    static function service_class get();
        if (me == null) me = new(); return me;
    endfunction
    extern virtual function void error (string msg);
endclass

service_class service_object = service_class::get();
service_object.error("A different error");
```

Object created at
compile-time

Globally accessible
at run-time

Non-static &
virtual

1-17

A singleton object is a static object that is automatically constructed without the user creating any instance of the class outside of the scope of the class. This singleton object's handle can be retrieved with the static get() method.

The execution of the non-static methods will require the user to first retrieve the single object's handle then execute the method through the retrieved handle as shown above.

Simple Singleton Proxy Object Example

- Provides universal service methods like `create()`

```
class proxy_class #(type T=base);
    typedef proxy_class#(T) this_type;           // For coding convenience (not required)
    static this_type me = get();                 // Constructs a static object of this proxy
                                                // type at compile-time

    protected function new(); endfunction
    static function this_type get();
        if (me == null) me = new(); return me;
    endfunction
    static function T create();
        create = new();
    endfunction
endclass

class environment;
    driver drv; monitor mon;
    function new();
       drv = driver::proxy::create();
        mon = monitor::proxy::create();
    endfunction
endclass

class driver extends base;
    typedef proxy_class#(driver) proxy;
endclass

class monitor extends base;
    typedef proxy_class#(monitor) proxy;
endclass
```

1-18

A "proxy" is something/someone who does a service for you. In ancient times, a king would send his proxy to negotiate in a dangerous area. The proxy has the power to speak on behalf of the king.

An OOP proxy class is a service mechanism to create objects.

The simple proxy class shown in the slide is interesting, but by itself is of limited use. You need to have a factory mechanism to make the proxy classes meaningful. To enable this, you will need a virtual proxy base class, a proxy class deriving from the proxy class and a factory class. The next few pages will go through the concept of factory.

Applying Proxy Class in Factory (1/5)

- To implement factory, two proxy class layers are needed:

- Polymorphic virtual proxy base class (this slide)
- A derived proxy class which implements full functionality
 - ◆ Act as molds in factory

- Polymorphic virtual proxy base class

- Only to enable OOP polymorphism for all proxy classes
- No object of this class exists
- Specifies required methods for proxy services

```
virtual class proxy_base;
    virtual function base create_object(string type_name);
        return null;
    endfunction
    pure virtual function string get_typename();
endclass
```

1-19

Applying Proxy Class in Factory (2/5)

Factory class

- Provide creation of object on demand
- Maintain a registry of proxy objects

```
class factory;
    static proxy_base registry[string];
    static factory me = get();
    static function factory get();
        if (me == null) me = new(); return me;
    endfunction

    function void register(proxy_base proxy);
        registry[proxy.get_typename()] = proxy;
    endfunction

    function base create_object_by_type(proxy_base proxy, string name);
        proxy = find_override(proxy);
        return proxy.create_object(name);
    endfunction

    // continued on next slide
```

1-20

Applying Proxy Class in Factory (3/5)

Factory class continued

- Maintains a registry of proxies to be replaced with overridden proxies

- If overridden, creates overridden objects

```
static string override[string];  
static function void override_type(string type_name, override_typename);  
    override[type_name] = override_typename;  
endfunction  
  
function proxy_base find_override(proxy_base proxy);  
    if (override.exists(proxy.get_typename()))  
        return registry[override[proxy.get_typename()]];  
    return proxy;  
endfunction  
endclass
```

Original proxy type

To be replaced by overridden proxy

1-21

Applying Proxy Class in Factory (4/5)

■ Proxy class

```
class proxy_class#(type T=base, string Tname="T") extends proxy_base;
  typedef proxy_class#(T, Tname) this_type;
  static string type_name = Tname;
  static this_type me = get();
  static function this_type get();
    if (me == null) begin me = new(); factory::get().register(me); end
    return me;
  endfunction
  static function T create(string name);
    $cast(create, factory::get().create_object_by_type(me, name));
  endfunction
  virtual function base create_object(string name);
    T object_represented = new(name);
    return object_represented;
  endfunction
  virtual function string get_typename(); return type_name; endfunction
endclass
```

Implements user access methods

Implements abstracted methods

Extends from polymorphic base class

Registers the proxy into factory

Use factory to create object

1-22

Applying Proxy Class in Factory (5/5)

Why go through all these trouble?

- Test can use factory to override any object created in environment without touching environment class

```
class delayed_driver extends driver;
  typedef proxy_class #(delayed_driver, "delayed_driver") proxy;
  // other code not shown
endclass

program automatic test;
  environment env;
  initial begin
    factory::override_type("driver", "delayed_driver");
    env = new();
  end
  class environment;
    driver drv; monitor mon;
    function new();
     drv = driver::proxy::create();
      mon = monitor::proxy::create();
    endfunction
  endclass
endprogram
```

The override will cause a **delayed_driver** to be created instead of **driver**

1-23

Singleton Object Core Service (1/2)

■ Enabling core service singleton

```
class core_service;
protected static core_service me = get();
static function core_service get();
    if (me == null) me = new(); return me;
endfunction
protected error_service_class err;
virtual function error_service_class get_error_service();
    if (err == null) err = new(); return err;
endfunction
virtual function void set_error_service(error_service_class err);
    this.err = err;
endfunction
endclass
```

Use core service to
retrieve service singleton

Core service class is a
service which returns the
desired service singleton

For each service, an instance and
the associated **set()** and **get()**
methods must be implemented

```
class error_service_class;
protected static error_service_class me = get();
static function error_service_class get();
    return core_service::get().get_error_service();
endfunction
extern virtual function void error (string msg);
endclass
```

1-24

Singleton Object Core Service (2/2)

■ Modify service method by registering new service

```
class new_error_service_class extends error_service_class;
    protected static new_error_service_class me = get();
    static function new_error_service_class get();
        if (me == null) me = new(); return me;
    endfunction
    extern virtual function void error (string msg);
endclass
```

Modify service method

```
program automatic test;
    initial begin
        error_service_class::get().error("TEST message");
        core_service::get().set_error_service(new_error_service_class::get());
        error_service_class::get().error("TEST message");
    end
endprogram
```

Will execute original method

Will execute modified method. No need to change reference class name.

1-25

This page was intentionally left blank

Agenda

DAY

1

1 OOP Inheritance Review

2 UVM Structural Overview



3 UVM Transaction



4 UVM Sequence



Synopsys 40-I-055-SSG-007

© 2019 Synopsys, Inc. All Rights Reserved

2-1

Unit Objectives



After completing this unit, you should be able to:

- **Describe the process of reaching verification goals**
- **Describe the UVM testbench architecture**
- **Describe the different components of a UVM testbench**
- **Bring different components together to create a UVM environment**

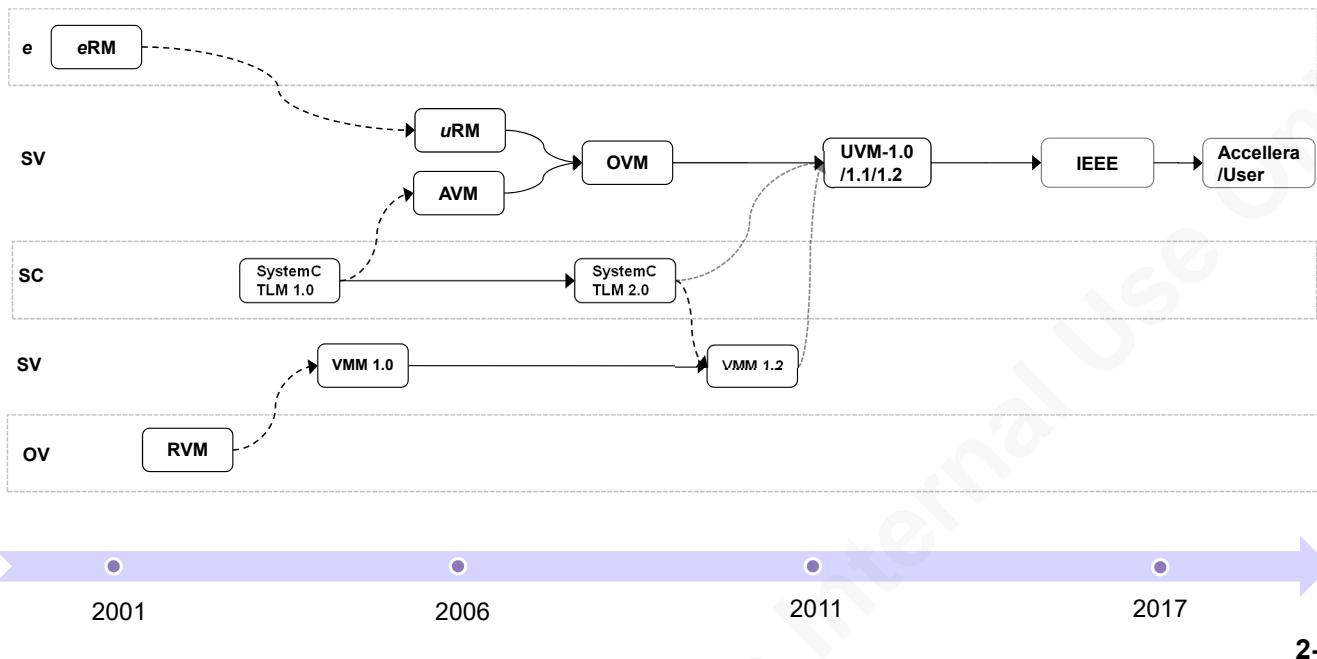
2-2

UVM - Universal Verification Methodology

- An effort (by an Accelerate committee) to define a standard verification methodology & base class library
 - Uses classes and concepts from VMM, OVM
 - UVM-1.2 has been adopted by IEEE as P1800.2 standard (with modifications)
- Related Websites:
 - UVM Public Website – <http://www.accellera.org/community/uvm>
 - ◆ Download Standard Universal Methodology Class Reference
 - ◆ Download UVM User Guide
- Synopsys SNUG & verification video:
 - <http://www.synopsys.com/Community/SNUG/Pages/default.aspx>
 - <https://www.synopsys.com/support/training/ces-training-videos-2016.html>
 - <https://www.youtube.com/user/synopsys/videos>

2-3

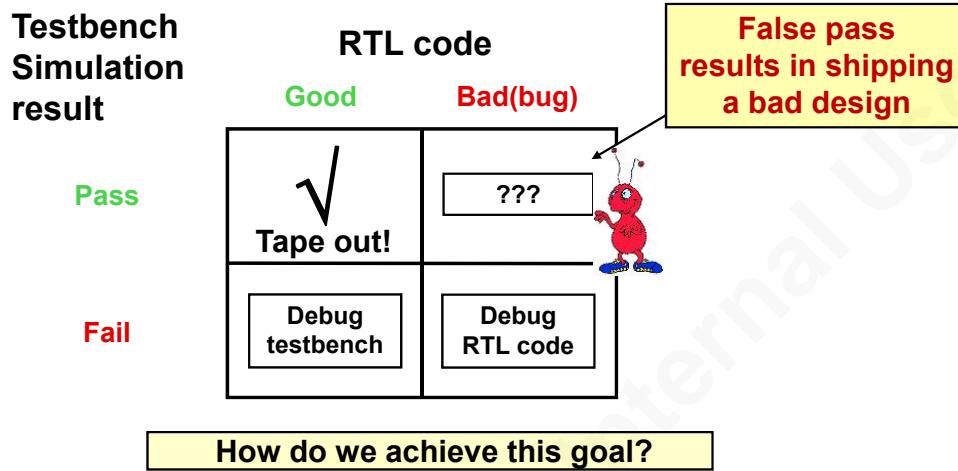
Origin of UVM



Verification Goal

- Ensure full conformance with specification:

- Must avoid false passes



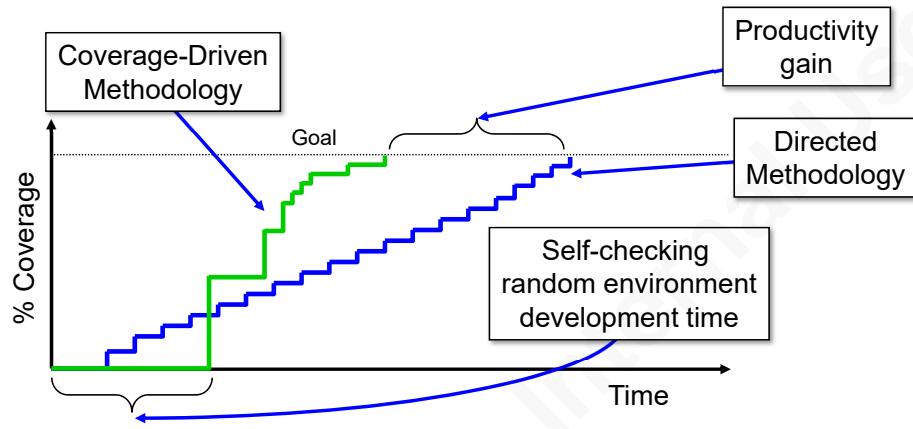
2-5

No one ever wants to ship bad (faulty) RTL. The goal of verification is to find all the bugs in the RTL code.

It is imperative that your verification environment expose as many bugs in the environment as possible.

Coverage-Driven Verification

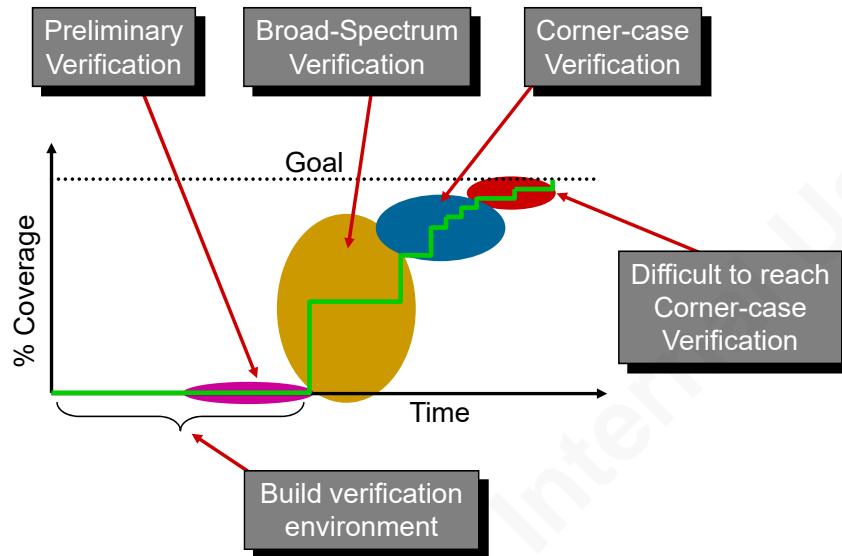
- Focus on uncovered areas
- Trade-off authoring time for run-time
- Progress measured using functional coverage metrics



2-6

Phases of Verification

- Start with fully random environment
- Continue with more and more focused guided tests



2-7

What does it mean to be done with testing?

Typically, the answer lies in the functional coverage spec within a verification plan.

Once the goal has been defined, the first step in constructing the testbench is to build the verification environment.

In order to verify the environment is set up correctly, preliminary verification tests are executed to wring out basic RTL and testbench errors.

When the testbench environment is deemed to be stable, broad-spectrum verification based on random stimulus generation is utilized to quickly detect and correct the majority of the bugs in both RTL code and testbench code.

During this broad-spectrum testing phase, each simulation run is terminated when a pre-determined goal for that run is reached. Post-simulation, an analysis of the functional coverage result is done. Then, the random stimulus constraints are adjusted to focus on cases not reached during simulation.

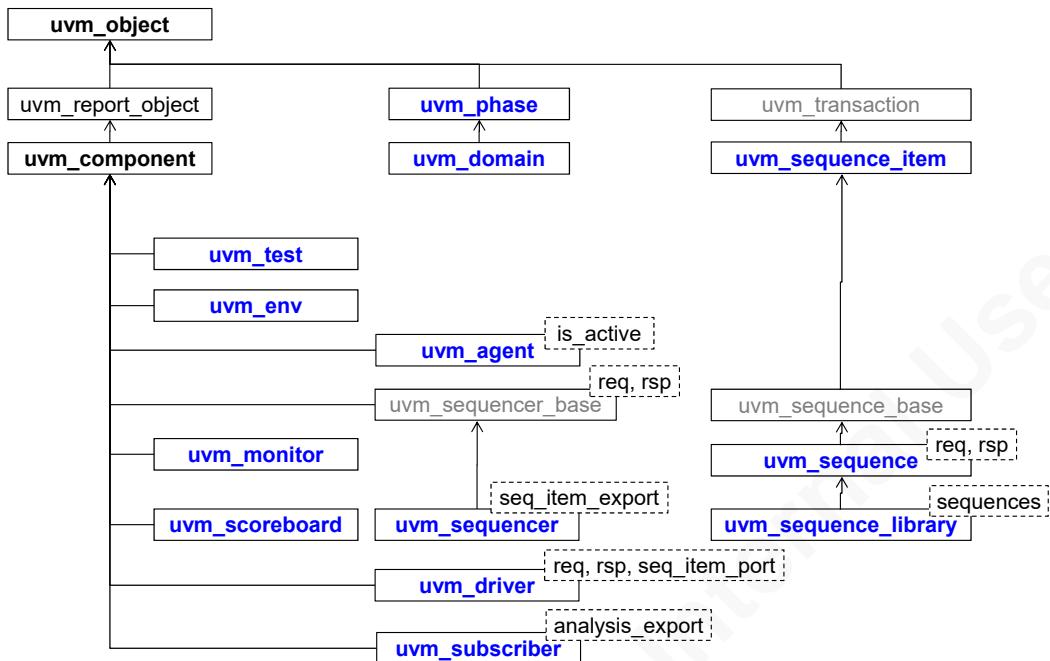
Finally, for the very difficult to reach corner cases, customized directed tests are used to close the coverage gap between test runs and the verification plan requirements.

Run More Tests, Write Less Code

- **Environment and component classes rarely change**
 - Sends good transactions as fast as possible
 - Keeps existing tests from breaking
 - Leave "hooks" so test can inject new behavior
 - ◆ Virtual methods, configuration properties, factories, callbacks
- **Test extends testbench classes**
 - Add constraints to reach corner cases
 - Override existing classes for new functionality
 - Inject errors, delays with callbacks
- **Run each test with hundreds of seeds**

2-8

UVM Class Tree (Partial)



2-9

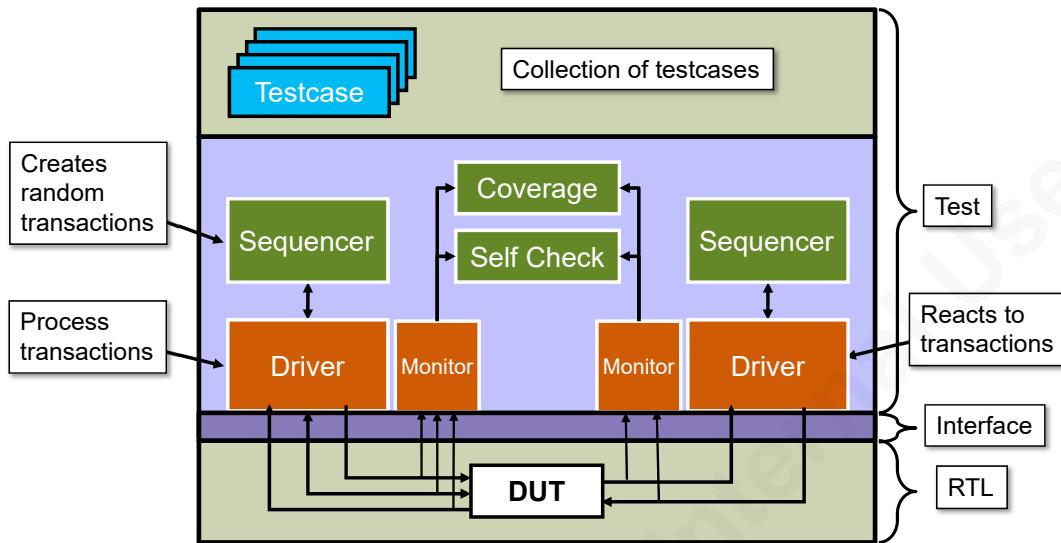
In UVM, structural (`uvm_component`) classes have its own branch of OOP hierarchy.

Structural objects have very specific rules of when it is legal to be constructed, configured, connected and error checked. (to be discussed as part of phasing discussion)

All other classes are freed from these rules.

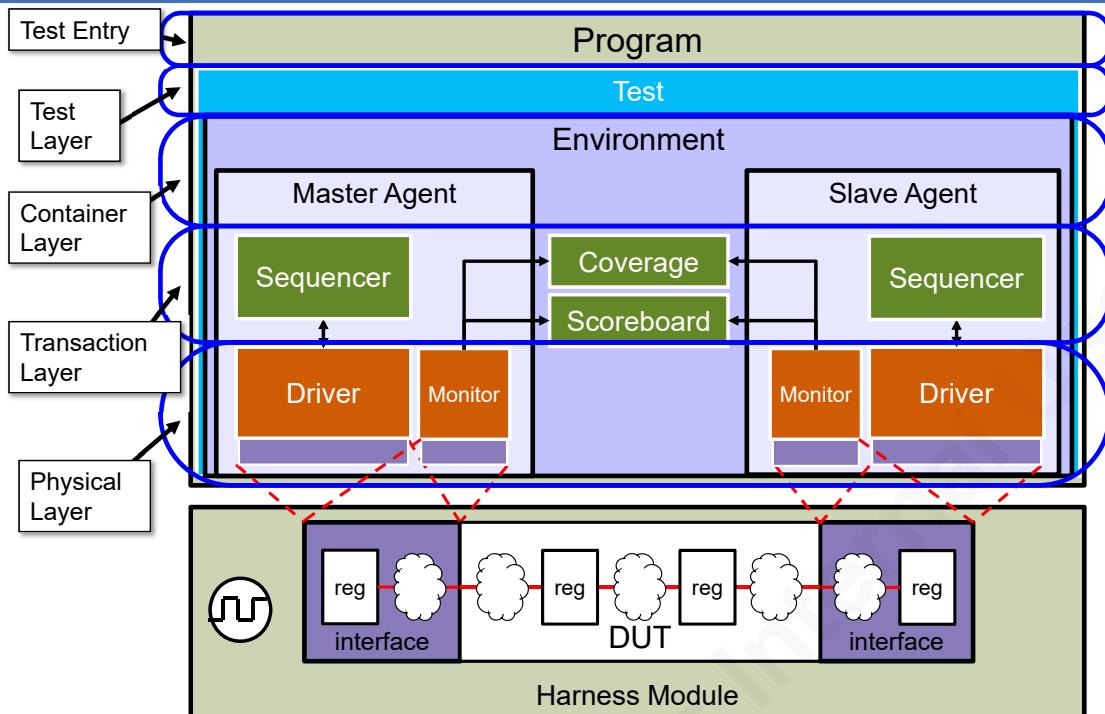
Typical Testbench Architecture

■ SystemVerilog testbench structure



2-10

UVM Testbench Architecture



2-11

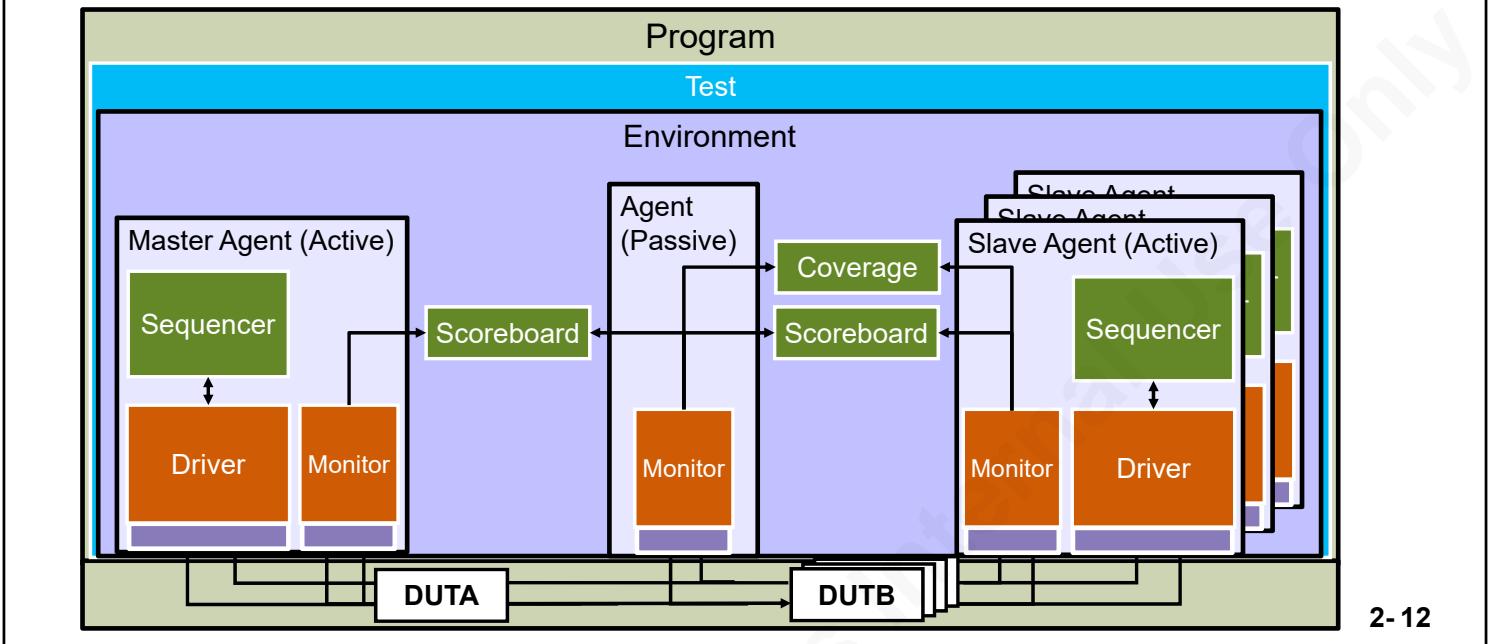
The harness module is the top-level module that instantiates the DUT, the clocks and all the interfaces associated with the DUT. To make development of device drivers in the test environment easier to manage, one should implement clocking blocks in the interface to emulate the behavior of the launch flop at the input of the DUT and the capture flop at the output of the DUT.

In the test environment, the protocol-specific blocks (sequencer, driver, monitor) are grouped together to create an agent that can be instantiated as a unit. This reduces the work to connect to an interface with the specific protocol.

The default environment and its components should be written once and, except for correcting bugs or embedding missing features, rarely changed for the rest of the project. If the default environment is constantly changing, many existing tests will break, resulting in bad consequences! The default environment should send good transactions, as fast as possible. Leave "hooks" in the classes so that this behavior can be changed by the testcase, without modifying the original component code.

UVM Structure is Scalable

- Agents are configurable for reuse across test/projects



Each agent should be externally configurable to either operate in the active or the passive mode. In the active mode, all three members (sequencer, driver and monitor) of the agents will exist. In the passive mode, only the monitor should be created.

With this configurability, the agent becomes highly reusable.

Structural Class Support in UVM

■ Structural & Behavioral

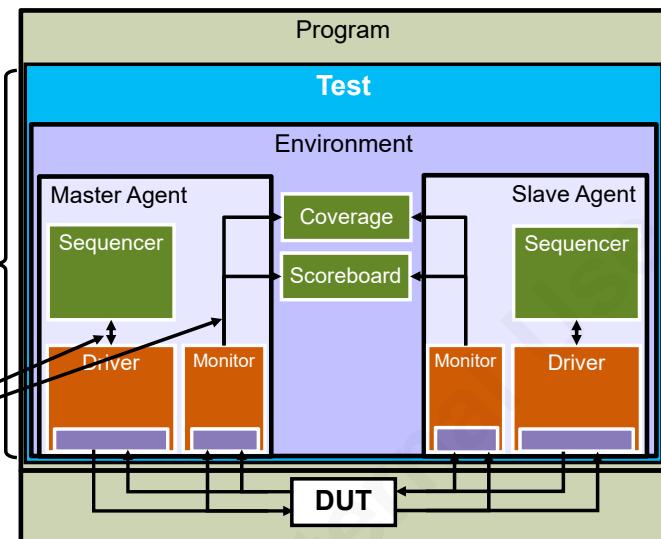
- uvm_component
 - ◆ uvm_test
 - ◆ uvm_env
 - ◆ uvm_agent
 - ◆ uvm_sequencer
 - ◆ uvm_driver
 - ◆ uvm_monitor
 - ◆ uvm_scoreboard

■ Communication

- uvm_*_port
- uvm_*_socket

■ Transaction

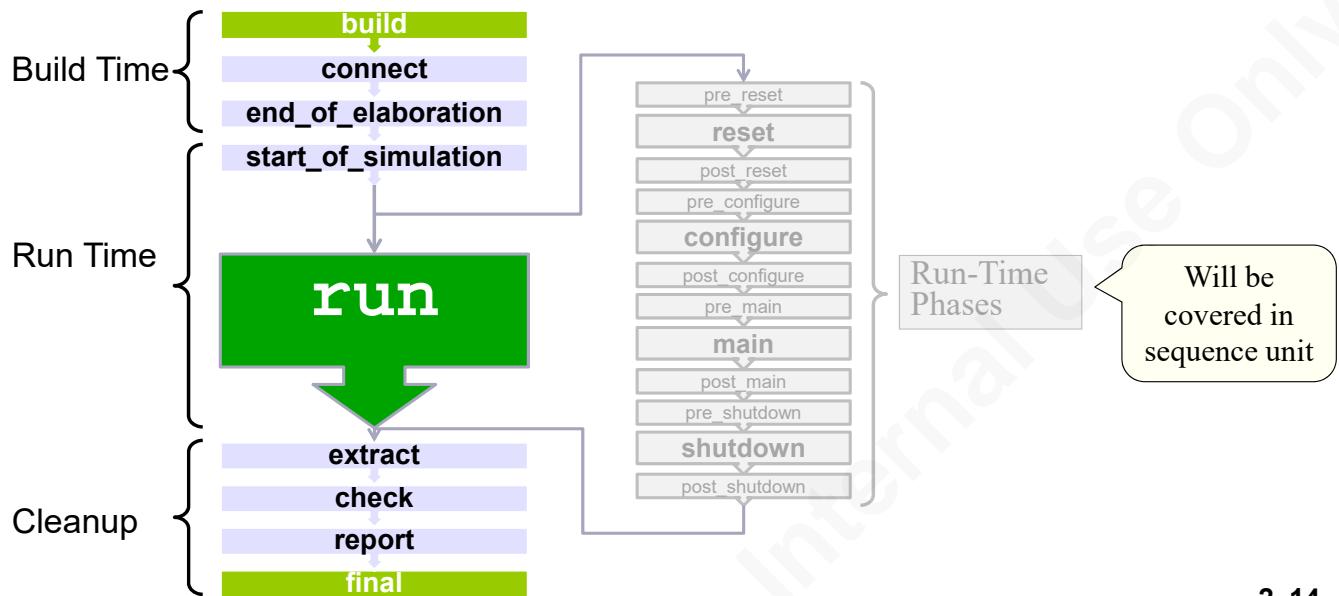
- uvm_sequence_item
- uvm_sequence



2-13

Structural Functional Support - Phasing

- run phase executes concurrently with the scheduled Run-Time phases



2-14

The build phase is called top down so that the higher level components (environments, agents) can decide whether or not to build child components, based on the user specified configuration.

In the connect phase, the parent components connect their children's TLM ports

In the end_of_elaboration phase, optimize and check for correctness of configurations and connections

In the start_of_simulation phase, print the testbench configuration

In the Run Time task phases, the functional verification code executes

In the extract phase, data is extracted from the environment

In the check phase, check the extracted data

In the report phase, report the simulation results

In the final phase, simulation is about to end, print final messages

The run phase should be used to emulate the behavior of the external RTL **always** block in driver and monitor classes and **initial** block in test class.

The Run-Time phases (pre_reset through post_shutdown) should only exist in test classes to manage order of execution of sequences.

UVM Hello World Example

- UVM tests are derived from `uvm_test` class
- Execution of UVM test is done via global task `run_test()`

The diagram shows the Verilog code for a UVM test class, `hello_world`, which extends the `uvm_test` base class. Annotations explain various parts of the code:

- Import UVM package**: Points to the `import uvm_pkg::*;

 class hello_world extends uvm_test;` lines.
- DUT functional verification code resides in one of the task phases**: Points to the `virtual task run_phase(uvm_phase phase);` block.
- Test base class**: Points to the `uvm_test` base class reference.
- Create and register test name**: Points to the `^uvm_component_utils(hello_world)` macro call.
- Execute UVM test**: Points to the `run_test();` call in the initial block.
- Message**: Points to the ``uvm_info("TEST", "Hello World!", UVM_MEDIUM);` message print statement.
- Use phase objection mechanism to stay in phase and execute content of method**: Points to the `phase.raise_objection(this);` and `phase.drop_objection(this);` calls.

```
program automatic test;
  import uvm_pkg::*;
  class hello_world extends uvm_test;
    ^uvm_component_utils(hello_world)
    function new(string name, uvm_component parent);
      super.new(name, parent);
    endfunction
    virtual task run_phase(uvm_phase phase);
      phase.raise_objection(this);
      `uvm_info("TEST", "Hello World!", UVM_MEDIUM);
      phase.drop_objection(this);
    endtask
  endclass
  initial
    run_test();
  endprogram
```

2-15

Your tests must be extended from the `uvm_test` base class.

The macro ``uvm_component_utils()` registers the class in the UVM factory registry.

The `run_phase()` executes time consuming verification code.

If phase objection is not raised in a task phase, that phase will terminate in 0 simulation time. To prevent task phase method from being terminated too early, phase objection must be raised. Message can be printed with the ``uvm_info()` macro with an ID, the message, and a verbosity level. The Verilog initial block is the test entry.

The `run_test()` global method called in the initial block starts the UVM test execution.

Class `hello_world` does not need to be instantiated and constructed!

Compile and Simulate

- Compile with `-ntb_opts uvm-1.2` switch
- Specify test to run with `+UVM_TESTNAME` switch

```
program automatic test;
    import uvm_pkg::*;
    class hello_world extends uvm_test;
        `uvm_component_utils(hello_world)
        function new(string name, uvm_component parent);
            super.new(name, parent);
        endfunction
        virtual task run_phase(uvm_phase phase);
            phase.raise_objection(this);
            `uvm_info("TEST", "Hello World!", UVM_MEDIUM);
            phase.drop_objection(this);
        endtask
    endclass
    initial run_test();
endprogram
```

test.sv

Test name

Compile with vcs: (using UVM in VCS installation)

vcs -sverilog `-ntb_opts uvm-1.2` test.sv

Simulate with:

simv `+UVM_TESTNAME=hello_world`

`UVM_INFO @ 0: reporter [RNTST] Running test hello_world ...`
`UVM_INFO ./test.sv(10) @ 0: uvm_test_top [TEST] Hello World!`

2-16

From VCS 2015.09 onwards, VCS ships with both uvm-1.1 and uvm-1.2 source code.

There are multiple switches possible for `-ntb_opt`: uvm, uvm-1.1 and uvm-1.2.

For each release of VCS, there is a default version of the official UVM source code supported. If you want to use the default version, then use `-ntb_opt uvm`. However, you need to be careful with this switch. VCS 2018.09's default UVM version is uvm-1.1. Other VCS versions may default to other UVM version.

For user customized UVM source code, if one wants to enable the recording one would need to compile as follows (for uvm-1.2):

```
% setenv UVM_HOME <path to UVM>
% vcs -sverilog test.sv \
+incdir+${UVM_HOME}/src \
${UVM_HOME}/src/dpi/uvm_dpi.cc -CFLAGS -DVCS \
${UVM_HOME}/src/uvm_pkg.sv \
+${VCS_HOME}/etc/uvm-1.2/vcs ${VCS_HOME}/etc/uvm-1.2/vcs/uvm_custom_install_vcs_recorder.sv
```

Inner Workings of UVM Simulation

- Macro registers the class in factory (`uvm_factory::get()`)

```
class hello_world extends uvm_test;  
  `uvm_component_utils(hello_world)
```

Registry table

"hello_world"

- UVM package contains a `uvm_root` singleton (`uvm_root::get()`)

```
import uvm_pkg::*;


```

simv

uvm_test_top

- `run_test()` causes `uvm_root` singleton to retrieve the test name from `+UVM_TESTNAME` and create a test component called `uvm_test_top`

```
initial  
  run_test();
```

simv +UVM_TESTNAME=hello_world

build_phase
connect_phase
end_of_elaboration_phase
start_of_simulation_phase
run_phase
extract_phase
check_phase
report_phase
final_phase

- Then, the `uvm_root` singleton executes the phase methods of components

2-17

The `uvm_pkg` package contains a global `uvm_root` singleton object that can be accessed via `uvm_root::get()`. It controls the phasing of all components including the test. It also provides the mechanism for locating components, printing component topology, and other global component structure-related functions.

When the global method `run_test()` is called, it executes the `run_test()` method of the `uvm_root` singleton object. The `uvm_root` singleton object then retrieves the test name from the command line switch (`+UVM_TESTNAME`) and uses the factory to construct a component of that class. This component instance is called "`uvm_test_top`". Afterwards, the `uvm_root` singleton object starts the phase method execution for all of the UVM components starting with the test object.

User Report Messages

■ Print messages with UVM macros

```
`uvm_fatal("CFGERR", "Fatal message")
`uvm_error("RNDERR", "Error message")
`uvm_warning("WARN", "Warning message")
`uvm_info("REGRESS", "Regression message", UVM_LOW)
`uvm_info("NORMAL", "Normal message", UVM_MEDIUM)
`uvm_info("TRACE", "Tracing execution", UVM_HIGH)
`uvm_info("FULL", "Debugging operation", UVM_FULL)
`uvm_info("DEBUG", "Verbose message", UVM_DEBUG)
```

Failure messages are set to **UVM_NONE**

Info messages need to specify verbosity

More verbose

IEEE does not define
UVM_DEBUG verbosity

■ Verbosity filter defaults to **UVM_MEDIUM**

- User can modify run-time filter via **+UVM_VERBOSITY** switch

```
simv +UVM_VERBOSITY=UVM_FULL +UVM_TESTNAME=hello_world
```

2-18

In the current Accellera implementation of UVM verbosity, there is a **UVM_DEBUG** verbosity that the user have access to. This level of verbosity was meant to support debugging the Accellera UVM source code. User should not be using this verbosity.

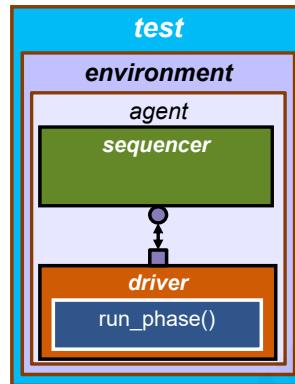
The IEEE committee has decided that all debugging mechanisms are up to the implementer to implement. So, all references to debugging mechanism built into the Accellera UVM source code have been removed from the IEEE UVM spec.

Will Accellera committee continue to embed this verbosity into the Accellera UVM source code? That's a hard question to answer. One can never say with 100% certainty whether that will be the case or not. Given these facts, we recommend that the user avoid using the **UVM_DEBUG** verbosity in their own code.

UVM Simple Structure Example

■ Structural classes

- Test class
 - ◆ uvm_test
- Environment class
 - ◆ uvm_env
- Agent class
 - ◆ uvm_agent
- Sequence execution class
 - ◆ uvm_sequencer
- Driver class
 - ◆ uvm_driver



2-19

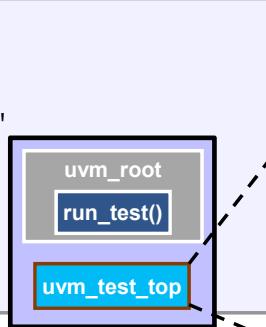
Starting UVM Execution

■ Can be in SystemVerilog program or module

- Includes test class files
- Start UVM execution in **initial** block
 - ◆ `uvm_root` singleton will construct and execute the test specified via `+UVM_TESTNAME`

```
program automatic test;
    import uvm_pkg::*;
    // include the test files
    `include "test_collection.sv"

    initial begin
        run_test();
    end
endprogram
```



```
simv +UVM_TESTNAME=test_base
```

2-20

Although UVM testbenches can be executed in the initial block of a program or module. Synopsys recommends using a program block to execute the testbench code to reduce potential race conditions between the testbench and the design under test (RTL code) in modules.

Structural Classes - Test

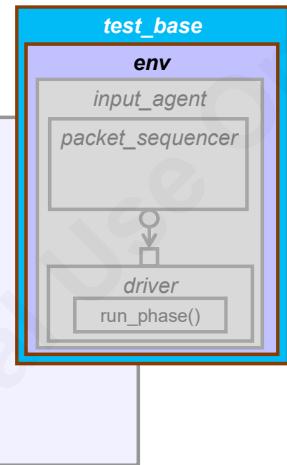
■ UVM test class is the top component of test structure

- Extends from the `uvm_test` base class
- Creates environment object

```
class test_base extends uvm_test;
    router_env env;
    `uvm_component_utils(test_base)
    // constructor not shown
    virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        env = router_env::type_id::create("env", this);
    endfunction
endclass
```

Construct environment object with
UVM factory create mechanism

- Then, develop targeted tests extending from `test_base`



2-21

All components in UVM testbench must be constructed in `build_phase`. If one attempts to construct components in any other phase, the simulation will terminate with a fatal error.

Within the `build_phase` method, whenever a child component is to be constructed, use the component class's `type_id::create(...)` method. Using the `create` method to construct child components will allow tests to replace these components with a customized component targeting a particular test goal.

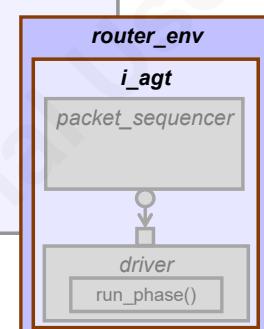
Structural Classes - Environment

■ Encapsulates DUT specific Verification Components

- Encapsulate agents, scoreboard and coverage
 - ◆ Scoreboard and coverage will be addressed in later units

```
class router_env extends uvm_env;
    input_agent i_agt;
    // utils macro and constructor not shown
    virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        i_agt = input_agent::type_id::create("i_agt", this);
    endfunction
endclass
```

In build phase,
create agent object



2-22

Structural Classes - Agent

■ Encapsulate sequencer, driver and monitor in agent

- In the example code, monitor is left off for simplicity
(more on monitor and agent in later unit)

```
class input_agent extends uvm_agent;           Extend from uvm_agent

    typedef uvm_sequencer #(packet) packet_sequencer;
    packet_sequencer sqr; driver drv;

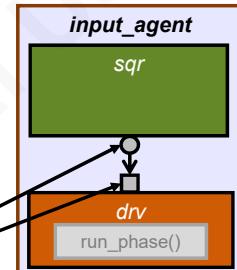
    // utils macro and constructor not shown

    virtual function void build_phase(uvm_phase phase);
        sqr = packet_sequencer::type_id::create("sqr", this);
        drv = driver::type_id::create("drv", this);
    endfunction

    In build phase, construct components

    virtual function void connect_phase(uvm_phase phase);
        drv.seq_item_port.connect(sqr.seq_item_export);
    endfunction

endclass                                         In connect phase, connect built-in TLM ports
```



2-23

The base class uvm_sequencer has the full functionality of the sequencer already coded . Users can use the class as is. However, to make usage a little simpler, one should use the typedef mechanism to eliminate the need to keep typing #() with an argument.

If one does not make use of the typedef, one would need to do the following:

```
class input_agent extends uvm_agent;
    uvm_sequencer#(packet) sqr;
    driver drv;
    virtual function void build_phase(uvm_phase phase);
        sqr = uvm_sequencer#(packet)::type_id::create("sqr", this);
        drv = driver::type_id::create("drv", this);
    endfunction
    ...
endclass
```

The potential of forgetting to type #(packet) thus leading to a fatal problem is too great. Use typedef to eliminate this risk.

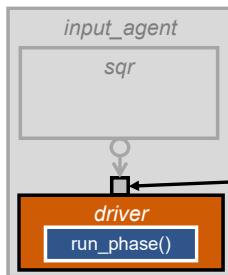
```
sqr = uvm_sequencer::type_id::create("sqr", this); // leads to fatal error
```

Structural Classes - Driver

■ Driver class extends from `uvm_driver` class

- `uvm_driver` class has a built-in TLM port

Must be typed to transaction class



Driver implements `run_phase` as an infinite loop to process all sequence items received from sequencer. `item_done()` in the TLM port must be called before retrieving next item.

2-24

A UVM driver typically receives transactions from the sequencer via the built-in TLM port (`seq_item_port`) and processes them. This one just prints them out.

A driver is a testbench structural component class. The composition path to it (`uvm_test_top.env.i_agt.drv`) must be established for the UVM configuration and override features to work. This is why it has an parent argument in the constructor.

The `run_phase` in the driver emulates the external RTL always block. Within the `run_phase`, the driver does not raise or drop objection. The objection mechanism will be controlled within the sequences or the test.

The `sprint()` function returns a formatted string with the contents of the object. The mechanism of implementing this method is discussed in Unit 3 (UVM Transaction).

Structural Classes - Debug

■ Topology of the test can be printed with

- `uvm_root::get().print_topology()`

```
UVM_INFO @ 0.0ns: reporter [UVMTOP] UVM testbench topology:  
-----  
Name          Type      Size  Value  
-----  
uvm_test_top    test_base -      @510  
  env          router_env -      @517  
    i_agt       input_agent -      @529  
    drv         driver     -      @666  
      rsp_port  uvm_analysis_port -      @681  
        recording_detail uvm_verbosity 32  UVM_FULL  
        sqr_pull_port   uvm_seq_item_pull_port -      @673  
          recording_detail uvm_verbosity 32  UVM_FULL  
          port_id       integral    32  -1  
          recording_detail uvm_verbosity 32  UVM_FULL  
          sqr          uvm_sequencer -      @557  
...  
function void test_base::start_of_simulation_phase(uvm_phase phase);  
  uvm_root::get().print_topology(); // defaults to table printer  
endfunction
```

2-25

The default printer policy is `uvm_default_table_printer`. The print out is nicely formatted in columns. Unfortunately, for topology printing, when the text characters exceeds the predefine width of the columns, truncation of the text happens.

See next slide for potential solution.

Print Format Can Be Specified

- Topology can also be printed in tree format

```
UVM_INFO @ 0.0ns: reporter [UVMTOP] UVM testbench topology:  
uvm_test_top: (test_base@510) {  
    env: (router_env@517) {  
        i_agt: (input_agent@529) {  
            drv: (driver@666) {  
                rsp_port: (uvm_analysis_port@681) {  
                    recording_detail: UVM_FULL  
                }  
                sqr_pull_port: (uvm_seq_item_pull_port@673) {  
                    recording_detail: UVM_FULL  
                }  
                port_id: -1  
                recording_detail: UVM_FULL  
            }  
            sqr: (uvm_sequencer@557) {  
                rsp_export: (uvm_analysis_export@564) {  
                    recording_detail: UVM_FULL  
                }  
            }  
        }  
    }  
    ...  
    function void test_base::start_of_simulation_phase(uvm_phase phase);  
        uvm_root::get().print_topology(uvm_default_tree_printer);  
    endfunction
```

2-26

The tree printer policy is more verbose than the table printer policy, but does not truncate text.

There are three default printer policies that the **uvm_pkg** provides:

uvm_default_table_printer
uvm_default_tree_printer
uvm_default_line_printer

General Debugging with Report Messages

- Create messages with macros:

```
`uvm_fatal(string ID, string MSG)
`uvm_error(string ID, string MSG)
`uvm_warning(string ID, string MSG)
`uvm_info(string ID, string MSG, verbosity)
```

- Example:

```
virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    if (!cfg.randomize()) begin
        `uvm_fatal("CFG_ERROR", "Failed Configuration randomization");
    end
endfunction
```



```
UVM_FATAL test.sv(14) @0.0ns: uvm_test_top[CFG_ERROR] Failed ...
```

Severity

Time

ID

MSG

File & line no.

Object name

2-27

One can turn the file and line numbers off with the compile time option:

```
+define+UVM_REPORT_DISABLE_FILE
```

(Caution: this is an Accellera implementation feature not specified in IEEE P1800.2)

Default Simulation Handling

Severity	Default Action
UVM_FATAL	UVM_DISPLAY UVM_EXIT
UVM_ERROR	UVM_DISPLAY UVM_COUNT
UVM_WARNING	UVM_DISPLAY
UVM_INFO	UVM_DISPLAY
Action	Description
UVM_EXIT	Exit from simulation immediately
UVM_COUNT	Increment global error count. Set count for exiting simulation with +UVM_MAX_QUIT_COUNT= run-time switch
UVM_STOP	Calls Verilog \$stop
UVM_DISPLAY	Display message on console
UVM_LOG	Captures message in a named file
UVM_RM_RECORD	Sends report to the recorder
UVM_NO_ACTION	Do nothing

2-28

Command Line Control of Report Messages

■ Control verbosity of components at specific phases or times

- id argument can be `_ALL_` for all IDs or a specific id
 - ◆ Wildcard for id argument not supported

`+uvm_set_verbosity=<comp>,<id>,<verbosity>,<phase>`

`+uvm_set_verbosity=<comp>,<id>,<verbosity>,time,<time>`

```
+uvm_set_verbosity=uvm_test_top.env.agt.*,_ALL_,UVM_FULL,build  
+uvm_set_verbosity=uvm_test_top.env.agt.*,_ALL_,UVM_FULL,time,800
```

■ Control report message action (like `set_report_*_action`)

`+uvm_set_action=<comp>,<id>,<severity>,<action>`

```
+uvm_set_action=uvm_test_top.env.*,_ALL_,UVM_ERROR,UVM_NO_ACTION
```

■ Control severity (like `set_report_*_severity_override`)

`+uvm_set_severity=<comp>,<id>,<current severity>,<new severity>`

```
+uvm_set_severity=uvm_test_top.*,BAD_CRC,UVM_FATAL,UVM_ERROR
```

2-29

User Filterable Code Block

- Control filtering of block of code

- Based on `uvm_report` mechanism

Verbosity

Severity

ID

```
if (uvm_report_enabled(UVM_HIGH, UVM_INFO, "FACTORY")) begin  
    uvm_factory::get().print();  
end
```

- Does not get tracked by system

ID will not be reported

```
** Report counts by id  
[Comparator Match] 154  
[Comparator Mismatch] 6  
[DRV_RUN] 160
```

2-30

Test For Understanding

- How long does the following run for in simulation?

```
program automatic test;
    import uvm_pkg::*;
    class hello_world extends uvm_test;
        `uvm_component_utils(hello_world)
        function new(string name, uvm_component parent);
            super.new(name, parent);
        endfunction
        virtual task run_phase(uvm_phase phase);
            #50ns;
            `uvm_info("TEST", "Hello World!", UVM_MEDIUM);
        endtask
    endclass
    initial
        run_test();
    endprogram
```

2-31

The above code will terminate in 0 simulation time because no phase objection was raised.

The correct **run_phase()** method implementation should be:

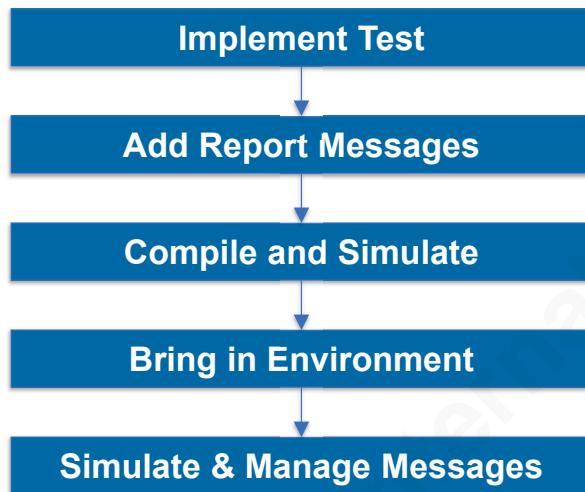
```
virtual task run_phase(uvm_phase phase);
    phase.raise_objection(this);
    #50ns;
    `uvm_info("TEST", "Hello World!", UVM_MEDIUM);
    phase.drop_objection(this);
endtask
```

Lab 1: Managing UVM Report Messages



45 minutes

Implement test, environment and report messages



2-32

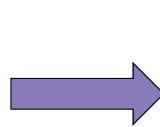
Key Structural Concept: Parent-Child

■ Parent-child relationships

- Set up at component creation
- Establishes component phasing execution order
- Establishes component hierarchical path for component configuration and factory override
- Composition hierarchy – Not OOP hierarchy

■ Phase execution order

- Each component follows the same order of phase execution
(* run phases are executed concurrently)



```
build
connect
end_of_elaboration
start_of_simulation
run*
extract
check
report
final
```

■ Search path allow tests to:

- Configure specified components
- Override specified components in environment

2-33

UVM puts components in a logical hierarchical composition tree, based on component name, and the parent handle that is established at the component's construction.

The component parent-child relationship is a composition relationship. It is **not** a OOP inheritance relationship.

In this composition structural hierarchy, the test ("**uvm_test_top**") sits at the very top of the hierarchy. The environment sits below the test. Components such as agents, drivers and monitors below the environment.

This composition structural hierarchy is important as it enables you to configure components, or replace them with modified behaviors, using the hierarchical names.

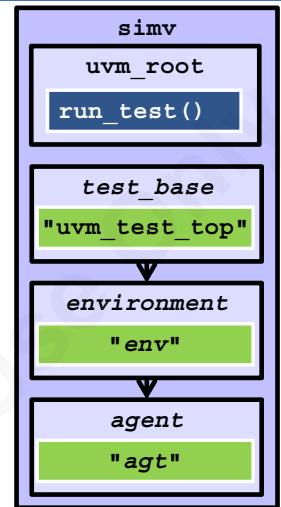
Key Component Concepts: Logical Hierarchy

```
class test_base extends uvm_test;
  environment env;
  `uvm_component_utils(test_base)
  function new(string name, uvm_component parent);
    virtual function void build_phase(uvm_phase phase);
      super.build_phase(phase);
      env = environment::type_id::create("env", this);
    endfunction
  endclass
  class environment extends uvm_env;
    agent agt;
    `uvm_component_utils(environment)
    function new(string name, uvm_component parent);
      virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        agt = agent::type_id::create("agt", this);
      endfunction
    endclass
    class agent extends uvm_agent;
      `uvm_component_utils(agent)
      function new(string name, uvm_component parent);
        super.new(name, parent);
      endfunction
      virtual task class_task(...);
    endclass
```

Establish parent-child relationship at creation

Set parent

Parent handle



2-34

This slide shows a simple hierarchy. At the bottom is the agent class. Its constructor has two arguments, the logical name, and its parent.

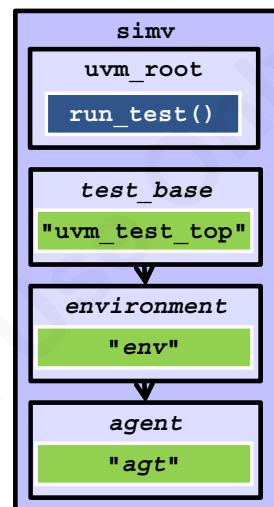
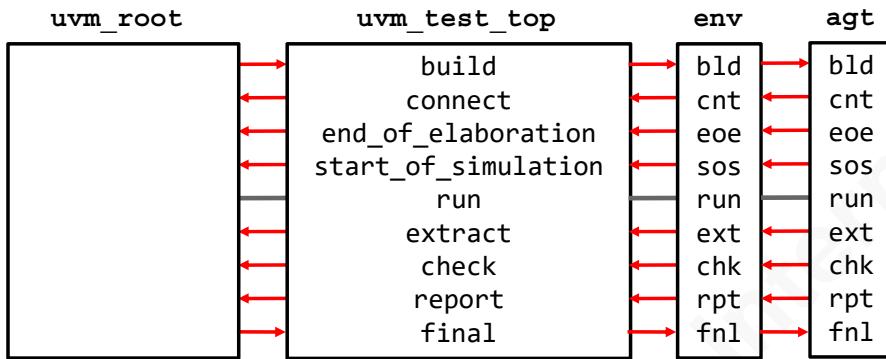
The parent component is the environment which contains an instance of the agent. With the execution of `agent::type_id::create()`, the UVM factory creates an `agent` object, under the environment.

At the top is the `test_base` class, which creates an instance of the environment object as its child component.

Key Component Concepts: Phase

■ Parent-child relationship dictates phase execution order

- Functions are executed bottom-up
 - ◆ Except for build and final phases which are executed top-down
- Tasks are forked into concurrent executing threads



2-35

The **uvm_root** singleton object, manages all the phase executions of the test by executing all the component phase methods.

All phase methods are zero-time functions, except for **run_phase()** and the concurrent Run-Time task phases.

All the function phase methods are executed bottom up except **build_phase** and **final_phase**.

The **build_phase** methods have to be executed top down so that configuration settings set at the top (test) can propagate down to the components. The **final_phase** methods are executed top down to allow the test to control what the lower layer **final_phase** methods will do.

Key Component Concepts: Override

```
class test_new extends test_base; ...
  `uvm_component_utils(test_new)
  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    set_inst_override_by_type("env.agt", agent::get_type(),
      new_agt::get_type());
  endfunction
endclass
```

Use component hierarchical path to execute component overrides

```
class environment extends uvm_env;
  ...
  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    agt = agent::type_id::create("agt", this);
  endfunction
endclass
```

```
class new_agt extends agent;
  `uvm_component_utils(new_agt)
  function new(string name, uvm_component parent);
    virtual task class_task(...);
      // modified component functionality
    endtask
  endclass
```

Simulate with:
simv +UVM_TESTNAME=test_new

create() used to construct component

Modify operation

```
graph TD
  simv[simv] --> uvm_root[uvm_root]
  uvm_root --> run_test[run_test()]
  test_new[test_new] --> uvm_test_top[uvm_test_top]
  environment[environment] --> env[env]
  new_agt[new_agt] --> agt[agt]
```

2-36

An important concept in UVM is that your testbench classes such as the environment, agent, drivers, etc, are written once at the start of the project, and modified very rarely if ever.

You can still change the behavior of components by using “hooks” such as the UVM factory to change the behavior of a testbench, without touching the existing code.

The UVM factory by default constructs the instantiated components, but enables you to replace it with a derived version.

In this example the *agent* object is being replaced by a *new_agt* object.

At the top level, the test tells the UVM factory to override the *agent* class with *new_agt*. Afterwards, when the environment constructs the *agt* object using the factory, it will get an object of *new_agt* class.

Unit Objectives Review

You should now be able to:

- Describe the process of reaching verification goals
- Describe the UVM testbench architecture
- Describe the different components of a UVM testbench
- Bring different components together to create a UVM environment



2-37

Appendix

Major Changes from UVM-1.1 to UVM-1.2

Calling UVM Messaging From Modules

Catch and Throw UVM Messages

UVM Reporter Control

UVM Command Line Processor

VCS Support for UVM

2-38

Migration from UVM1.1 to UVM1.2

- Non-Backward compatible changes outlined in migration document
- Migration script:
 - `./bin/uvm11-to-uvm12.pl`
- Release note:
 - `release-notes.txt` lists mantis items and backward compatibility

2-39

Global Handles

- Global handles `uvm_top` and `factory` in `uvm_pkg` have been removed

```
function void test_base::start_of_simulation_phase(uvm_phase phase);  
    super.start_of_simulation_phase(phase);  
    uvm_top.print_topology(); // Will not compile in UVM-1.2  
    factory.print(); // Will not compile in UVM-1.2  
endfunction
```



- Call the `get()` method of the class to retrieve the singleton handle

```
function void test_base::start_of_simulation_phase(uvm_phase phase);  
    super.start_of_simulation_phase(phase);  
    uvm_root::get().print_topology(); // Works in UVM-1.1 & UVM-1.2  
    uvm_factory::get().print(); // Works in UVM-1.1 & UVM-1.2  
endfunction
```

- Call get methods of the core service object

```
uvm_coreservice_t cs = uvm_coreservice_t::get();  
cs.get_root().print_topology(); // Will not work in UVM-1.1  
cs.get_factory().print(); // Will not work in UVM-1.1
```



2-40

`uvm_coreservice_t` is the uvm-1.2 mechanism for accessing all the central UVM services such as `uvm_root`, `uvm_factory`, `uvm_report_server`, etc.

The class provides a static `get()` method which returns the singleton handle of the class with `set_<facility>` `get_<facility>` pairs providing access to these services

`get_root()` returns the `uvm_root` instance

`get_factory()` returns the currently enabled `uvm_factory`

`get_report_server()` returns the current global `report_server`

`get_default_tr_database()` returns the current default record database

`get_component_visitor()` retrieves the current component visitor

(the visitor is being used for the traversal at `end_of_elaboration_phase` for instance, name checking)

`set_factory()` sets the current `uvm_factory`

`set_report_server()` sets the central `report_server`

`set_default_tr_database()` sets the current default record database

`set_component_visitor()` sets the component visitor

Sequences

■ Command line implicit sequence execution

```
+uvm_set_default_sequence=*.i_agt.sqr.main_phase,my_seq
```

■ starting_phase has been removed

- Either call `set_automatic_phase_objection()` method
- Or call `get/set_starting_phase()` method instead

```
function my_seq::new(string name="my_seq");
    super.new(name);
    set_automatic_phase_objection(1);           // UVM-1.2 ONLY!
endfunction

task my_seq::pre_start();
    get_starting_phase().raise_objection(this); // UVM-1.2 ONLY!
endtask
task my_seq::post_start();
    get_starting_phase().drop_objection(this);  // UVM-1.2 ONLY!
endtask
```

2-41

Phasing

- **phase.get_objection_count()** returns pending objection count for the phase
- Phase state transitions callback

```
class my_cb_class extends uvm_phase_cb; // constructor not shown
    virtual function void phase_state_change(uvm_phase phase,
                                              uvm_phase_state_change change);
        uvm_phase_state state = change.get_state();
        `uvm_info("CALLBACK",
                  $sformatf("State changed to %p for phase %s",
                            state, phase.get_name()), UVM_FULL);
    endfunction
endclass
```

```
my_cb_class my_cb_inst = new();
uvm_callbacks#(uvm_phase, uvm_phase_cb)::add(null, my_cb_inst);
```

2-42

UVM_INFO test340.sv(29) @ 0: reporter [CALLBACK] State changed to UVM_PHASE_ENDED for phase end_of_elaboration
UVM_INFO test340.sv(29) @ 0: reporter [CALLBACK] State changed to UVM_PHASE_CLEANUP for phase end_of_elaboration
UVM_INFO test340.sv(29) @ 0: reporter [CALLBACK] State changed to UVM_PHASE_DONE for phase end_of_elaboration
UVM_INFO test340.sv(29) @ 0: reporter [CALLBACK] State changed to UVM_PHASE_SCHEDULED for phase start_of_simulation
UVM_INFO test340.sv(29) @ 0: reporter [CALLBACK] State changed to UVM_PHASE_SYNCING for phase start_of_simulation
UVM_INFO test340.sv(29) @ 0: reporter [CALLBACK] State changed to UVM_PHASE_STARTED for phase start_of_simulation
UVM_INFO test340.sv(29) @ 0: reporter [CALLBACK] State changed to UVM_PHASE_EXECUTING for phase start_of_simulation
UVM_INFO test340.sv(29) @ 0: reporter [CALLBACK] State changed to UVM_PHASE_ENDED for phase start_of_simulation
UVM_INFO test340.sv(29) @ 0: reporter [CALLBACK] State changed to UVM_PHASE_CLEANUP for phase start_of_simulation

uvm_config_db

- **set_config_* and get_config_* are deprecated**
 - Migration script can make the conversion to uvm_config_*
- **Meta characters / regex in field names are deprecated**
 - Due to problems encountered in usage

```
uvm_config_int::set(this, "", "/z?mycomplexint/", 4);  
uvm_config_string::set(this, "", "/mycomplexint/", "xx");  
uvm_config_int::set(this, "", "/my*int/", 2);  
uvm_config_int::set(this, "", "/my_complex.*/", 3);
```

2- 43

Reporting

■ New message macros

- `uvm_*_begin`, `uvm_*_end`
- Add value/string/object print



Not in IEEE LRM

```
uvm_report_message msg = new("msg");
`uvm_info_begin("MY_ID", "My message", UVM_MEDIUM, msg)
  `uvm_message_add_tag("tag_label", "MY_MESSAGE")
  `uvm_message_add_int(my_int, UVM_HEX, "int_label")
  `uvm_message_add_string("my_string", "string_label")
  `uvm_message_add_object(my_object, "object_label")
`uvm_info_end
```

■ User can record for debugger display

- Consult your EDA vendor on support for this feature

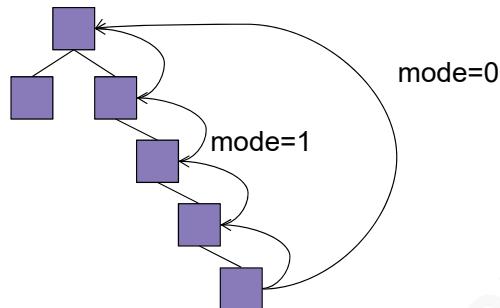
```
uvm_text_tr_database msg_db = new("msg_db");
uvm_tr_stream      stream = msg_db.open_stream("my_stream");
uvm_recorder       rec   = stream.open_recorder("my_recorder");
msg.record(rec);
uvm_process_report_message(msg);
```

2-44

IEEE committee has decided that no debugging mechanism shall be included in the IEEE LRM. For Accellera source code or EDA vendor source code, these may exist. Please check with your EDA vendor.

Objection Performance

- Use `set_propagate_mode()` in `uvm_objection` object of the phase to avoid rippling of objections through hierarchy
 - Useful for high-frequency raise/drop scenarios



2- 45

Appendix

Major Changes from UVM-1.1 to UVM-1.2

Calling UVM Messaging From Modules

Catch and Throw UVM Messages

UVM Reporter Control

UVM Command Line Processor

VCS Support for UVM

2- 46

Calling UVM Messaging From Modules (1/2)

- The UVM messaging works well in the testbench, but what about non-SystemVerilog RTL?

- Create a global module to connect RTL and UVM

```
module GLOBAL;
    import uvm_pkg::*;
    //enumeration literals have to be separately imported
    import uvm_pkg::UVM_HIGH; // And all the rest...
    // Verilog-2001 compatible function
    function reg uvm_info(input logic[100*8:1] ID,
                          input logic[1000*8:1] message,
                          input int           verbosity);
        `uvm_info(ID, message, verbosity);
        uvm_info = 1;
    endfunction
endmodule
//optional macros
`define G_UVM_HIGH      GLOBAL.UVM_HIGH
`define G_UVM_INFO(i,m,v) GLOBAL.uvm_info(i,m,v)
```

2-47

This is a SystemVerilog module that contains functions that can be called from non-SystemVerilog code such as legacy RTL

- Use *GLOBAL* as the module name as “global” is a reserved keyword in SystemVerilog.
- Import the specific enum values such as **UVM_LOW**, **UVM_MEDIUM**, etc. from the UVM package
- Create a wrapper function for every UVM messaging macro such as **uvm_fatal**, **uvm_error**, **uvm_warning**
- Call the macro from inside the function, and return a default value
- (Optional) Create a set of macros to simplify calling the functions and the enumerated values

Calling UVM Messaging From Modules (2/2)

- Call UVM messaging from RTL

- Code does not require SystemVerilog constructs

```
module dut (...);  
    parameter DISPLAY_WIDTH = 1000;  
    reg [DISPLAY_WIDTH*8:1] message;  
  
    always @ (negedge reset_n) begin  
        message = "reset asserted";  
        `G_UVM_INFO("RESET", message, `G_UVM_HIGH);  
    end  
endmodule
```

2- 48

The example shows non-SystemVerilog code that uses the UVM reporting functions. If your modules are SystemVerilog, you can use the UVM macros directly.

The message variable is for formatted strings. You can pass strings directly, as shown in the ID argument.

Appendix

Major Changes from UVM-1.1 to UVM-1.2

Calling UVM Messaging From Modules

Catch and Throw UVM Messages

UVM Reporter Control

UVM Command Line Processor

VCS Support for UVM

2- 49

Catch and Throw UVM Report Messages

```
class my_error_demoter extends uvm_report_catcher;
  // uvm_object_utils and constructor not shown
  // This example demotes "CFGERR" Error
  function action_e catch();
    if(get_severity() == UVM_ERROR && get_id() == "CFGERR")
      set_severity(UVM_WARNING);
    return THROW;
  endfunction
endclass

class test_report extends test_base;
  // uvm_component_utils and constructor not shown
  function void build_phase(uvm_phase phase);
    my_error_demoter demoter = new();
    super.build_phase(phase);
    // To affect all reporters, use null for the object
    uvm_report_cb::add(null, demoter);
    // To affect some set of components use the component name
    uvm_report_cb::add_by_name("*.drv*", demoter, this);
  endfunction
endclass
```

2- 50

Appendix

Major Changes from UVM-1.1 to UVM-1.2

Calling UVM Messaging From Modules

Catch and Throw UVM Messages

UVM Reporter Control

UVM Command Line Processor

VCS Support for UVM

2- 51

Embed UVM Reporter Control in Test

```
class report_control extends test_base;
  UVM_FILE log_file = $fopen("log_file", "w") // log file
  // uvm_component_utils and constructor not shown
  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    uvm_root::get().set_* (see method below)
  endfunction
  function void final_phase(uvm_phase phase);
    super.final_phase(phase);
    $fclose(log_file);
  endfunction
endclass
```

```
function void set_report_verbosity_level(int verbosity_level);
function void set_report_id_verbosity(string id, int verbosity);
function void set_report_severity_id_verbosity(uvm_severity severity, string id, int verbosity);
function void set_report_severity_action(uvm_severity severity, uvm_action action);
function void set_report_id_action(string id, uvm_action action);
function void set_report_severity_id_action(uvm_severity severity, string id, uvm_action action);
function void set_report_default_file(UVM_FILE file);
function void set_report_id_file(string id, UVM_FILE file);
function void set_report_severity_file(uvm_severity severity, UVM_FILE file);
function void set_report_severity_id_file(uvm_severity severity, string id, UVM_FILE file);
function void set_report_severity_override(uvm_severity cur_severity, uvm_severity new_severity);
function void set_report_severity_id_override(uvm_severity cur_severity, string id, uvm_severity new_severity);
```

2-52

Appendix

Major Changes from UVM-1.1 to UVM-1.2

Calling UVM Messaging From Modules

Catch and Throw UVM Messages

UVM Reporter Control

UVM Command Line Processor

VCS Support for UVM

2- 53

UVM Command Line Processor

■ User can query run-time arguments

```
class test_base extends uvm_test; // simplified code
  uvm_cmdline_processor clp = uvm_cmdline_processor::get_inst();
  string t, v, args[$], m;
  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase); // simplified code
    t = clp.get_tool_name();
    v = clp.get_tool_version();
    `uvm_info("TOOL", $sformatf("Tool: %s, Version: %s", t, v), UVM_LOW);
    clp.get_arg_value("+packet_item_count=", value);
    uvm_config_db#(int)::set(this, "*", "item_count", value.atoi());
    clp.get_args(args); // retrieve all plusargs
    clp.get_arg_matches("+ces_uvm_plusargs=", args); // retrieve args
    clp.get_arg_values("+ces_uvm_plusargs=", args); // retrieve values
    foreach (args[i]) begin
      $sformat(m, {m, args[i], " "});
    end
    `uvm_info("ARGS", $sformatf("Simv cmd values are:%s", m), UVM_LOW);
  endfunction
endclass
```

2-54

simv +UVM_TESTNAME=test_base +packet_item_count=20

The complete list of get arguments are as follows:

```
function void get_args (output string args[$]);
function void get_plusargs (output string args[$]);
function void get_uvm_args (output string args[$]);
function int get_arg_matches (string match, ref string args[$]);
function int get_arg_value (string match, ref string value);
function int get_arg_values (string match, ref string values[$]);
```

Please see UVM class reference document for details

Appendix

Major Changes from UVM-1.1 to UVM-1.2

Calling UVM Messaging From Modules

Catch and Throw UVM Messages

UVM Reporter Control

UVM Command Line Processor

VCS Support for UVM

2- 55

Compiling UVM with VCS

■ Single compile flow

```
% vcs -sverilog file.sv ... -ntb_opts uvm-1.2 ...
```

■ UUM compile flow

Compile UVM library
first with no source files

```
% vlogan -sverilog -work work1 -ntb_opts uvm-1.2  
% vlogan -sverilog -work work1 -ntb_opts uvm-1.2 file1.v  
% vlogan -sverilog -work work2 -ntb_opts uvm-1.2 file2.v  
% vhdlan -work work3 file3.vhd  
% vcs top ... -ntb_opts uvm-1.2
```

When using the VPI-based backdoor access mechanism included in the UVM library, the "+acc" and "+vpi" command-line options must also be used.

2- 56

From VCS 2015.09 onwards, VCS ships with both uvm-1.1 and uvm-1.2 source code.

There are multiple switches possible for –ntb_opt: uvm, uvm-1.1 and uvm-1.2.

For each release of VCS, there is a default version of the official UVM source code supported. If you want to use the default version, then use –ntb_opt uvm. However, you need to be careful with this switch. VCS 2018.09's default UVM version is uvm-1.1. Other VCS versions may default to other UVM version.

For user customized UVM source code, if one wants to enable the recording one would need to compile as follows (for uvm-1.2):

```
% setenv VCS_UVM_HOME <path to UVM/src>  
% vcs -sverilog test.sv +incdir+${VCS_HOME}/etc/uvm-1.2/vcs\  
${VCS_HOME}/etc/uvm-1.2/vcs/uvm_custom_install_vcs_recorder.sv
```

For IEEE UVM source code, transaction recording is not yet implemented.

Agenda

DAY

1

1 OOP Inheritance Review

2 UVM Structural Overview



3 UVM Transaction



4 UVM Sequence



Synopsys 40-I-055-SSG-007

© 2019 Synopsys, Inc. All Rights Reserved

3-1

Unit Objectives

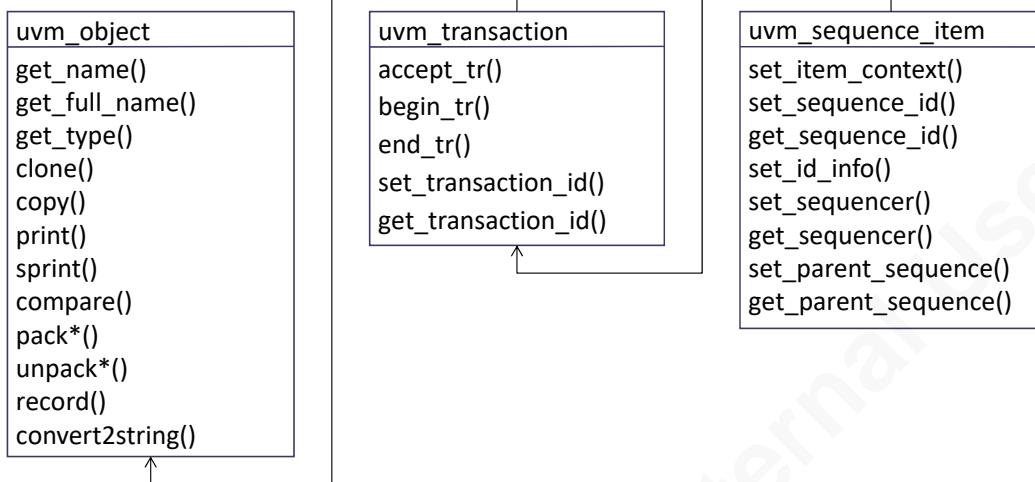


After completing this unit, you should be able to:

- Build data models by inheriting from `uvm_sequence_item`
- Use macros or implement method to enable processing of `uvm_sequence_item` fields
- Modify constraint with inheritance and override
- Implement parameterized classes
- Simplify report messages

3-2

UVM Transaction Base Classes

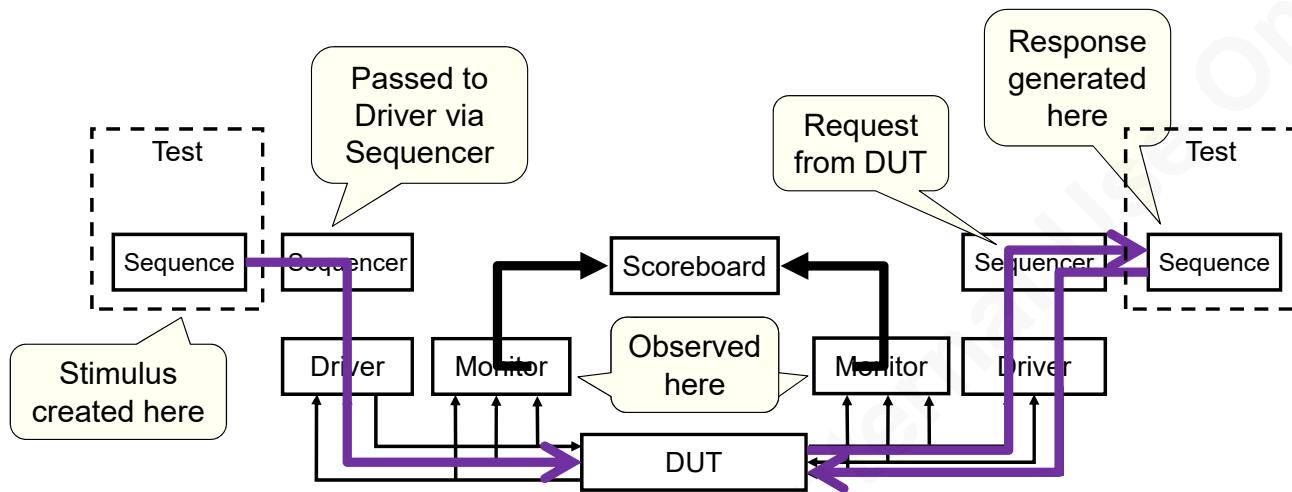


3-3

The `uvm_object` and `uvm_transaction` classes are virtual classes.

UVM Transaction Flow

- Transactions typically do not have a fixed component parent



3-4

Modeling Transactions

- Derive from **uvm_sequence_item** base class
 - Built-in support for stimulus creation, printing, comparing, etc.
- Properties should be public by default 
- Properties should be rand by default
 - Must be visible to constraints in other classes
- Properties should be rand by default
 - Can be turned off with rand_mode

```
class packet extends uvm_sequence_item;
  `uvm_object_utils(packet)
  rand bit [47:0] sa, da;
  rand bit [15:0] len;
  rand bit [ 7:0] payload[$];
  rand bit [31:0] crc;
  function new(string name = "packet");
    super.new(name);
    this.crc.rand_mode(0);
  endfunction
endclass
```

No **uvm_component** parent

Default required

3-5

For your transaction class, don't extend from **uvm_transaction**. Extend from **uvm_sequence_item**, which has additional properties and methods, so that these transactions can be used in a sequence.

- If you don't make a variable public, then it can not be controlled in external random constraints.
- If you don't make a variable random, it can not be made random later on. But a random variable can be made non-random by calling **handle.var.rand_mode(0)**

While OOP usually recommends to hide data and not allow access from outside, this does not apply to randomization in verification. Tests need to bend, distort, rip, and break protocols to make sure every corner case is covered.

Other Properties to be Considered (1/2)

■ Embed transaction descriptor

- Component interprets transaction to execute

```
class cpu_data extends uvm_sequence_item;
  typedef enum {READ, WRITE} kind_e;
  rand int delay = 0;
  rand kind_e kind;
  rand bit [31:0] addr, data;
  function new(string name="cpu_data");
    super.new(name);
    this.delay.rand_mode(0);
  endfunction
endclass
```

```
class cpu_driver extends uvm_driver #(cpu_data);
  virtual task execute(cpu_data tr);
    repeat(tr.delay) @(vif.drvClk);
    case (tr.kind) begin
      cpu_data::READ:
        tr.data = this.read(tr.addr);
      cpu_data::WRITE:
        this.write(tr.addr, tr.data);
    endcase
  endtask
endclass
```

3-6

The *cpu_data* class contains a single transaction. The *kind* property is a random enumerated variable that tells the flavor of the transaction.

The *cpu_driver* class is a UVM component. The *run_phase()* method, not shown, calls the *execute()* method to interpret the CPU transaction.

Other Properties to be Considered (2/2)

■ Embed transaction status flags

- Set by component for execution status

```
class cpu_data extends uvm_sequence_item;
    typedef enum {IS_OK, ERROR, HAS_X} status_e;
    rand status_e status = IS_OK; ...
    function new(string name="cpu_data");
        super.new(name); ...
        this.status.rand_mode(0);
    endfunction
endclass

class cpu_driver extend uvm_driver #(cpu_data);
    virtual task execute(cpu_data tr);
        repeat(tr.delay) @(vif.drvClk);
        case (tr.kind) begin ... endcase
        if (error_condition_encountered) begin
            tr.status = cpu_data::ERROR;
            `uvm_info("DRV_ERR", tr.sprint(), UVM_HIGH);
        end
    endtask
endclass
```

3-7

Transactions: Must-Obey Constraints

- Define constraint block for the **must-obey** constraints
 - Never turned off
 - Never overridden
 - Name "class_name_valid"
- Example:
 - Non-negative values for `int` properties

```
class packet extends uvm_sequence_item;
    rand int len;
    ...
    constraint packet_valid {
        len > 0;
    }
endclass
```

3-8

These constraints must always be satisfied otherwise the transaction is not valid.

Transactions: Should-Obey Constraints

■ Define constraint block for **should-obey** constraints

- Can be turned off to inject errors
- One block per relationship set
 - ◆ Can be individually turned off or overloaded
- Name "class_name_rule"

```
class packet extends uvm_sequence_item;
    ...
    constraint packet_sa_local {
        sa[41:40] == 2'b0;
    }
    constraint packet_ieee {
        len inside { [46:1500] };
        data.size() == len;
    }
    ...
    ...
    constraint packet_fcs {
        crc == 32'h0000_0000;
    }
endclass
```

3-9

By default, testbenches produce good transactions as fast as possible.

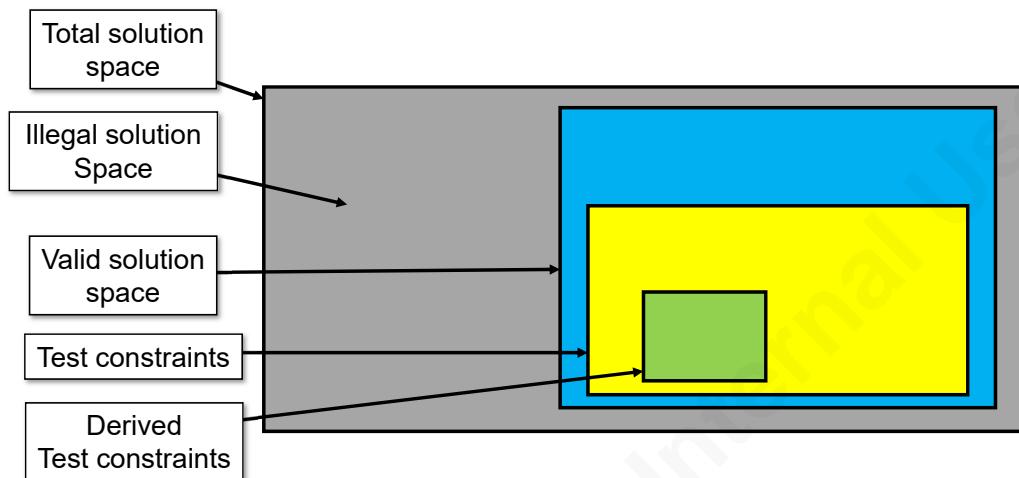
Should-obey constraints make sure errors are not injected unless you want to.

These constraints should be made as small as possible so that you have fine grain control of error injection.

Transactions: Constraint Considerations

- **Can't accidentally violate valid constraints**

- Constraint solver will fail if the user constraints conflict with valid constraints



3-10

Transaction Class Methods

- Transaction processing methods are not virtual!

For debugging

```
virtual function void print(uvm_printer printer = null)
virtual function string sprint(uvm_printer printer = null)
virtual function uvm_object clone(); // exception!
virtual function void copy(uvm_object rhs);
virtual function bit compare(uvm_object rhs,
                           uvm_comparer comparer = null);
virtual function int pack(ref bit bitstream[],
                         input uvm_packer packer = null);
virtual function int unpack(ref bit bitstream[],
                           input uvm_packer packer = null)
virtual function void record(uvm_recorder recorder = null);
```

For making copies

```
virtual function uvm_object clone(); // exception!
```

Used by checker

```
virtual function bit compare(uvm_object rhs,
```

4

~~virtual~~ function int **pack**(ref bit bitstream[],

Used by transactors

~~virtual~~ function int pack(ref bit bitstream[],
 input_wm_packer pac)

Use to record
transactions for
debugger

```
virtual function int unpack(ref bit bitstream[],  
                           input uvm_packer packer = null)  
virtual function void pack(uvm_packer packer = null)
```

```
virtual function int unpack(ref bit bitstream[],  
                           input uvm_packer packet);  
virtual function void unpack(uvm_message message);
```

```
virtual function void record(uvm_recorder recorder = null);
```

- User must NOT override these methods



- The non-virtual method calls two virtual methods

3- 11

With the exception of the `clone()` method, these transaction support methods are not virtual.

As was mentioned in OOP Inheritance Review (Unit 1), base class developers typically do not want the user to change the usage model of a user accessible public method. The way to accomplish this is to make the public accessible methods non-virtual. These are such examples.

To allow user to supplement these non-virtual methods, these methods do call override-able virtual methods.

For the UVM classes, the two override-able methods called by the non-virtual methods are: `do_*()` and `m_field_automation()` (see next slides).

The `do_*` methods can be manually populated by the user. Whereas the `m_field_automation()` method is populated by the `^uvm_field_*` macros (as shown in later slides).

Customization of Field Processing Methods

- The first way for processing fields of the transaction class is through user implementation of the following `do_` methods

```
virtual function void do_print(uvm_printer printer = null)
virtual function void do_copy(uvm_object rhs);
virtual function bit do_compare(uvm_object rhs,
                                uvm_comparer comparer = null);
virtual function int do_pack(ref bit bitstream[],
                            input uvm_packer packer = null);
virtual function int do_unpack(ref bit bitstream[],
                            input uvm_packer packer = null);
virtual function void do_record(uvm_recorder recorder = null);
```

3-12

Example for `do_print`:

```
class mytype extends uvm_object;
  data_obj data;
  int f1;
  string my_name;
`uvm_object_utils(mytype)
virtual function void do_print (uvm_printer printer);
  super.do_print(printer);
  printer.print_field("f1", f1, $bits(f1), UVM_DEC);
  printer.print_string("my_name", my_name);
  printer.print_object("data", data);
endfunction
endclass
```

Field Automation Enabled Field Processing

- The alternative way of processing the transaction fields is through the `uvm_field macros

- The following actions are supported via the macro:

Print, copy, compare, record, pack (enabled by UVM_ALL_ON flag)  Please see note

Individual field processing
are enabled with
`uvm_field_macros

```
class packet extends uvm_sequence_item;
    rand bit [47:0] sa, da;
    rand bit [7:0] payload[$];
    packet next;
`uvm_object_utils_begin(packet)
`uvm_field_int(sa, UVM_ALL_ON | UVM_NOCOMPARE)
`uvm_field_int(da, UVM_ALL_ON)
`uvm_field_queue_int(payload, UVM_ALL_ON)
`uvm_field_object(next, UVM_ALL_ON)
`uvm_object_utils_end
endclass
```

Class registry macro must have _begin

Class registry macro must end with _end

3-13

The UVM transaction methods are print, copy, compare, etc. that are used by the testbench to manipulate transaction objects. These will be described in a few slides.

The uvm_object_utils_begin(T) macro expands into:

- Register the class T in the registry so it can be created and overridden from the factory
- Define a proxy class to create (construct) the object, as needed by the factory
- Define a virtual method that returns the name of the object
- Define the first part of the function **m_field_automation()** that provides functionality for the print, copy, compare, etc. methods

The uvm_field_macros expand into case statements to perform the print, copy, compare actions.

The uvm_object_utils_end finishes the **m_field_automation()** method.

See appendix for more details on uvm_object_utils macro.

Note: In UVM documentation, you will also see a UVM_DEFAULT setting. It is set to UVM_ALL_ON by default in IEEE UVM. Recommendation: use UVM_ALL_ON to avoid confusion.

In Accellera uvm-1.2, user is required to enter the 2nd argument of the `uvm_field_* macro. Leaving the 2nd argument out results in compilation error.

IEEE P1800.2 changes this by adding a default to the 2nd argument as follows:

`uvm_field_int(ARG, FLAG=UVM_DEFAULT)

`uvm_field_* Field Automation Macros

■ Scalar and array properties are supported

```
`uvm_field_int(ARG, FLAG)          // Any scalar numeric property
`uvm_field_real(ARG, FLAG)
`uvm_field_event(ARG, FLAG)
`uvm_field_object(ARG, FLAG)
`uvm_field_string(ARG, FLAG)
`uvm_field_enum(T, ARG, FLAG)      // T - enum data type
`uvm_field_sarray_*(ARG, FLAG)    // fixed size array: _type
`uvm_field_array_*(ARG, FLAG)     // dynamic array:       _type
`uvm_field_queue_*(ARG, FLAG)     // queue:           _type
`uvm_field_aa_*_*(ARG, FLAG)      // associative array: _type_index
```

■ For object recursion, default is UVM_DEEP

- Copy nested object

■ Can be changed to UVM_SHALLOW* Please see note

- Copy object handle

```
// changing to UVM_SHALLOW
`uvm_field_object(object, UVM_ALL_ON | UVM_SHALLOW)
```

3-14

Examples of array macros:

```
`uvm_field_sarray_int(my_ints, UVM_ALL_ON) // Fixed array of integrals values
`uvm_field_array_real(my_reals, UVM_ALL_ON) // dynamic array of reals
`uvm_field_queue_object(my_objs, UVM_ALL_ON) // queue of uvm_object
`uvm_field_aa_int_string(lookup, UVM_ALL_ON) // associative array of integrals values indexed with string
```

Important note!!!

In UVM-1.1 and 1.2, there is a major bug with UVM_SHALLOW, it does NOT work! To get shallow copy, you need to set the FLAG to UVM_REFERENCE! This makes no sense.

IEEE UVM now corrected this bug. In Accellera IEEE UVM UVM_SHALLOW will work as intended. UVM_REFERENCE is deprecated and will no longer compile under IEEE UVM.

Print Radix Specified by FLAG

- Additional FLAG can be used to set radix for print method

Radix for printing and recording can be specified by OR'ing one of the following constants in FLAG argument

UVM_BIN	- Print/record field in binary (base-2)
UVM_DEC	- Print/record field in decimal (base-10)
UVM_UNSIGNED	- Print/record field in unsigned decimal (base-10)
UVM_OCT	- Print/record the field in octal (base-8)
UVM_HEX	- Print/record the field in hexadecimal (base-16)
UVM_STRING	- Print/record the field in string format
UVM_TIME	- Print/record the field in time format
UVM_REAL	- Print/record the field in floating number format

```
// printing in decimal  
`uvm_field_int(field, UVM_ALL_ON | UVM_DEC)
```

3-15

Examples of Using Transaction Methods

```
packet pkt0, pkt1, pkt2;           Name string
bit bit_stream[];
pkt0 = packet::type_id::create("pkt0");
pkt1 = packet::type_id::create("pkt1");
pkt0.sa = 10;
pkt0.print();                      // display content of object on stdio
pkt0.copy(pkt1);                  // copy content of pkt1 into memory of pkt0
                                   // name string is not copied
$cast(pkt2, pkt1.clone());        // make pkt2 an exact duplication of pkt1
                                   // name string is copied
if(!pkt0.compare(pkt2)) begin      // compare the contents of pkt0 against pkt2
  `uvm_error("MISMATCH", "Error found");
end
pkt0.pack(bit_stream);            // pack content of pkt0 into bit_stream array
pkt2.unpack(bit_stream);          // unpack bit_stream array into pkt2 object
```

3-16

The **clone()** method returns an handle of type `uvm_object`, so you need `$cast` to assign this to your transaction handle.

`pkt2 = pkt1.clone();` won't compile, as `clone()` returns a base type

The pack/unpack methods has the following variation:

<code>pack</code>	- <code>bit</code> array
<code>pack_bytes</code>	- <code>byte unsigned</code> array
<code>pack_ints</code>	- <code>int unsigned</code> array
<code>pack_longint</code>	- <code>longint unsigned</code> array

Modify Constraint in Transactions by Type

```
class packet extends uvm_sequence_item; // utils macro and constructor not shown
  rand bit[3:0] sa, da;
  rand bit[7:0] payload[];
  constraint valid {payload.size() inside {[2:10]};}
  ...
endclass

class packet_da_3 extends packet;
  constraint da_3 {da == 3;}
  `uvm_object_utils(packet_da_3)
  function new(string name = "packet_da_3");
    super.new(name);
  endfunction
endclass

class test_da_3_type extends test_base; // utils macro and constructor not shown
  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    //set_type_override_by_name("packet", "packet_da_3");
    set_type_override_by_type(packet::get_type(), packet_da_3::get_type());
  endfunction
endclass
```

The most common transaction modification is change/add constraint

Preferred override (compile-time check)

All **packet** instances are now **packet_da_3**

3-17

```
// Constructor for class packet
function new(string name = "packet");
  super.new(name);
endfunction
```

The *_override_by_type() is the preferred override method because the type check is done at compile-time whereas the *_override_by_name() method uses string as arguments resulting in type check done at run-time.

Compile-time error check is preferred over run-time check.

Alternatively, one can also do the type-wide override with:

```
packet::get_type().set_type_override(packet_da_3::get_type());
```

Transaction Replacement Results

Simulate with:

```
simv +UVM_TESTNAME=
```

```
+UVM_TESTNAME=
```

test_da_3_type

UVM_INFO @ 0: uvm_test_top.env.i_agt.drv [Normal] Item in Driver

```
req: (packet_da_3@95) {
```

```
    sa: 'h7
```

```
    da: 'h3
```

```
    ...
```

```
}
```

```
...
```

```
#### Factory Configuration (*)
```

```
Type Overrides:
```

Requested Type	Override Type
packet	packet_da_3

```
-----
```

```
-----
```

Factory configuration is displayed with a call to:

```
uvm_factory::get().print()
```

3-18

Modify Constraint in Transaction by Instance

```
class test_da_3_inst extends test_base; // utils and constructor not shown
    virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase);
    // set_inst_override_by_name("env.i_agt.sqr*", "packet", "packet_da_3");
        set_inst_override_by_type("env.i_agt.sqr*", packet::get_type(),
                                   packet_da_3::get_type());
    endfunction
endclass
```

Only **packet** instances in matching sequencers are now **packet_da_3**

Simulate with:

```
simv +UVM_TESTNAME=test_da_3_inst
```

```
UVM_INFO @ 0: uvm_test_top.env.i_agt.drv [Normal] Item in Driver
req: (packet_da_3@95) {
```

```
    sa: 'h7
    da: 'h3
}
```

Instance Overrides:

Requested Type	Override Path	Override Type
packet	uvm_test_top.env.i_agt.sqr.*	packet_da_3

3-19

Alternatively, one can also do the instance override with:

```
packet::get_type().set_inst_override(packet_da_3::get_type(),"*.req");
```

Command-line Override

- User can override factory objects at command line

- Objects must be constructed with the factory class_name::type_id::create(...) method

```
+uvm_set_inst_override=<req_type>,<override_type>,<inst_path>
```

```
+uvm_set_type_override=<req_type>,<override_type>
```

- Works like:

- `set_inst_override()`
- `set_type_override()`

Example:

```
+uvm_set_inst_override=packet,my_packet,*.agt.*
```

```
+uvm_set_type_override=packet,my_packet
```

No space character!

3-20

Parameterized Transaction Class (1/3)

- Parameterized transaction requires a different macro

```
`uvm_object_param_utils(cname#(param))
```

Or

```
`uvm_object_param_utils_begin(cname#(param))
  `uvm_field_*(ARG, FLAG)
`uvm_object_utils_end
```

- Use typedef to make it more manageable

```
class my_data #(width=48) extends uvm_sequence_item;
  typedef my_data#(width) this_type;

  `uvm_object_param_utils(this_type)
  ...
endclass
```

3-21

Parameterized Transaction Class (2/3)

■ Must supplement macro with definition of type_name

- For non-parameterized classes, the macro creates these
- For parameterized classes, user must define these

```
class my_data #(width=48) extends uvm_sequence_item;
  typedef my_data#(width) this_type;
  `uvm_object_param_utils(this_type)
  const static string type_name = $sformatf("my_data#(%0d)", width);
  virtual function string get_type_name();
    return type_name;
  endfunction
  ...
endclass
```

■ If not done, print will show uvm_sequence_item as type

Name	Type	Size	Value

req	uvm_sequence_item	-	@1548
address	integral	4	'hc
data	integral	16	'hc472

3-22

Parameterized Transaction Class (3/3)

- Creation of parameterized transaction object requires parameter (even if none is needed)

```
class my_component extends uvm_component;
    my_data d;
    virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        d = my_data#()::type_id::create("d", this);
    endfunction
endclass
```



Will not compile without #()

- Use typedef to eliminate potential problem

```
class my_component extends uvm_component;
    typedef my_data#() my_data_t;
    my_data_t d;
    virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        d = my_data_t::type_id::create("d", this);
    endfunction
endclass
```

3-23

Simplifying Report Messages

■ The `print/sprint()` methods may be too verbose

- Good for debug, bad for quick analysis

```
task driver::run_phase(uvm_phase phase);
  forever begin
    seq_item_port.get_next_item(req);
    `uvm_info("RUN", { "\n", req.sprint() }, UVM_MEDIUM);
    ...
  end
endtask
```

```
UVM_INFO driver.sv(34) @ 12.0ns: uvm_test_top.env.agt.drv [RUN]
```

Name	Type	Size	Value

req	packet	-	@1548
sa	integral	4	'h3
da	integral	4	'h4
...			

3-24

Applying convert2string()

- Implement and use convert2string() method

```
function string packet::convert2string();
    return $sformatf("sa = %2d, da = %2d", sa, da);
endfunction

task driver::run_phase(uvm_phase phase);
    forever begin
        seq_item_port.get_next_item(req);
        `uvm_info("RUN", req.convert2string(), UVM_LOW);
        ...
    end
endtask
```

- Output now simplifies to

```
UVM_INFO driver.sv(34) @ 12.0ns: uvm_test_top.env.agt.drv [RUN] sa = 3, da = 4
UVM_INFO driver.sv(34) @ 24.0ns: uvm_test_top.env.agt.drv [RUN] sa = 14, da = 15
UVM_INFO driver.sv(34) @ 36.0ns: uvm_test_top.env.agt.drv [RUN] sa = 5, da = 1
UVM_INFO driver.sv(34) @ 48.0ns: uvm_test_top.env.agt.drv [RUN] sa = 12, da = 6
UVM_INFO driver.sv(34) @ 60.0ns: uvm_test_top.env.agt.drv [RUN] sa = 5, da = 1
```

3-25

Unit Objectives Review

You should now be able to:

- Build data models by inheriting from `uvm_sequence_item`
- Use macros or implement method to enable processing of `uvm_sequence_item` fields
- Modify constraint with inheritance and override
- Implement parameterized classes
- Simplify report messages



3-26

Appendix

uvm_object_utils Macro

uvm_object Class Common Members

3-27

uvm_object_utils Macro

`uvm_object_utils[_begin] (cname)

■ Implements:

```
typedef uvm_object_registry #(cname, "cname") type_id;
static const string type_name = "cname";
static function type_id get_type();
virtual function uvm_object_wrapper get_object_type();
virtual function string get_type_name();
```

Macro creates a proxy class called **type_id**

```
class uvm_object_registry #(type T=uvm_object, string Tname=<unknown>) extends uvm_object_wrapper;
typedef uvm_object_registry #(T,Tname) this_type;
const static string type_name = Tname;
local static this_type me = get();
static function this_type get();
if (me == null) begin
  uvm_factory f = uvm_factory::get();
  me = new;
  f.register(me);
end
return me;
endfunction

static function T create(string name="", uvm_component parent=null, string context(""));
static function void set_type_override(uvm_object_wrapper override_type,bit replace=1);
static function void set_inst_override(uvm_object_wrapper override_type,string inst_path,uvm_component parent=null);
virtual function string get_type_name();
virtual function uvm_object create_object(string name="");
endclass
```

Proxy class (**type_id**) is a service agent that creates objects of the class it represents

A proxy singleton object is registered in **uvm_factory** at beginning of simulation

Creation of objects is done via proxy class' (**type_id**) methods

3-28

Appendix

`uvm_object_utils` Macro

`uvm_object` Class Common Members

3-29

uvm_object Class Common Members

```
virtual class uvm_object extends uvm_void;
  extern function new (string name="";
  static bit use_uvm_seeding = 1;
  function void reseed ();
  virtual function void set_name (string name);
  virtual function string get_name ();
  virtual function string get_full_name ();
  virtual function int get_inst_id ();
  static function int get_inst_count();
  static function uvm_object_wrapper get_type ();
  virtual function uvm_object_wrapper get_object_type ();
  virtual function string get_type_name ();
  virtual function uvm_object clone ();
  virtual function string convert2string();
  function void print (uvm_printer printer=null);
  function string sprint (uvm_printer printer=null);
  function void record (uvm_recorder recorder=null);
  function void copy (uvm_object rhs);
  function bit compare (uvm_object rhs, uvm_comparer comparer=null);
  function int pack (ref bit bitstream[], input uvm_packer packer=null);
  function int pack_bytes (ref byte unsigned bytestream[], input uvm_packer packer=null);
  function int pack_ints (ref int unsigned intstream[], input uvm_packer packer=null);
  function int unpack (ref bit bitstream[], input uvm_packer packer=null);
  function int unpack_bytes (ref byte unsigned bytestream[], input uvm_packer packer=null);
  function int unpack_ints (ref int unsigned intstream[], input uvm_packer packer=null);
  virtual function * do_* (*); // * for print, record, copy, compare, pack and unpack
endclass
```

3-30

Agenda

DAY

1

1 OOP Inheritance Review

2 UVM Structural Overview



3 UVM Transaction



4 UVM Sequence



Synopsys 40-I-055-SSG-007

© 2019 Synopsys, Inc. All Rights Reserved

4-1

Unit Objectives



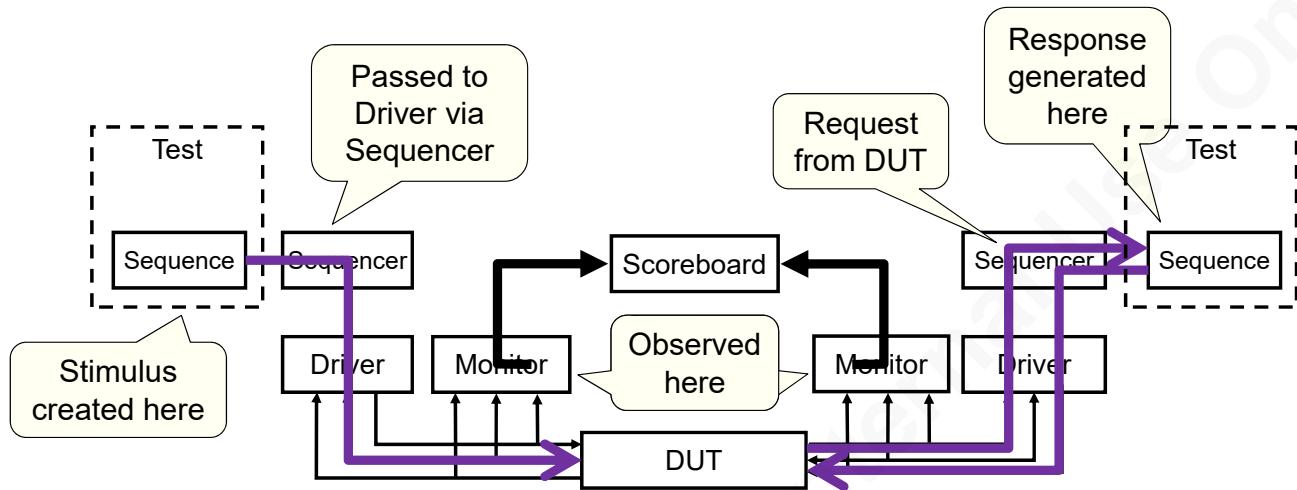
After completing this unit, you should be able to:

- Build stimulus generator class by inheriting from `uvm_sequence`
- Execute sequence explicitly and implicitly

4-2

UVM Transaction Flow - Continued

- Focused on sequence



4-3

Stimulus Generation Classes

uvm_object
get_name()
get_full_name()
get_type()
clone()
copy()
print()
sprint()
compare()
pack*()
unpack*()
record()
convert2string()

uvm_sequence_item
set_item_context()
set_transaction_id()
get_transaction_id()
set_sequence_id()
get_sequence_id()
set_id_info()
set_sequencer()
get_sequencer()
set_parent_sequence()
get_parent_sequence()

uvm_sequence_base
set_starting_phase()
get_starting_phase()
set_automatic_phase_objection()
starting_phase
start()
pre_start()
pre_body()
body()
start_item()
pre_do()
mid_do()
finish_item()
post_do()
post_body()
post_start()
lock()
unlock()
grab()
ungrab()

UVM 1.2 &
IEEE UVM

UVM 1.1

The differences between
these versions
are covered in this unit

uvm_sequence#(REQ, RSP)
REQ req
RSP rsp
get_response()

4-4

Sequence Class

- The stimulus generator in UVM is called a sequence
- User sequence must extend from uvm_sequence class and parameterized to the transaction type to be generated
 - Two handles are available:
 - ◆ `req` for request to driver and
 - ◆ `rsp` for response back from driver

The second parameter defaults to first parameter if not specified

uvm_sequence #(REQ, RSP)
REQ req
RSP rsp
get_response()

```
class packet_sequence extends uvm_sequence #(packet);  
  `uvm_object_utils(packet_sequence)  
  function new(string name = "packet_sequence");  
    super.new(name);  
  endfunction  
  // see next slide
```

The string argument in constructor must have a default value, typically the class name

4-5

Generate Transactions in Sequence Class

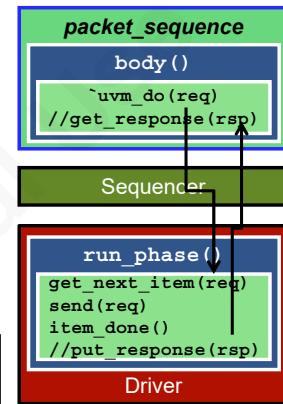
■ Sequence functional code resides in `body()` task

- ``uvm_do` creates, randomizes and passes transaction to driver through sequencer
- ``uvm_do_with` is the same as ``uvm_do` but with additional constraints
- Optional `get_response()` retrieves response from driver through sequencer
 - ◆ See appendix for response implementation

```
// Continued from previous slide
virtual task body();
    repeat (10) begin
        `uvm_do(req);
        // or with constraint: `uvm_do_with(req, {da == 3});
        // optional: get_response(rsp);
    end
endtask
endclass
```



Caution: IEEE is changing the ``uvm_do` macro
(see note)



4-6

A sequence generates one or more sequence items for the driver. Each sequence item is created, randomized and passed to the driver in the `body()` task, typically using a pre-defined UVM macros such as ``uvm_do()`. To simplify user code, in the `uvm_sequence` base class, there is a sequence item handle typed to the parameter class argument called `req` that can be used by the macro.

Change in IEEE UVM: ``uvm_do` is expanded to support sequencer, priority and constraint (``uvm_do_with` is not needed with IEEE UVM):

```
`uvm_do(SEQ_OR_ITEM, SEQR=get_sequencer(), PRIORITY=-1, CONSTRAINTS={})
```

If a sequence with inline constraint needs to support uvm-1.1, uvm-1.2 and uvm-ieee, then the code need to be written as follows:

```
virtual task body();
    repeat(10) begin
        `ifndef UVM_VERSION // IEEE versions will have this flag defined
            // For UVM-1.1 & UVM-1.2
            `uvm_do_with(req, {da == 3});
        `else
            // For IEEE UVM
            `uvm_do(req, , , {da == 3});
        `endif
    end
endtask
```

User Can Manually Create and Send Item

- **uvm_do macro effectively implements the following:**

```
// * - Equivalent code, not actual code
`define uvm_do(UVM_SEQ_ITEM) \
  `uvm_create(UVM_SEQ_ITEM) \
  start_item(UVM_SEQ_ITEM); \
  UVM_SEQ_ITEM.randomize(); \
  finish_item(UVM_SEQ_ITEM);
```



See note for IEEE definition

- **User can manually execute the methods:**

```
virtual task body();
// Instead of `uvm_do_with(req, {da == 3;})
// User can call these mechanisms individually
`uvm_create(req); // constructs sequence item and sets association with sequence
start_item(req); // wait for parent sequencer to get request from driver
req.randomize() with {da == 3;};
finish_item(req); // use parent sequencer to pass item to driver and wait for done
endtask
```

4-7

From IEEE P1800.2 definition

B.3.1.3 `uvm_rand_send

`uvm_rand_send(SEQ_OR_ITEM, PRIORITY=-1, CONSTRAINTS={})

This action processes a child item or sequence, with randomization.

a) For items, the implementation shall perform the following steps in order.

1) The **start_item** method (see 14.2.7.2) is called on this sequence, with *SEQ_OR_ITEM* as *item* and *PRIORITY* as *set_priority*.

2) *SEQ_OR_ITEM* is randomized with *CONSTRAINTS*.

3) The **finish_item** method (see 14.2.7.3) is called on this sequence, with *SEQ_OR_ITEM* as *item* and *PRIORITY* as *set_priority*.

B.3.1.4 `uvm_do

`uvm_do(SEQ_OR_ITEM, SEQR=get_sequencer(), PRIORITY=-1, CONSTRAINTS={})

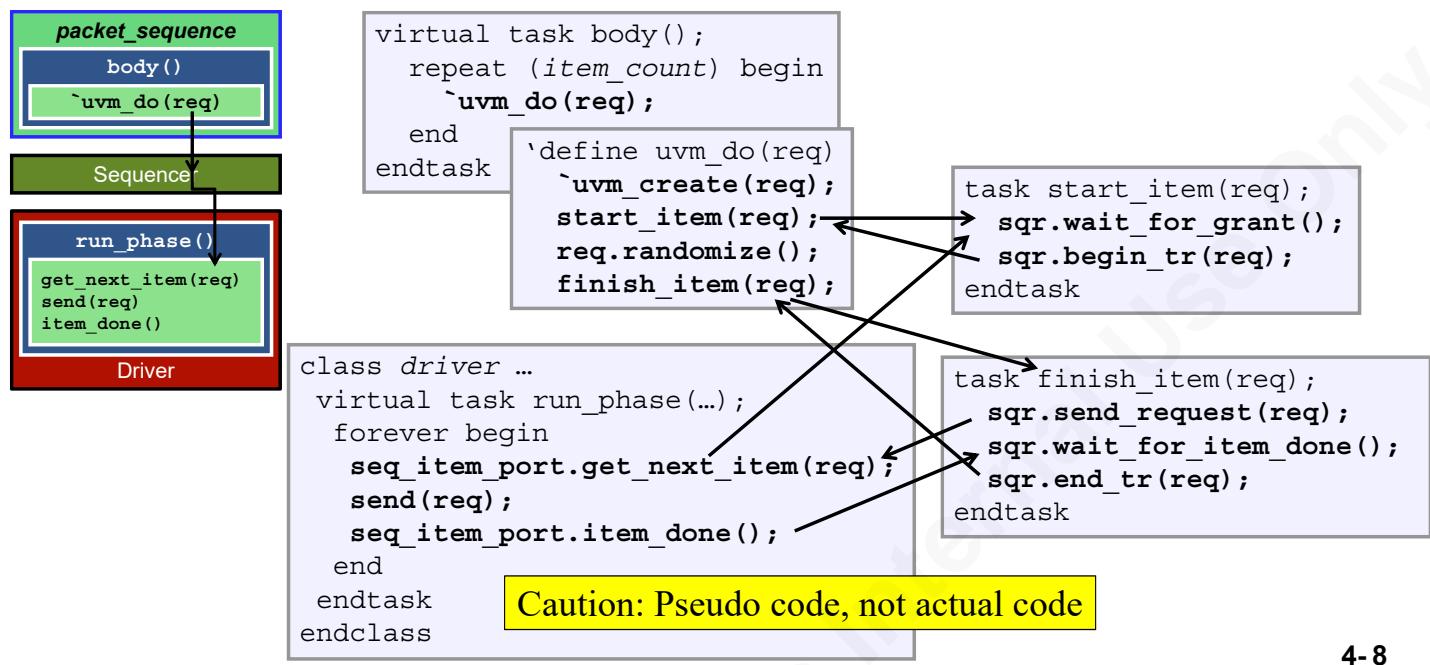
This action creates and processes a child item or sequence, with randomization.

The implementation shall perform the following steps in order.

a) **uvm_create** (see B.3.1.1) is passed *SEQ_OR_ITEM* and *SEQR*.

b) **uvm_rand_send** (see B.3.1.3) is passed *SEQ_OR_ITEM*, *PRIORITY*, and *CONSTRAINTS*.

`uvm_do Macro Interaction Detailed



4-8

This diagram shows the detailed control flow between the driver and sequencer.

- 1 – The sequence's ``uvm_do()` macro expands out to four major operations
- 2 – The sequence creates the transaction, then waits for its parent sequencer to grant permission to start the transaction.
- 3 – The granting process is initiated by the driver when it calls `get_next_item(req)` on the TLM port connected to the sequencer (`seq_item_port`).
- 4 – The sequencer arbitrates and grants sequence permission to continue.
- 5 – The sequence, then, marks the transaction as started (time stamped).
- 6 – Afterwards, the sequence randomizes the transaction and calls `finish_item(req)` to have the sequencer pass the transaction to the driver through the TLM port.
- 7 – The driver picks up the transaction, processes the transaction then issues a done acknowledgement through the TLM port.
- 8 – The sequence completes the macro execution by marking the transaction as ended (ending time stamp) and continues on to the next iteration within `body()`.

Side note:

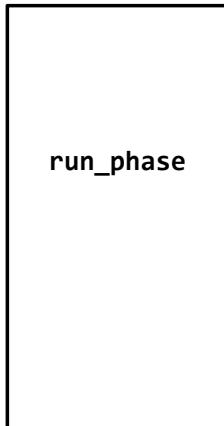
The auto recording of transactions can be turned off with the following compile-time switch:

`+define+UVM_DISABLE_AUTO_ITEM_RECORDING` (caution – globally switches off all auto recording)

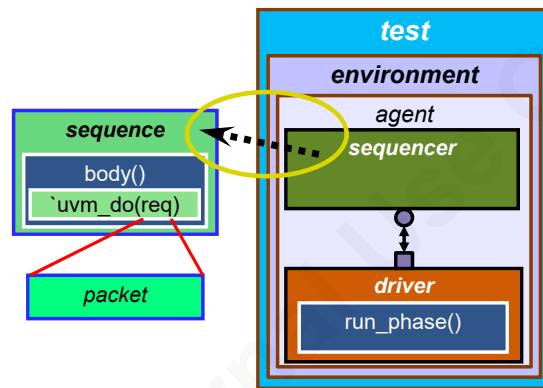
In UVM-1.2 and IEEE UVM, you can individually turn the auto recording off by calling the `disable_auto_item_recording()` method of the sequence item port (`seq_item_port`) of the driver.

Sequence Execution: Starting a Sequence

- Sequences are executed in task phases



```
pre_reset_phase  
reset_phase  
post_reset_phase  
pre_configure_phase  
configure_phase  
post_configure_phase  
pre_main_phase  
main_phase  
post_main_phase  
pre_shutdown_phase  
shutdown_phase  
post_shutdown_phase
```



- Sequence executes when its `start()` method is called

4-9

start() Method in Sequence Class

- The start() method calls the body() method in turn in the order shown below

```
start() {  
    pre_start() (task) // UVM-1.1  
    pre_body() (task) // UVM-1.0  
    body() (task) // Your stimulus generation code  
    post_body() (task) // UVM-1.0  
    post_start() (task) // UVM-1.1  
}
```

- The purpose of the pre/post hook/callback methods is explained in the upcoming slides

4-10

Caution:

By default, **pre_body()** and **post_body()** are only called for the sequence directly being executed by the sequencer. If a sequence is embedded in another sequence (you will see this in the nested sequence slide), the nested sequence is a child sequence. The nested sequence's **pre_body()** and **post_body()** will not be executed. This can lead to unexpected problems.

Recommendation:

Treat **pre_body()** and **post_body()** as deprecated.

Sequence Execution Methodologies

■ There are two ways to execute a sequence

- Explicitly
 - ◆ Test writer must create the sequence object then call the `start()` method

```
packet_sequence seq;
seq = packet_sequence::type_id::create("seq", this);
seq.start(env.agt.sqr);
```

- Implicitly

- ◆ Test writer populates the `uvm_config_db` with the intended sequence execution
 - Can be done in test code or via run-time switch
- ◆ The **sequencer** will pick it up from the `uvm_config_db`, create the sequence object and call the `start()` method automatically

```
uvm_config_db #(uvm_object_wrapper)::set(this, "* .sqr.main_phase",
"default_sequence", packet_sequence::get_type());
```

4-11

Explicit Sequence Execution

```
class test_base extends uvm_test;
  // Other code not shown
  virtual task main_phase(uvm_phase phase);
    packet_sequence seq;
    phase.raise_objection(this);
    seq = packet_sequence::type_id::create("seq", this);
    seq.randomize(); // optional
    seq.start(env.agt.sqr);
    phase.drop_objection(this);
  endtask
endclass
```

Raise and drop phase objection

Construct sequence object

Execute sequence

- Only do this in test!
 - Simple to implement but hard to control and reuse
- Must implement the phase method
- Must raise and drop phase objection in phase method
- Must call sequence's start () method and specify a sequencer

4-12

The `raise_objection()` / `drop_objection()` calls are to prevent the Run Time task phase that the sequence is being executed in from terminating before the sequence completes. Without these calls, when the sequence is executed in a Run Time task phase, that phase will execute but can terminate in 0 time. When the phase terminates, all child thread of that phase will be killed including all pending sequence executions.

Objection must be raised for that phase upon entry and dropped upon exit.

Objections are like students who raise their hand during lecture. Instructor cannot move on to next slide (phase) until all questions are answered (objections dropped).

There is an optional second argument to the objection method:

```
phase.raise_objection(this, $sformatf("Starting %s", seq.get_type_name()));
```

When simulated with `+UVM_OBJECTION_TRACE` run-time switch will display:

```
UVM_INFO @ 0.0s: main [OBJTN_TRC] Object uvm_test_top raised 1 objection(s) (Starting packet_sequence): count=1 total=1
```

Note on `+UVM_OBJECTION_TRACE`: this switch is implemented in Accellera UVM source code, but not documented in the IEEE UVM LRM.

Implicit Sequence Execution (1/2)

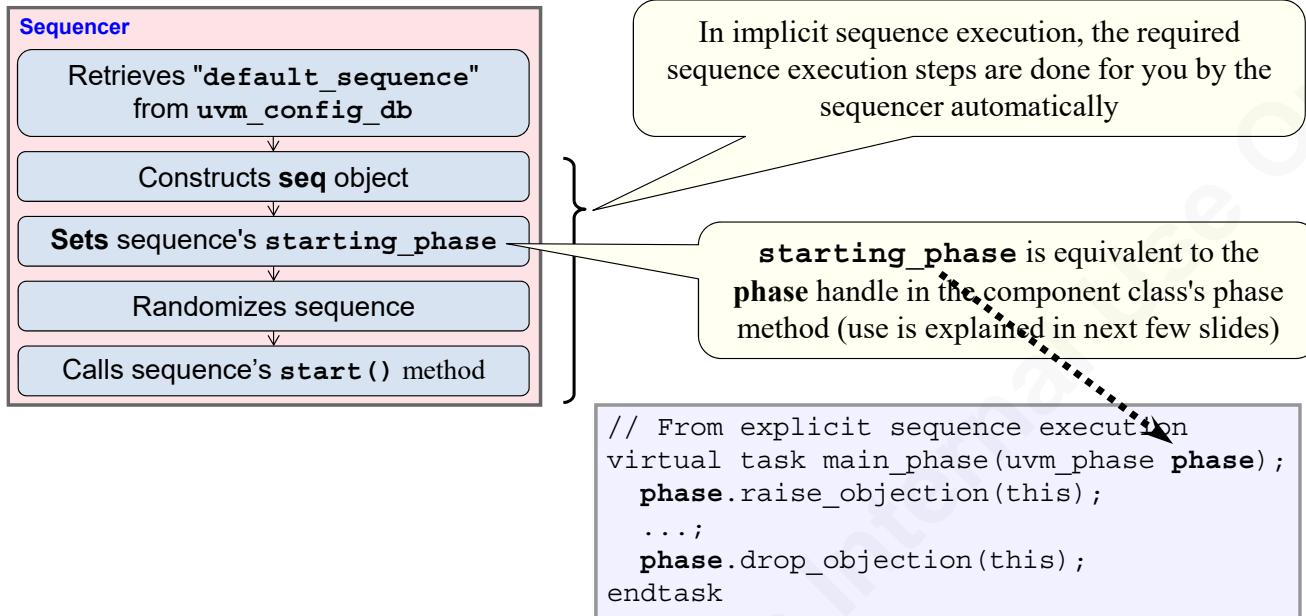
```
class test_base extends uvm_test; // Other code not shown
    virtual function void build_phase(uvm_phase phase); ...;
        uvm_config_db #(uvm_object_wrapper)::set(this, ".sqr.main_phase",
            "default_sequence", packet_sequence::get_type());
    endfunction
endclass
```

Populate sequence in `uvm_config_db`

- **Populate `uvm_config_db` with sequence to be executed by the chosen sequencer in phase specified**
 - Set in `build_phase`
 - Can be done in environment class or test class
 - Can be overridden in higher layer structure, derived class or run-time switch
- **Recommended Methodology**
 - Requires more code in sequence class (coming up)

Implicit Sequence Execution (2/2)

- Automatically happens in the configured phase:



starting_phase is the phase in which the sequencer executes the configured sequence. It is a handle to a **uvm_phase** object.

Example of the set in test:

```
uvm_config_db #(uvm_object_wrapper)::set(this, "*.sqr.main_phase",
    "default_sequence", packet_sequence::get_type());
```

Implicit Sequence Execution and Objection

- In implicit sequence execution, no phase objections are raised or dropped in test

Cannot raise or drop phase objection for sequence execution in **build_phase**

```
class test_base extends uvm_test; // Other code not shown
    virtual function void build_phase(uvm_phase phase); ...;
        uvm_config_db #(uvm_object_wrapper)::set(this, "*.sqr.main_phase",
            "default_sequence", packet_sequence::get_type());
    endfunction
endclass
```

- The sequence must manage objection!

- Implementations in UVM-1.1 and UVM-1.2 are different!
- Shown in the next few slides



4-15

This is sometimes called phase-aware sequence.

UVM-1.1 Sequence Phase Objection

■ Where to raise and drop phase objection in sequence?

- A `uvm_phase (starting_phase)` is built-in to the `uvm_sequence` base class
- Use this handle to raise phase objection in `pre_start()`
- Then, drop phase objection in `post_start()`

```
class packet_sequence extends uvm_sequence #(packet);
    virtual task pre_start();
        if ((get_parent_sequence() == null) && (starting_phase != null))
            starting_phase.raise_objection(this, "Starting");
    endtask
    virtual task post_start();
        if ((get_parent_sequence() == null) && (starting_phase != null))
            starting_phase.drop_objection(this, "Ending");
    endtask
    virtual task body(); // Same as before
endclass
```

Optional debug message

UVF-1.1 ONLY!
Will not compile in
UVF-1.2

4-16

The reason that one needs to guard for `starting_phase` from being `null` is because sequences can be executed by the test writer either explicitly or implicitly.

In the implicit execution case, the sequencer will assign the `starting_phase` handle to the correct phase object for you.

In the explicit execution case, this assignment does not happen. If you do not guard against the `starting_phase` from being `null`, then your simulation will terminate at the `starting_phase.raise_objection()` line with an error message saying that `starting_phase` is a null object.

You should write your sequences so that they can be executed either explicitly or implicitly.

In addition, you should also make sure that excessive raising and dropping of objection does not happen for nested sequences. The way to take care of this is to check whether or not the sequence has a parent. Only raise and drop objections if there are no parent.

UVM-1.2 & IEEE UVM Sequence Phase Objection

■ Where to raise and drop phase objection in sequence?

- `starting_phase` handle access has been deprecated

```
task packet_sequence::pre_start();
    if ((get_parent_sequence() == null) && (starting_phase != null))
        starting_phase.raise_objection(this, "Starting");
endtask
```

Compile error in UVM-1.2

- Replaced by `get_starting_phase()` method
 - ◆ But, you still need to implement `pre_start()` and `post_start()` methods
- A better way is to automate this with a new UVM-1.2 method called `set_automatic_phase_objection()`
 - ◆ Argument of "1" turns it on, "0" turns it off

```
function packet_sequence::new(string name = "packet_sequence");
    super.new(name);
    set_automatic_phase_objection(1); // UVM-1.2 & IEEE UVM Only!
endfunction
```

4-17

Code That Can Work in UVM-1.1 and UVM-1.2

■ Use UVM VERSION to be compile-able under either version

```
function packet_sequence::new(string name = "packet_sequence");
    super.new(name);
    `ifndef UVM_VERSION_1_1
        set_automatic_phase_objection(1);
    `endif
endfunction

`ifdef UVM_VERSION_1_1
task packet_sequence::pre_start();
    if ((get_parent_sequence() == null) && (starting_phase != null))
        starting_phase.raise_objection(this);
endtask

task packet_sequence::post_start();
    if ((get_parent_sequence() == null) && (starting_phase != null))
        starting_phase.drop_objection(this);
endtask
`endif
```

uvm_pkg provides compile-time switches to let you figure out which version of UVM you are compiling for

4-18

Or, you can use the migration scripts provided by the UVM committee to do the conversion. You can find this script in the /bin/uvm11-to-uvm12.pl of the Accellera UVM-1.2 source code directories.

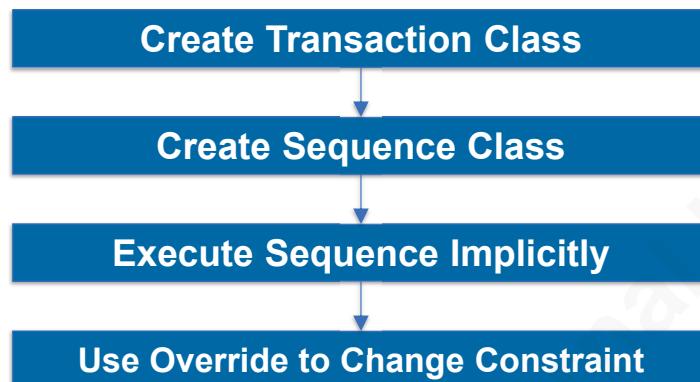
There are no conversion scripts for migration to IEEE UVM.

Lab 2: Generate Stimulus



30 minutes

Implement transaction and sequence classes



4- 19

Sequence with randc Transaction Property

- `uvm_do macro does not support randc behavior

```
class transaction extends uvm_sequence_item;
  randc bit[3:0] value;
endclass

class my_sequence extends uvm_sequence#(transaction);
  virtual task body();
    repeat(10) begin
      `uvm_do(req); // Does not work for randc!
    end
  endtask
endclass
```

Does not work for **randc**!
`uvm_do always creates a new object
whereas **randc** variables require
randomization on the same object

- Implement discrete code :

```
virtual task body(); transaction rand_obj;
  `uvm_create(rand_obj);
  repeat(10) begin
    `uvm_create(req);
    start_item(req);
    rand_obj.randomize(); // Randomize same object
    req.copy(rand_obj);
    finish_item(req); // Pass a copy of the
  end                         object to driver
endtask
```

4-20

If you like macro, you can create your own macro:

(Only for sequence_item classes. Additional code needed for sequence classes)

```
`define do_randc(rand_obj, req) \
begin \
  `uvm_create(req); \
  start_item(req); \
  if(!rand_obj.randomize()) \
    `uvm_fatal("RAND_ERR", {"\n", rand_obj.sprint()}); \
  req.copy(rand_obj); \
  finish_item(req); \
end
```

Then, the sequence code becomes:

```
virtual task body();
  transaction rand_obj;
  `uvm_create(rand_obj);
  repeat(10) begin
    `do_randc(rand_obj, req);
  end
endtask
```

Creating a Sequence of Related Items

```
class packet_scenario extends sequence_base;
    // macro and constructor not shown
    rand packet handles[];
    int item_count = 10;
    constraint array_constraint {
        foreach(handles[i]) {
            (i > 0) -> handles[i].sa == handles[i-1].sa + 1;
        }
    }
    function void pre_randomize();
        super.pre_randomize();
        handles = new[item_count];
        foreach(handles[i]) begin
            `uvm_create(handles[i]); // create transaction objects
        end
    endfunction
    virtual task body();
        foreach (handles[i]) begin
            `uvm_send(handles[i]); // only executes start_item() and finish_item()
        end
    endtask
endclass
```

Use array of rand sequence items to simplify defining relationship via constraint



4-21

This example creates a set of packets with monotonically increasing *sa* values.

Not shown is the fact that the parent sequencer randomizes the sequence before it starts executing it. When that happens, the *packet* array *arr* is randomized using the constraint that the *sa* value must be larger than the one before it.

The **body()** method contains steps similar to **`uvm_do**, but skips the randomize step. Instead, **body()** takes the already populated array of random packets and pass them to the driver.

Nested Sequences

■ Sequences can be nested

```
class noise_sequence extends sequence_base; // other code
    virtual task body();
        class burst_sequence extends sequence_base; // other code
            virtual task body();
                class congested_sequence extends sequence_base; // other code
                    virtual task body();
                        class nested_sequence extends sequence_base;
                            // utils macro and constructor not shown
                            noise_sequence      noise;
                            burst_sequence      burst;
                            congested_sequence congestion;
                            virtual task body();
                                `uvm_do(noise);
                                `uvm_do(burst);
                                `uvm_do(congestion);
                            endtask
                        endclass
                    endclass
                endclass
            endclass
        endclass
    endclass
endclass
```

4-22

`uvm_do macro also applies to sequences

A sequence can have subsequences. In this case, the **nested_sequence** contains three other sequences that are executed sequentially. If you want them to run concurrently, put them inside a fork-join.

In UVM, the implementation of **`uvm_do** macro for sequence calls the **start()** method with an additional flag. The following is the simplified code of **`uvm_do** for sequence:

```
// *Equivalent code not actual code
`define uvm_do(UVM_SEQ) \
    `uvm_create(UVM_SEQ, ) \
    UVM_SEQ.randomize(); \
    UVM_SEQ.start(get_sequencer(), this, .call_pre_post(0));
```

when set to 0, pre_ and post_body will not execute

The **pre/post body** of the subsequences do not execute. If you want the **pre/post body** of the subsequence to execute, then you need to make the following explicit calls:

```
virtual task body();
    `uvm_create(noise);
    noise.randomize();
    noise.start(get_sequencer(), this); // pre_and_post_body executes
    `uvm_do(burst); // no pre/post_body execution
    `uvm_do(congestion); // no pre/post_body execution
endtask
```

call_pre_post defaults to 1

Because of this problem, the recommendation is to avoid using **pre_** and **post_body** methods.

Implicit Sequence Execution Overrides

- Default test (`test_base`) executes default sequence

```
class test_base extends uvm_test; // other code not shown
  virtual function void build_phase(...); super.build_phase(...);
    uvm_config_db#(uvm_object_wrapper)::set(this, "env.agt.sqr.main_phase",
      "default_sequence", packet_sequence::get_type());
  // other code not shown
endfunction
endclass
```

sequence set by base class

- Derived test can be override or turn it off

```
class test_nested extends test_base; // other code not shown
  virtual function void build_phase(...); super.build_phase(...);
    uvm_config_db#(uvm_object_wrapper)::set(this, "env.agt.sqr.main_phase",
      "default_sequence", <usr_seq>); // tells sqr to run usr_seq in main phase
  endfunction
endclass
```

Setting `<usr_seq>` to `null` turns off implicit sequence execution
Otherwise, overrides with specified `<usr_seq>`

4-23

Implicit Sequence Execution at Phases

■ Sequences be targeted for a chosen phase

- Typically done at the testcase level

```
class simple_seq_RST extends sequence_base;
class simple_seq_CFG extends sequence_base;
class simple_seq_MAIN extends sequence_base;

class test_multi_phase extends test_base;
  typedef uvm_config_db #(uvm_object_wrapper) seq_phase;
  // utils macro and constructor not shown
  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    seq_phase::set(this, "env.agt.sqr.reset_phase",
                  "default_sequence", simple_seq_RST::get_type());
    seq_phase::set(this, "env.agt.sqr.configure_phase",
                  "default_sequence", simple_seq_CFG::get_type());
    seq_phase::set(this, "env.agt.sqr.main_phase",
                  "default_sequence", simple_seq_MAIN::get_type());
  endfunction
endclass
```

4-24

For reusability, implicit sequence execution is the preferred implementation. The reason is that at higher level testbench code, one can chose to override what's configured at the lower layer.

However, if there are synchronization needing to be done among different sequences in different sequencers in the same phase, the synchronization code could get very complicated. For synchronization purposes, one may want to consider executing sequences explicitly at the test level where the order of execution is up to the implementer of the test.

Unit Objectives Review

You should now be able to:

- Build stimulus generator class by inheriting from `uvm_sequence`
- Execute sequence explicitly and implicitly



4- 25

Appendix

Sequence Priority/Weight

Sequencer-Driver Response Port

Out-Of-Order Sequencer-Driver Port

4- 26

Sequence Priority/Weight

■ Sequences can be assigned priority/weight

```
class nested_sequence extends sequence_base;
  // utils macro and other code not shown
  virtual task body();
    uvm_sequencer_base sqr = get_sequencer();
    sqr.set_arbitration(UVM_SEQ_ARB_STRICT_FIFO)
    fork
      `uvm_do_pri(noise, 1000);
      `uvm_do_pri(burst, 50);
      `uvm_do(congestion);
    join
  endtask
endclass
```

Must set arbitration
(Defaults to UVM_SEQ_ARB_FIFO)

! See note for IEEE definition

Defaults to priority/weight of 100
Higher value has higher priority

UVM_SEQ_ARB_FIFO	Requests are granted in FIFO order (default)
UVM_SEQ_ARB_WEIGHTED	Requests are granted randomly by weight
UVM_SEQ_ARB_RANDOM	Requests are granted randomly
UVM_SEQ_ARB_STRICT_FIFO	Requests at highest priority granted in fifo order
UVM_SEQ_ARB_STRICT_RANDOM	Requests at highest priority granted randomly
UVM_SEQ_ARB_USER	Arbitration is delegated to the user-defined function, <code>user_priority_arbitration()</code>

4-27

In IEEE UVM, the priority value is embedded in `uvm_do macro:

`uvm_do(SEQ_OR_ITEM, SEQR=get_sequencer(), **PRIORITY=-1**, CONSTRAINTS={})

Caution for implementing user arbitration:



Existing Accellera implementation:

```
virtual function integer uvm_sequencer_base::user_priority_arbitration(integer avail_sequences[$]);
```

IEEE UVM spec:

```
virtual function int uvm_sequencer_base::user_priority_arbitration(integer avail_sequences[$]);
```

Once Accellera implements the IEEE definition, you may need to go back and update your code.

When the sequencer arbitration mode is set to UVM_SEQ_ARB_USER (via the set_arbitration method), the sequencer will call this function each time that it needs to arbitrate among sequences.

Derived sequencers may override this method to perform a custom arbitration policy. The override must return one of the entries from the avail_sequences queue, which are indexes into an internal queue, arb_sequence_q.

The default implementation behaves like UVM_SEQ_ARB_FIFO, which returns the entry at avail_sequences[0].

Appendix

Sequence Priority/Weight

Sequencer-Driver Response Port

Out-Of-Order Sequencer-Driver Port

4- 28

Sequencer-Driver Response Port

- Driver can send response back to sequence

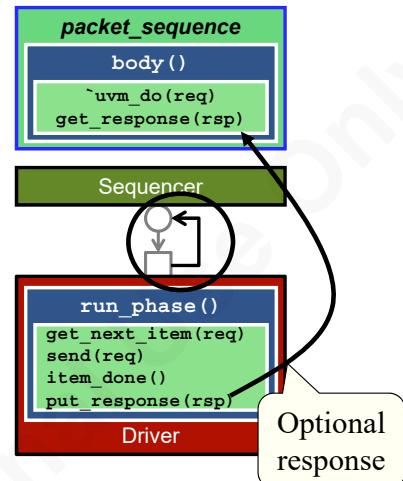
```
task body();
  repeat (item_count) begin
    `uvm_do(req);
    get_response(rsp);
    // process response
  end

class driver extends uvm_driver #(packet);
  task run_phase(uvm_phase phase);
    forever begin
      seq_item_port.get_next_item(req);
      send(req);
      rsp = packet::type_id::create("rsp", this);
      rsp.set_id_info(req);
      // set rsp response
      seq_item_port.item_done();
      seq_item_port.put_response(rsp);
    end
  endtask
endclass
```

Retrieve response

Copy response id
(required!)

Set response



4-29

Note that the driver picks up req but sends back rsp.

Appendix

Sequence Priority/Weight

Sequencer-Driver Response Port

Out-Of-Order Sequencer-Driver Port

4-30

Sequence with Out-Of-Order Response (1/2)

```
class packet_sequence extends packet_sequence_base;
    packet pkt_req[int];
    packet rand_obj = packet::type_id::create("rand_obj");
    task body();
        repeat(item_count) begin
            wait(pkt_req.size() < 5); <-- Throttle traffic
            `uvm_create(req);
            start_item(req);
            if (!rand_obj.randomize()) ...;
            req.copy(rand_obj);
            finish_item(req);
            pkt_req[req.get_transaction_id()] = req;
            fork
                packet in_driver = req;
                process_rsp(in_driver); <-- Wait for specific
                join_none // see next slide transaction response
            end
            wait(pkt_req.size() == 0); <-- Block until array is empty
        endtask
    endclass
```

4-31

Sequence with Out-Of-Order Response (2/2)

```
virtual task process_rsp(packet in_driver);
    packet from_driver;
    get_response(from_driver, in_driver.get_transaction_id());
    // process response
    pkt_req.delete(from_driver.get_transaction_id());
endtask
endclass
```

Retrieve response based on transaction_id

Remove response from in-operation array

4-32

Out-Of-Order Driver (1/2)

- Need a queue to store transactions
- Implement threads to get and execute transactions
- User need to disable auto recording and manually record the transactions

```
class driver extends uvm_driver#(packet) ; // other code not shown
  packet pkt_q[$] ;
  virtual task run_phase(uvm_phase phase) ;
    `ifndef UVM_VERSION_1_1
      seq_item_port.disable_auto_item_recording() ; // UVM-1.2 & IEEE
    `endif
    // All versions can use +UVM_DISABLE_AUTO_ITEM_RECORDING option
    fork
      get_item();
      execute_item();
    join
  endtask
  // see next slide for method implementation
endclass
```

Disable auto recording

Call thread to get transaction

Call thread to execute transaction

4-33

Out-Of-Order Driver (2/2)

```
virtual task get_item()
  forever begin
    seq_item_port.get_next_item(req);           Get transaction
    accept_tr(req);                          // transaction accepted
    pkt_q.push_back(req);
    seq_item_port.item_done();                Store transaction in queue
  end
endtask

virtual task execute_item();
  forever begin
    int index;
    wait (pkt_q.size() != 0);                Wait for transaction to process
    index = $urandom_range(pkt_q.size()-1);
    begin_tr(pkt_q[index]);                  // transaction recording starts
    // process transaction code left off
    seq_item_port.put_response(pkt_q[index]); // indicate transaction completed
    end_tr(pkt_q[index]);                   // transaction recording ends
    pkt_q.delete(index);                   Indicate which transaction has
  end                                         completed processing
endtask
```

4-34

Agenda

DAY
2

5 UVM Configuration & Factory

6 UVM Component Communication



7 UVM Scoreboard & Coverage

8 UVM Callback



Synopsys 40-I-055-SSG-007

© 2019 Synopsys, Inc. All Rights Reserved

5-1

Unit Objectives



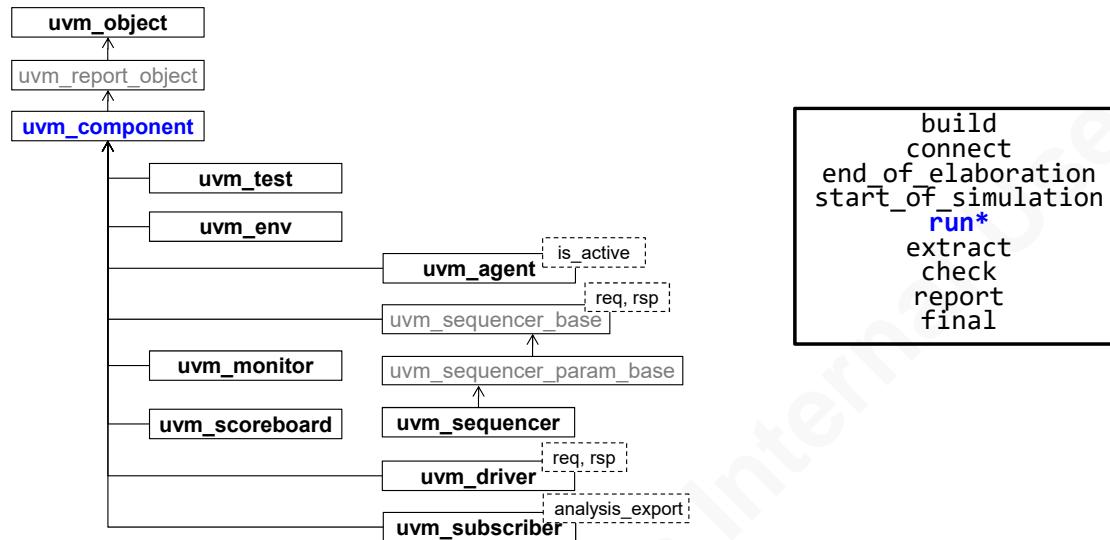
After completing this unit, you should be able to:

- **Describe component logical hierarchy**
- **Use logical hierarchy to get/set component configuration fields**
- **Use factory to create test replaceable transaction and components**

UVM Component Base Class Structure

■ Behavioral base class is `uvm_component`

- Has logical parent-child relationship
 - ◆ Used for phasing control, configuration and factory override



5-3

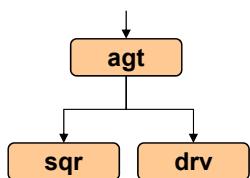
Component Parent-Child Relationships

- Logical relationship is established at creation of component object

```
class driver extends uvm_driver #(packet); // utils macro
    function new(string name, uvm_component parent);
        super.new(name, parent);
    endfunction
    ...
endclass

class agent extends uvm_agent; // utils macro
    typedef uvm_sequencer #(packet) sequencer;
    sequencer sqr; driver drv;

    function new(string name, uvm_component parent); ...
    function void build_phase(...); super.build_phase(...);
        sqr = sequencer::type_id::create("sqr", this);
        drv = driver::type_id::create("drv", this);
    endfunction
endclass
...
agent agt = agent::type_id::create("agt", this);
```



Pass parent in via constructor
(components only)

Establish logical hierarchy

5-4

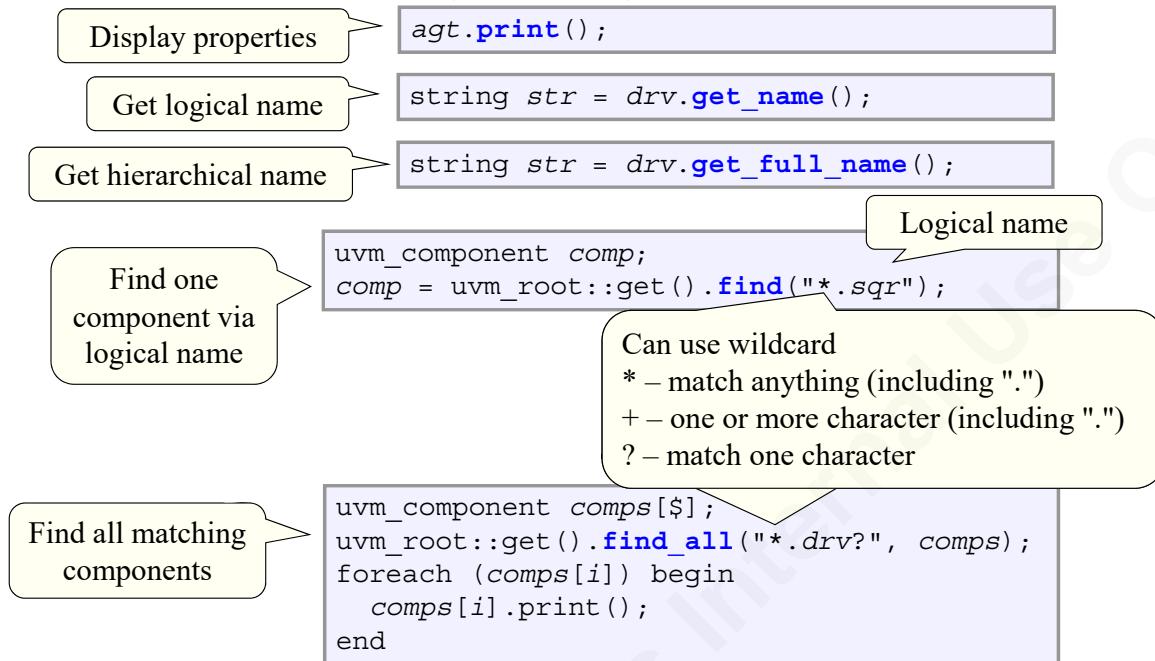
The most common reason to create a logical hierarchy tree is so the test can find and change the functionality of a component. For example, a test can call `uvm_config_db#()` and use a relative hierarchical path to specify the component path. The hierarchy tree can also be used to change factory patterns.

Component parent-child relationship also establishes the function phase execution order.

If the parent argument is not set (blank), the parent argument will default to null and cause the component to be at the top of the UVM component structure. This happens when `uvm_root` creates the test objection from the `+UVM_TESTNAME` runtime switch. User code should not set null as the parent handle. User code should always populate the parent argument with "this".

Display & Querying

■ Rich set of methods for query & display



5-5

Here are a few simple examples of regular expression :

<code>top\..*</code>	all of the scopes whose top-level component is top
<code>top\env\..*\monitor</code>	all of the scopes in env that end in monitor; i.e. all the monitors two levels down from env
<code>.*\monitor</code>	all of the scopes that end in monitor i.e. all the monitors (assuming all monitors are named "monitor")
<code>top\u[1-5]\.*</code>	all of the scopes named u1, u2, u3, u4 or u5 and their subscopes.

A simpler version for regular expressions is globs. A glob only has three metacharacters: *, +, and ?. Range is not supported. The following table shows glob metacharacters.

char	meaning	regular expression equivalent
*	0 or more characters	.*
+	1 or more characters	.+
?	exactly one character	.

Three of above examples can be written as globs. Last one cannot (range is not supported in glob).

regular expression	glob equivalent
<code>top\..*</code>	<code>top.*</code>
<code>top\env\..*\monitor</code>	<code>top.env.*.monitor</code>
<code>.*\monitor</code>	<code>*.monitor</code>

The resource database supports both regular expression and glob syntax. Regular expressions are identified as such when they surrounded by '\' characters. All others are treated as glob expressions.

Query Hierarchy Relationship

■ Easy to get handle to object parent/children

Get handle to parent

```
uvm_component comp;  
comp = this.get_parent();
```

Finding object via logical name

```
uvm_component comp;  
comp = vip.get_child("sqr");
```

Logical name

Determine number of children

```
int num_ch = vip.get_num_children();
```

Iterate through children

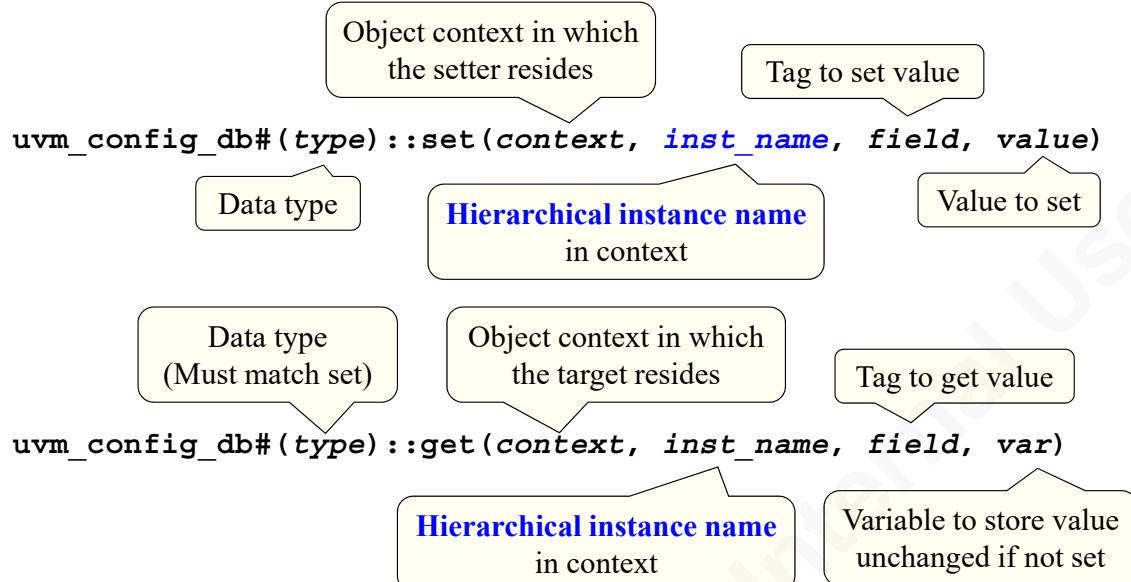
```
string name;  
uvm_component child;  
if (vip.get_first_child(name)) do begin  
    child = vip.get_child(name);  
    child.print();  
end while (vip.get_next_child(name));
```

5-6

UVM maintains a tree showing the hierarchy between parents and children. Remember, this tree is the hierarchy of the testbench, and not an OOP inheritance relationship. The methods on this slide allow your testbench to become self-aware and explore the components' connections.

Use Logical Hierarchy in Configuration

■ Mechanism for configuring object properties



5-7

`uvm_config_db#():get()` method returns a 1 if there is a matching set. If matched, the var field will be updated with the set value. If no match, the return value is 0 and the var field is unmodified.

`uvm_config_db#():set()` method returns a void.

Component Configuration Example

■ Agent component field configuration

- Environment should target agent
 - ◆ Agent then configures its child components

```
class router_env extends uvm_env;
    // utils macro and constructor not shown
    virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        uvm_config_db #(int)::set(this, "i_agt[10]", "port_id", 10);
    endclass
    class agent extends uvm_agent;
        // constructor not shown
        int port_id = -1; // default value
        `uvm_component_utils_begin(agent)
            `uvm_field_int(port_id, UVM_ALL_ON)
        `uvm_component_utils_end
        virtual function void build_phase(...); ...
            uvm_config_db #(int)::get(this, "", "port_id", port_id);
            uvm_config_db #(int)::set(this, "*", "port_id", port_id);
        endfunction
    endclass
```

Configure agents in
build_phase

Set configuration
targeting agent



Please see note

Retrieve configuration in agent

Then, set children component configuration

5-8

When `uvm_field_*` is used for processing members of the component class, an automatic retrieval of the field is executed in the uvm_component base class when super.build_phase(phase) is executed.

So, in the code above, the retrieval of the port_id from configuration database will be redundantly executed twice. This can lead to a hit in simulation performance.

For uvm-1.1 and uvm-1.2, the solution is not to retrieve the field from the configuration database explicitly (meaning do not call `uvm_config_db#(...)::get(...)`). Instead, let `super.build_phase(phase)` do the work for you.

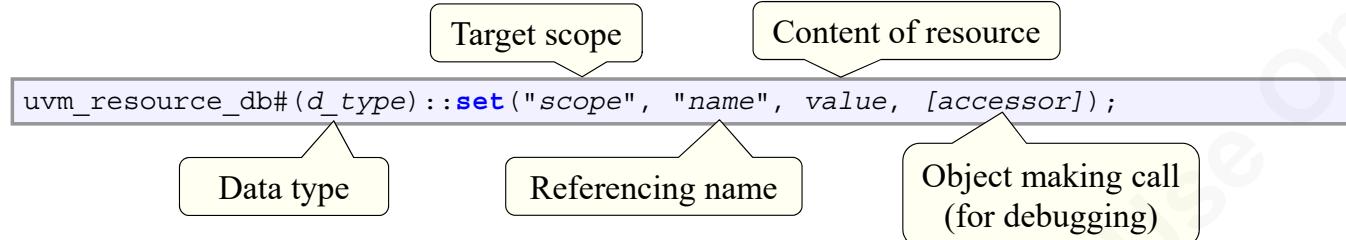
When you do this, you may still get a performance hit. The reason is that `super.build_phase(phase)` will attempt to retrieve all fields listed with the `uvm_field macro whether you need (want) it or not.

In IEEE UVM, this is solved with a new flag call `UVM_NOSET` in the macro. (Not yet implemented) example:

```
`uvm_component_utils_begin(agent)
    `uvm_field_int(port_id, UVM_ALL_ON | UVM_NOSET)
`uvm_component_utils_end
```

UVM Resource

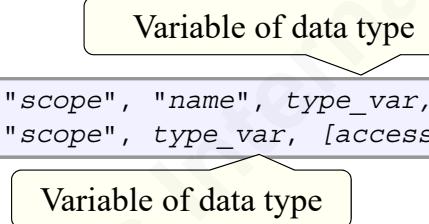
- **uvm_config_db targets instances of objects**
- **For global configuration, use uvm_resource_db**



- **Retrieval of the resources can be done in two ways**

- Read by name
- Read by type

```
uvm_resource_db#(d_type)::read_by_name ("scope", "name", type_var, [accessor]);
uvm_resource_db#(d_type)::read_by_type ("scope", type_var, [accessor]);
```



5-9

Manage DUT Interface Configuration

- In program/module, use `uvm_resource_db::set()`

```
program automatic test;
import uvm_pkg::*;

initial begin
    uvm_resource_db#(virtual router_io)::set("router_vif", "",
                                              router_test_top.router_if);
    uvm_resource_db#(virtual reset_io)::set("reset_vif", "",
                                              router_test_top.reset_if);
    $timeformat(-9, 1, "ns", 10);
    run_test();
end
endprogram
```

In program, access interface via XMR
In module, access interface directly

`uvm_resource_db` is used instead of `uvm_config_db` to insulate program/module code from test hierarchy changes.

5-10

Test Configures Agents with Interfaces

- In test, use `uvm_resource_db::read_by_type()` to retrieve interface
- Use `uvm_config_db::set()` to configure targeted agents

```
class test_base extends uvm_test; // other code left off
    virtual router_io router_vif;
    virtual reset_io  reset_vif;
    virtual function void build_phase(uvm_phase phase); // other code left off
        uvm_resource_db#(virtual reset_io)::read_by_type("reset_vif",
                                                        reset_vif, this);
        uvm_config_db#(virtual reset_io)::set(this, "env.r_agt",
                                              "vif", reset_vif);
        uvm_resource_db#(virtual router_io)::read_by_type("router_vif",
                                                        router_vif, this);
        uvm_config_db#(virtual router_io)::set(this, "env.i_agt[*]",
                                              "vif", router_vif);
    endfunction
endclass
```

Configure agent, NOT children of agent!

5-11

Configuring Component's Interface (1/2)

■ In driver/monitor

- Call `uvm_config_db#()::get()` in `build_phase`
- Check for correctness in `end_of_elaboration_phase`

```
class driver extends uvm_driver#(packet); // other code not shown
    virtual router_io vif;

    virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        uvm_config_db#(virtual router_io)::get(this,"","vif",vif)
    endfunction

    virtual function void end_of_elaboration_phase(uvm_phase phase);
        super.end_of_elaboration_phase(phase);
        if (vif == null) begin
            `uvm_fatal("CFGERR", "Driver DUT interface not set");
        end
    endfunction
endclass
```

5-12

Configuring Component's Interface (2/2)

■ In agent

- In `build_phase`
 - ◆ Call `uvm_config_db#()::get()` to retrieve interface
 - ◆ Call `uvm_config_db#()::set()` to set interface for children of agent

```
class input_agent extends uvm_agent; // other code not shown
    virtual router_io vif;
    virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        uvm_config_db#(virtual router_io)::get(this, "", "vif", vif);
        uvm_config_db#(virtual router_io)::set(this, "*", "vif", vif);
    endfunction
endclass
```

5-13

It is not absolutely necessary to check for correctness of the configuration at the agent level if the agent is not using the configuration.

An alternative that you may see some people do in the agent class is to bypass the `uvm_config_db` when setting up the configuration of the children components. Instead, they would assign the configuration directly as shown below. They do this for two reason: easier to debug and by-passing `uvm_config_db` improves performance. If you do this, make sure you check for the correctness of the configuration in the agent class!

```
class input_agent extends uvm_agent; // other code not shown
    driver drv; packet_sequencer sqr; monitor mon;
    virtual router_io vif;
    virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        // construction of children components not shown
        uvm_config_db#(virtual router_io)::get(this, "", "vif", vif);
//      uvm_config_db#(virtual router_io)::set(this, "*", "vif", vif);
        drv.vif = vif;
        mon.vif = vif;
    endfunction
endclass
```

Additional Needs: Manage Test Variations

- Tests need to introduce class variations, e.g.
 - Adding constraints
 - Modify the way data is sent by the driver
- Variation can be instance based or global
- Control object construction for all or specific instances of a class
- Create generic functionality
 - Deferring exact object creation to runtime

Solution : Built-in UVM Factory

5- 14

Test Requirements: Transaction

- How to manufacture transaction instances with additional information without modifying the original source file?

```
class monitor extends uvm_monitor;  
  ...;  
  virtual task run_phase(uvm_phase phase);  
    forever begin  
      packet pkt;  
      pkt = new("pkt");  
      get_packet(pkt);  
    end  
  endtask  
endclass
```

Type of object determines memory allocated for the instance

Poor coding style
No way of overriding the transaction object with a derived type

Impossible to add new members or modify constraint later

```
class bad_packet extends packet; ...;  
  bit bad;  
  virtual function int compute_crc();  
endclass
```

5-15

For example, to add a serial number (as shown) or modify the constraints, or inject errors, or create a stream of objects that have additional fields used to help in the scoreboarding. If these objects are created by a complex monitor or Verification IP, you do not want (or simply cannot) to modify the source code, especially to add functionality required by a single test.

Test Requirements: Components

- How to manufacture component instances with additional information without modifying the original source file?

```
class input_agent extends uvm_agent;
    packet_sequencer sqr;
    driver drv;
    ...
    virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        sqr = new("sqr", this);
        drv = new("drv", this);
        ...
    endfunction
endclass
```

No way of modifying the component behavior

```
class NewDriver extends driver;
    virtual task run_phase(uvm_phase phase);
        if (dut.vif.done == 1)
            ...
    endtask
endclass
```

Impossible to change behavior for test

5-16

Factories in UVM

Implementation flow

■ Factory instrumentation/registration

- `^uvm_object_utils(Type)`
- `^uvm_component_utils(Type)`

Macro creates a proxy class (called `type_id`) to represent the object/component **and** registers an instance of the proxy class in `uvm_factory`

■ Construct object using static proxy class method

- `ClassName obj = ClassName::type_id::create(...);`

Use proxy class to create object

■ Class overrides

- `set_type_override_by_type(...);`
- `set_inst_override_by_type(...);`

Created objects can now be overridden with a new implementation

Transaction Factory

■ Construct transaction object via `create()` method

Required! Macro defines a proxy class called `type_id`
An instance of proxy class is registered in `uvm_factory`

```
class packet extends uvm_sequence
  rand bit[3:0] sa, da;
  `uvm_object_utils_begin(packet)
    `uvm_field_int(sa, UVM_ALL_ON)
  ...
endclass

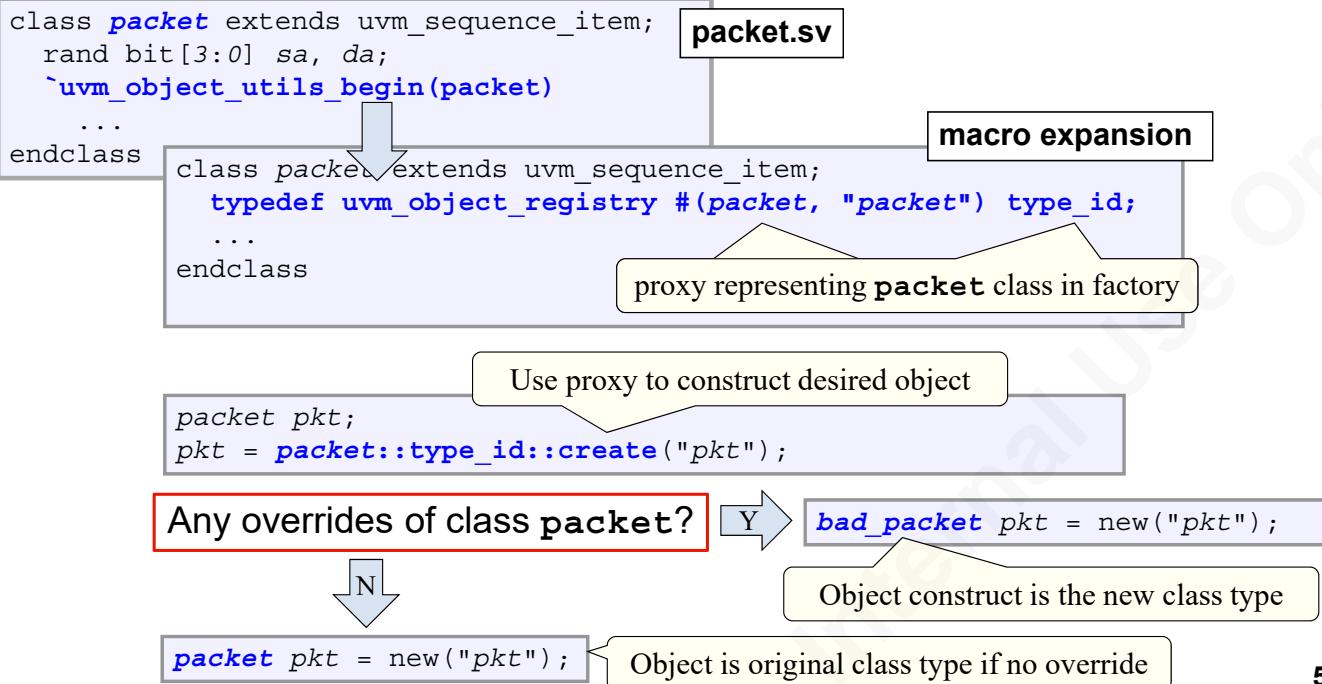
class monitor extends uvm_monitor;
  task run_phase(uvm_phase phase);
    forever begin
      packet pkt;
      pkt = packet::type_id::create("pkt");
      ...
    end
  endtask
endclass
```

Use proxy's `create()` method
to construct transaction object

5-18

The macro `uvm_object_utils` creates a parameterized `uvm_object_registry` class typed to `packet`. This class has the function `create()` that returns the parameter type, or `packet` in the example above.

UVM Factory Transaction Creation



How does the UVM factory creation system work?

When you declare a transaction, the macro `uvm_object_utils[_begin](T)` expands into:

```
typedef uvm_object_registry #(T, "T") type_id;
```

Defined within the `uvm_object_registry` class is a static `create` method:

```
static function T create (string name="", uvm_component parent=null, string context="");
```

The name string is the matching tag used in search when `set_*`() method is called. The second argument should be the parent component handle where the transaction resides. The parent handle then establishes the context in which the matching of the tag field takes place. Alternatively, the third argument can be used to establish the context for the search to match.

At execution, the `create()` method looks to see if there are any overrides of the class. If there is a matching override, construct an object of that override type, and return the handle. If there are no overrides, construct an object of the default type, ie. `packet`.

Overrides are created with `set_type_override_by_type()` or `set_inst_override_by_type()`, as shown in the next slide.

Component Factory

■ Construct object via `create()` using factory class

- Similar to transaction creation

```
class driver extends uvm_driver #(packet);  
  `uvm_component_utils(driver)  
  ...  
endclass
```

Required! Macro defines a proxy class called `type_id`
An instance of proxy class is registered in `uvm_factory`

```
class router_env extends uvm_env;  
  `uvm_component_utils(router_env)  
  driver drv;  
  ...  
  function void build_phase(uvm_phase phase);  
    super.build_phase(phase);  
    drv = driver::type_id::create("drv", this);  
  ...  
endfunction  
endclass
```

Parent component handle **required!**

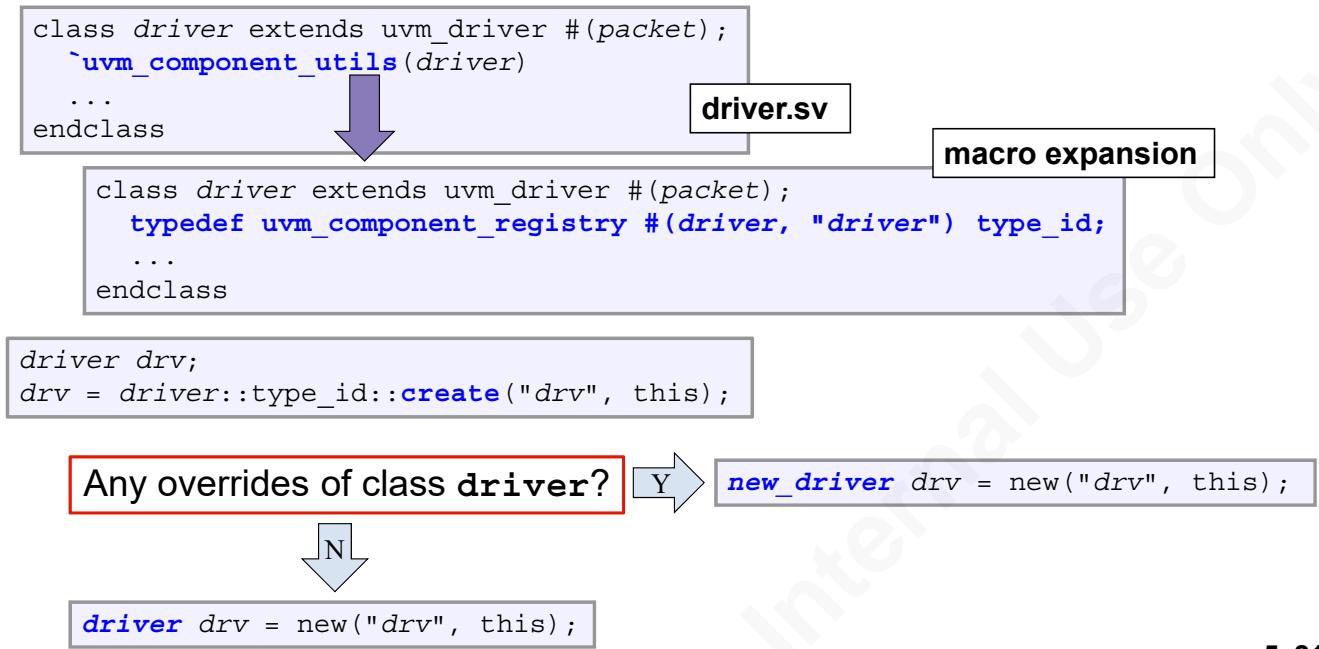
Use `create()` method of proxy
class to construct component object

5-20

The macro `uvm_component_utils` instantiates the class `uvm_component_registry`, with the parameter `driver`. This class has the function `create()` that returns the parameter type, or `driver` in the example above.

In `uvm_component_registry::create()`, the second argument, parent, is required and is usually `this`.

UVM Factory Component Creation



5-21

The component creation is almost identical to the transaction.

When you declare a component, the macro `uvm_component_utils(T)` expands into:

```
typedef uvm_component_registry #(T, "T") type_id;
```

Defined within the `uvm_component_registry` class is a static create method:

```
static function T create (string name="", uvm_component parent=null, string context="");
```

The name string is the matching tag used in search when `set_*`() method is called. The second argument should be the parent component handle where the transaction resides. The parent handle then establishes the context in which the matching of the tag field takes place. Alternatively, the third argument can be used to establish the context for the search to match.

At execution, the `create()` method looks to see if there are any overrides of the class. If there is a matching override, construct an object of that override type, and return the handle. If there are no override, construct an object of the default type, ie. `driver`.

Override in Test

■ User can chose type of override

- `set_inst_override_by_type()`
- `set_type_override_by_type()`

```
class test_override extends test_base;
  // utils and constructor not shown
  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    set_type_override_by_type(driver::get_type(), newDriver::get_type());
    set_inst_override_by_type("env.agt.sqr.seq",
      packet::get_type(), bad_packet::get_type());
  endfunction
endclass
```

Only packet instances in matching sequence are now `bad_packet`

5-22

Command-Line Override

■ User can override factory objects at command line

- Objects must be constructed with the factory `class_name::type_id::create(...)` method

```
+uvm_set_type_override=<req_type>,<override_type>  
+uvm_set_inst_override=<req_type>,<override_type>,<inst_path>
```

■ Works like the test class overrides

- `set_type_override()`
- `set_inst_override()`

■ Example:

```
+uvm_set_type_override=driver,newDriver  
+uvm_set_inst_override=packet,bad_packet,uvm_test_top.env.agt.sqr.seq
```

No space character!

5-23

Visually Inspect Factory Overrides (1/2)

- You should always check the overrides with
 - `uvm_factory::get().print()`
- Instance overrides are under Instance Overrides
- Global type overrides are under Type Overrides

Instance Overrides:

Requested Type	Override Path	Override Type
packet	uvm_test_top.env.agt.sqr.seq	bad_packet

Type Overrides:

Requested Type	Override Type
driver	newDriver

```
function void start_of_simulation_phase(uvm_phase phase);  
    uvm_factory::get().print();  
endfunction
```

5-24

Visually Inspect Factory Overrides (2/2)

- You should also check the structural overrides with

- `uvm_root::get().print_topology(...)`

```
UVM_INFO @ 64250.0ns: reporter [UVMTOP] UVM testbench topology:  
uvm_test_top: (test_base@510) {  
    env: (router_env@517) {  
        i_agt: (input_agent@529) {  
            drv: (newDriver@666) {  
                rsp_port: (uvm_analysis_port@681) {  
                    recording_detail: UVM_FULL  
                }  
                sqr_pull_port: (uvm_seq_item_pull_port@673) {  
                    recording_detail: UVM_FULL  
                }  
            }...  
        }  
    }  
}
```

Make sure new class type is applied

```
function void start_of_simulation_phase(uvm_phase phase);  
    uvm_factory::get().print();  
    uvm_root::get().print_topology(uvm_default_tree_printer);  
endfunction
```

Can select printer policy

5-25

Parameterized Component Class

■ Parameterized component requires a different macro

- ◆ `uvm_component_param_utils(cname)

```
class my_driver #(width=8) extends uvm_driver #(packet);
  typedef my_driver#(width) this_type;
  `uvm_component_param_utils(this_type)
  const static string type_name = $sformatf("my_driver#(%0d)", width);
  virtual function string get_type_name();
    return type_name;
  endfunction // other code not shown
```

■ Creation of object requires parameter

```
class my_agent extends uvm_agent;
  typedef my_driver#() my_driver_t;
  my_driver_t drv;
  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    drv = my_driver_t::type_id::create("drv", this);
  endfunction
endclass
```

Use typedef to minimize typo

5-26

Unit Objectives Review

You should now be able to:

- Describe component logical hierarchy
- Use logical hierarchy to get/set component configuration fields
- Use factory to create test replaceable transaction and components



5-27

Appendix

Run-time Switch Configuration

Sequence Configuration

Array Configuration

Enum Configuration

Configuration Debugging

Disabling Auto Field Configuration

5-28

Run-time Configuration Switch

■ Issue with `uvm_config_db`

- Run-time switch `+uvm_set_config_int` only works if the `get` data type is specified as `uvm_bitstream_t`

```
uvm_config_db#(uvm_bitstream_t)::get(this, "", "port_id", port_id);  
uvm_config_db#(int)::get(this, "", "port_id", port_id); // Unsupported by +uvm_set_config_int switch!
```

- The way around this is to wrap your integral data type around the `uvm_bitstream_t` data type as follows:

```
class driver extends uvm_driver#(packet);  
    int port_id = -1;           // other code left off  
    virtual function void build_phase(uvm_phase phase)  
        super.build_phase(phase);  
        if (!uvm_config_db#(uvm_bitstream_t)::get(this, "", "port_id", port_id)) begin  
            uvm_config_db#(int)::get(this, "", "port_id", port_id);  
        end // other code left off  
    endfunction  
endclass
```

5-29

Caution with this strategy:

Because there are now two ways to configure the field, one must be very careful about how to use either approach.

The recommendation is to reserve the use of `uvm_config_db#(uvm_bitstream_t)` only for run-time switch configuration. Do not use `uvm_config_db#(uvm_bitstream_t)::set(...)` within test classes!

In test classes, only use the actual data type (`uvm_config_db#(int)::set(...)`) for setting the configuration field.

Retrieve int Configuration Field via User Macro

- Instead of manually typing:

```
if (!uvm_config_db#(uvm_bitstream_t)::get(this, "", "port_id", port_id)) begin  
    uvm_config_db#(int)::get(this, "", "port_id", port_id);  
end
```

- One can create a macro to do majority of the call:

```
`define config_db_int_get(field_hash, field_name=field_hash) \  
if (!uvm_config_db#(uvm_bitstream_t)::get(this, "", field_hash, field_name)) begin \  
    uvm_config_db#(int)::get(this, "", field_hash, field_name); \  
end
```

- The call would then be:

```
`config_db_int_get(port_id)
```

Appendix

- Run-time Switch Configuration
- Sequence Configuration**
- Array Configuration
- Enum Configuration
- Configuration Debugging
- Disabling Auto Field Configuration

5-31

Configuring Sequences (Instance-Based)

■ Set in test

```
class test_20_items extends test_base; // other code not shown
    virtual function void build_phase(...); super.build_phase(...);
        uvm_config_db#(int)::set(this, "env.agt.sqr.seq", "count", 20);
    endfunction
endclass
```

Full path to sequence object (instance name not class name)

■ Retrieve configuration field through `get_full_name()`

```
class packet_sequence extends sequence_base; // other code not shown
    int count = 10;
    virtual task pre_start();
        super.pre_start();
        if (!uvm_config_db#(uvm_bitstream_t)::get(null, this.get_full_name(), "count", count))
            uvm_config_db#(int)::get(null, this.get_full_name(), "count", count);
    endtask
endclass
```

Full path to sequence object

5-32

The following will not compile.

```
uvm_config_db#(int)::get(this, "", "item_count", item_count);
```

The reason is that the first argument of the `get()` method must be a `uvm_component` (or its derived) handle.

Configuring Sequences (Class-Based)

■ Set in test

```
class test_20_items extends test_base; // other code not shown
    virtual function void build_phase(...); super.build_phase(...);
        uvm_config_db#(int)::set(this, "env.agt.sqr.packet_sequence", "count", 20);
    endfunction
endclass
```

Full path to class type name

■ Reference configuration field through get_type_name()

```
class packet_sequence extends sequence_base; // other code not shown
    int count = 10;
    function new(string name = "packet_sequence");
        virtual task pre_start();
            super.pre_start();
            if (!uvm_config_db#(uvm_bitstream_t)::get(get_sequencer(), this.get_type_name(), "count", count))
                uvm_config_db#(int)::get(get_sequencer(), this.get_type_name(), "count", count);
        endtask
    endclass
```

Class type name

Sequencer path

 Recommended

5-33

Configuring Sequences (Sequencer-Based)

■ Set in test

```
class test_20_items extends test_base;
    // utils and constructor not shown
    virtual function void build_phase(...); super.build_phase(...);
        uvm_config_db#(int)::set(this, "env.agt.sqr", "count", 20);
    endfunction
endclass
```

Full path to sequencer

■ Reference configuration field through get_sequencer()

```
class packet_sequence extends sequence_base; // macros not shown
    int count = 10;
    virtual task pre_start();
        super.pre_start();
        if (!uvm_config_db#(uvm_stream_t)::get(this.get_sequencer(), "", "count", count))
            uvm_config_db#(int)::get(this.get_sequencer(), "", "count", count);
    endtask
endclass
```

Sequencer path

Empty string

5-34

Configuring Sequences (Agent-Based)

■ Set in test

```
class test_agent_configuration extends test_base;
    virtual function void build_phase(...); ...
    uvm_config_db#(int)::set(this, "env.agt", "port_id", 3);
endfunction
endclass
```

Diagram annotations:

- A callout box labeled "Agent configuration field" points to the line `"env.agt", "port_id", 3;`.
- A callout box labeled "Path to agent" points to the line `uvm_config_db#(int)::set(this, "env.agt", "port_id", 3);`.

■ Retrieve agent configuration field through sequencer

```
class packet_sequence extends sequence_base; // other code not shown
int port_id;
virtual task pre_start();
    uvm_sequencer_base my_sqr = get_sequencer();
    super.pre_start();
    if (!uvm_config_db#(uvm_bitstream_t)::get(my_sqr.get_parent(), "", "port_id", port_id))
        uvm_config_db#(int)::get(my_sqr.get_parent(), "", "port_id", port_id);
endtask
endclass
```

Diagram annotations:

- A callout box labeled "Empty string" points to the line `if (!uvm_config_db#(uvm_bitstream_t)::get(my_sqr.get_parent(), "", "port_id", port_id))`.
- A callout box labeled "Full path to agent" points to the line `my_sqr.get_parent()`.
- A callout box labeled "Agent configuration field" points to the line `"port_id", port_id);`.

Appendix

- Run-time Switch Configuration
- Sequence Configuration
- Array Configuration**
- Enum Configuration
- Configuration Debugging
- Disabling Auto Field Configuration

5-36

Configuring Array Members

- The following does not compile

```
class packet_sequence extends base_sequence; // some code left off
    bit valid_ports[16]; // Array to be configured - can be any type of array
    virtual task pre_start();
        uvm_config_db#(bit)::get(get_sequencer(), "", "valid_ports", valid_ports);
    endtask
endclass
```

Configuration can not handle arrays directly

- The following works but not suitable if width changes

```
class packet_sequence extends base_sequence; // some code left off
    bit      valid_ports[16];
    bit[15:0] packed_ports;
    uvm_agent agt = get_sequencer().get_agent();
    virtual task pre_start();
        uvm_config_db#(bit[15:0])::get(agt, "", "valid_ports", packed_ports);
        valid_ports = {>>{packed_ports}};
    endtask
endclass
```

What if size changes?

5-37

Configuring Array

- Create a user array data type

```
typedef bit bit_array_t [];
```

Can be any array type, can be any size

- Use user created data type to configure arrays

```
class my_test extends uvm_test; // other code left off
    bit_array_t valid_ports;
    virtual function void build_phase(uvm_phase phase); // other code
        valid_ports = new[16] ('{default:1'b1});
        uvm_config_db#(bit_array_t)::set(this, "env.agt", "valid_ports", valid_ports);
    endfunction
endclass
```

Database handles all data types defined by typedef

```
class packet_sequence extends base_sequence; // other code left off
    bit_array_t valid_ports;
    uvm_agent agt = get_sequencer().get_agent();
    virtual task body(); // other code
        uvm_config_db#(bit_array_t)::get(seq, "", "valid_ports", valid_ports);
    endtask
endclass
```

5-38

Appendix

- Run-time Switch Configuration
- Sequence Configuration
- Array Configuration
- Enum Configuration**
- Configuration Debugging
- Disabling Auto Field Configuration

5-39

Configure enum Field via string

■ New class: uvm_enum_wrapper

- Converts **string** to **enum**



UVM-1.2 and IEEE
ONLY!

```
class uvm_enum_wrapper#(type T=uvm_active_passive_enum);
    static function bit from_name(string name, ref T value);
endclass
```

■ Component field can be an **enum** variable and configured as a **string**

```
class my_component extends uvm_component; // other coder left off
    typedef enum {BASIC, INTERMEDIATE, ADVANCED} mode_e;
    typedef uvm_enum_wrapper#(mode_e) enum_wrapper_t;
    string mode_string; mode_e mode;
    virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        uvm_config_db#(string)::get(this, "", "mode", mode_string);
        enum_wrapper_t::from_name(mode_string, mode);
    endfunction
endclass
```

```
simv +uvm_set_config_string=*, mode, ADVANCED
```

5-40

Appendix

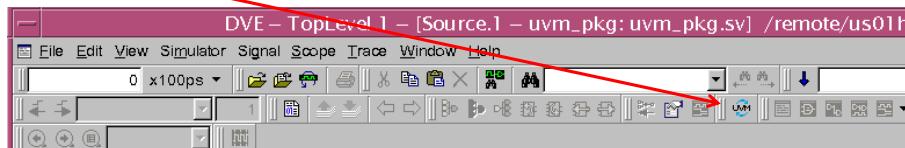
- Run-time Switch Configuration**
- Sequence Configuration**
- Array Configuration**
- Enum Configuration**
- Configuration Debugging**
- Disabling Auto Field Configuration**

5- 41

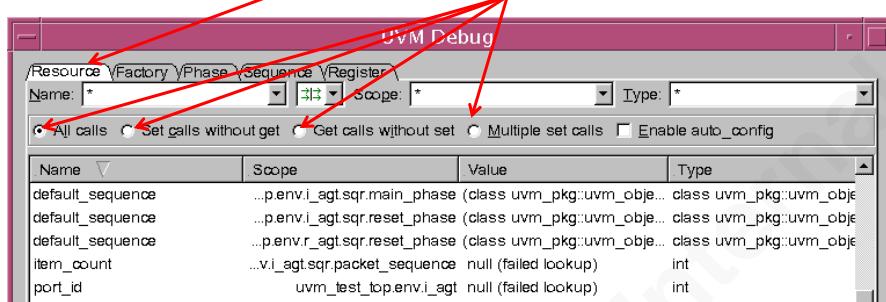
UVM DVE Configuration Debugging

- Run interactive simulation to the point where configuration failed

Then, click on 



- Select Resource and choose what you want to see



5-42

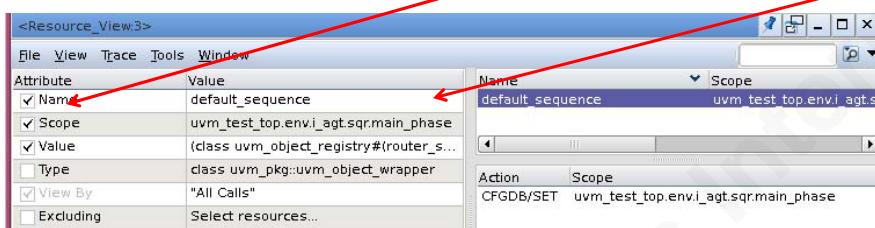
UVM Verdi Configuration Debugging

- Run interactive simulation to the point where configuration failed

Then click and select UVM -> Resource/Config View



- In the Resource View window, click on Attributes and set the Value that you want to use as filter to choose what you want to see



5-43

Appendix

- Run-time Switch Configuration**
- Sequence Configuration**
- Array Configuration**
- Enum Configuration**
- Configuration Debugging**
- Disabling Auto Field Configuration**

5- 44

UVM Field Macro Auto Configuration Issues

```
class packet extends uvm_sequence_item;
  rand bit [47:0] data;
  `uvm_object_utils_begin(packet)
    `uvm_field_int(data, UVM_ALL_ON)
  `uvm_object_utils_end
endclass
```

```
class driver extends uvm_driver#(packet);
  int port_id;
  virtual router_io vif;
  `uvm_component_utils_begin(driver)
    `uvm_field_int(port_id, UVM_ALL_ON)
  `uvm_component_utils_end
endclass
```

■ **`uvm_field_* issues**

- Inconsistent behavior
 - ◆ Automatically retrieves field from `uvm_config_db` in component classes
 - ◆ Does **NOT** retrieve field from `uvm_config_db` in non-component classes
- Performance hit in component classes
 - ◆ `uvm_config_db#(...)::get(...)` is executed for all fields even if retrieval is not wanted (can lead to performance hit)

5- 45

Disabling Auto Configuration Retrieval: UVM-1.1 & 1.2

- Automatic `uvm_config_db` retrieval can be disabled by overriding the `apply_config_settings()` method

- Leave out `super.apply_config_settings()` call
 - ◆ Generally not a good idea
 - Loose all other needed behavior of base class implementation of `apply_config_settings()` method



UVM-1.1 and UVM-1.2

```
class driver extends uvm_driver#(packet);
    int port_id;
    virtual router_io vif;
    `uvm_component_utils_begin(driver)
        `uvm_field_int(port_id, UVM_ALL_ON)
    `uvm_component_utils_end
    virtual function void apply_config_settings(bit verbose=0);
    endfunction
endclass
```

Do not call `super.apply_config_settings()`!

5-46

Disabling Auto Configuration Retrieval: IEEE UVM (1/2)

- Automatic `uvm_config_db` retrieval can be disabled by overriding the `use_automatic_config` method
 - Does not affect other base class `apply_config_settings()` implementation

```
class driver extends uvm_driver#(packet);
    int port_id;
    virtual router_io vif;
    `uvm_component_utils_begin(driver)
        `uvm_field_int(port_id, UVM_ALL_ON)
    `uvm_component_utils_end
    virtual function bit use_automatic_config();
        return 0;
    endfunction
endclass
```



IEEE UVM Only

Called by `apply_config_settings()`
return value of 0 disables auto retrieval of configuration database

Disabling Auto Configuration Retrieval: IEEE UVM (2/2)

- One can selectively disable auto retrieval in `uvm_field macro
 - UVM_NOSET
 - ◆ The field will not participate in the apply_config_settings operation

```
class driver extends uvm_driver#(packet);
    int port_id;
    virtual router_io vif;
    `uvm_component_utils_begin(driver)
        `uvm_field_int(port_id, UVM_ALL_ON | UVM_NOSET)
    `uvm_component_utils_end
endclass
```



IEEE UVM Only

5- 48

Agenda

DAY

2

5 UVM Configuration & Factory

6 UVM Component Communication



7 UVM Scoreboard & Coverage

8 UVM Callback



Synopsys 40-I-055-SSG-007

© 2019 Synopsys, Inc. All Rights Reserved

6-1

Unit Objectives



After completing this unit, you should be able to:

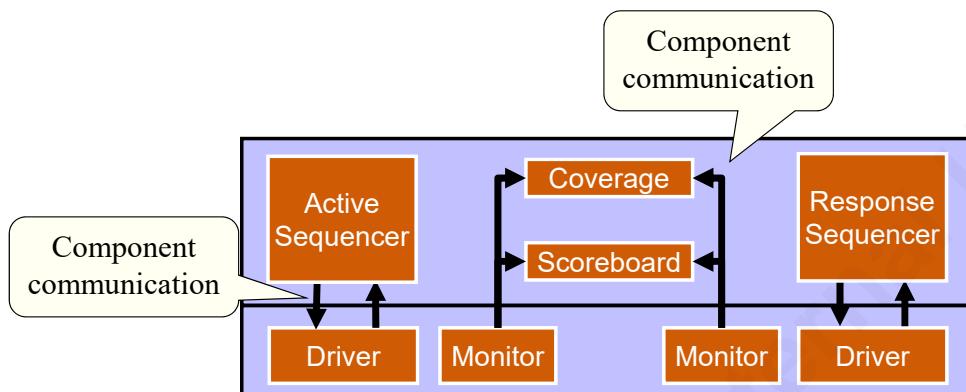
- **Describe and implement TLM port/socket for communication between components**

6-2

Component Communication: Overview

- **Transaction exchanges between verification environment components**

- Sequencer → Driver
- Monitor → Collectors (Scoreboard, Coverage)



6-3

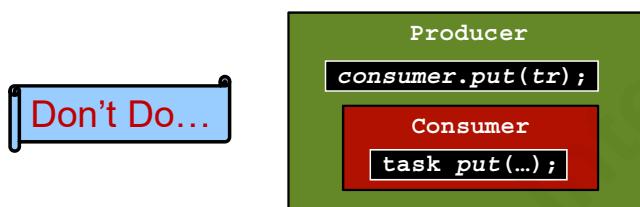
Component Communication: Method Based

- Component can embed method for communication

```
class Consumer extends uvm_component;  
  ...  
  virtual task put(transaction tr);  
endclass
```

- But, the communication method should not be called through the component object's handle

- Code becomes too inflexible for testbench structure
 - ◆ In example below, Producer is stuck with communicating with only a specific Consumer type



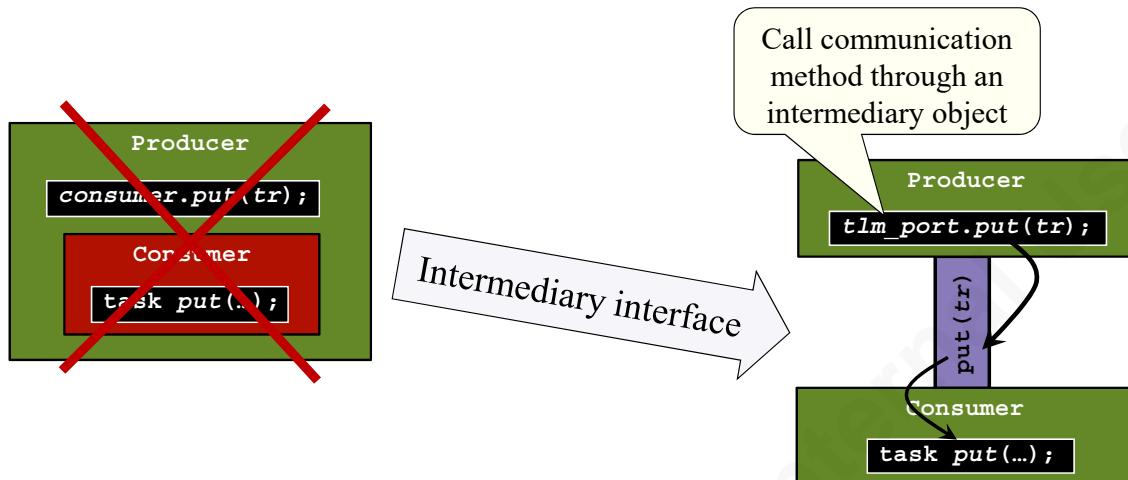
6-4

In the example at the bottom of the slide, because the Producer class uses a handle of type Consumer, it is restricted to communicating with just that type of object.

Another issue is that in a random configuration, there might not be a consumer object there to receive the transaction. So the producer will get a null object error.

Component Communication: TLM

- Use an intermediary object (TLM) to handle the communication



6-5

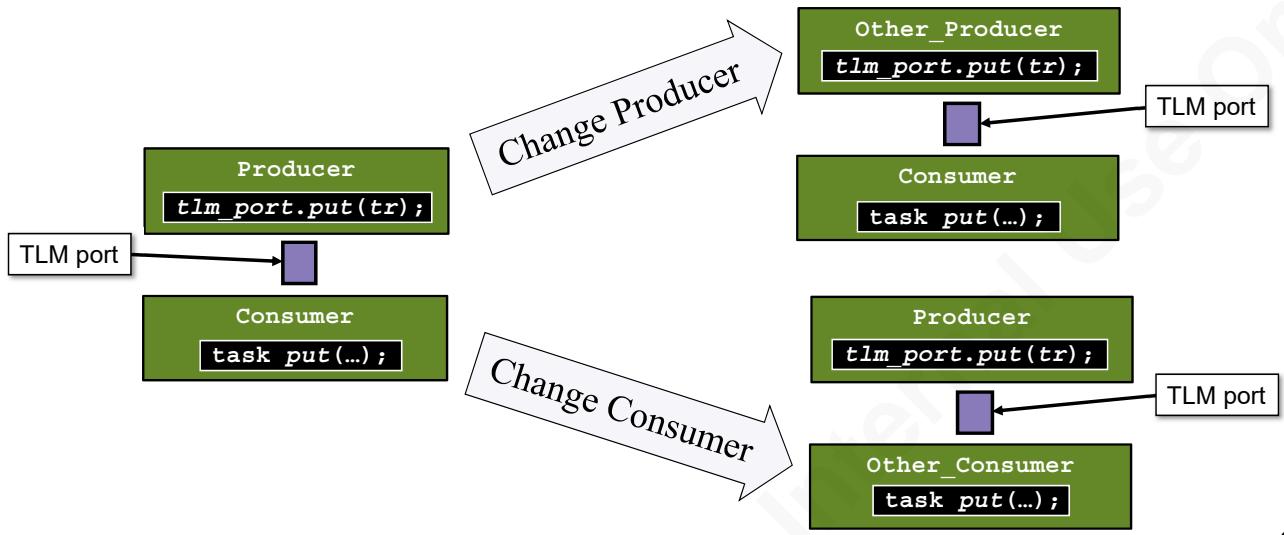
The TLM object provides a level of indirection between the producer and consumer so that they can be decoupled.

The Producer only knows about the generic TLM port, not the consumer object. The Consumer just needs to provide a put() method.

The connections between the components are established at the parent level.

Component Communication: TLM

- Through the intermediary object (TLM), components can be re-connected to any other component on a testcase by testcase basis



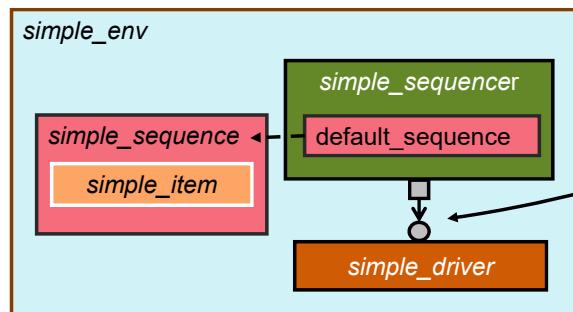
6-6

Now your components are independent of each other and thus can be reused more easily.

Communication in UVM: TLM 1.0, 2.0

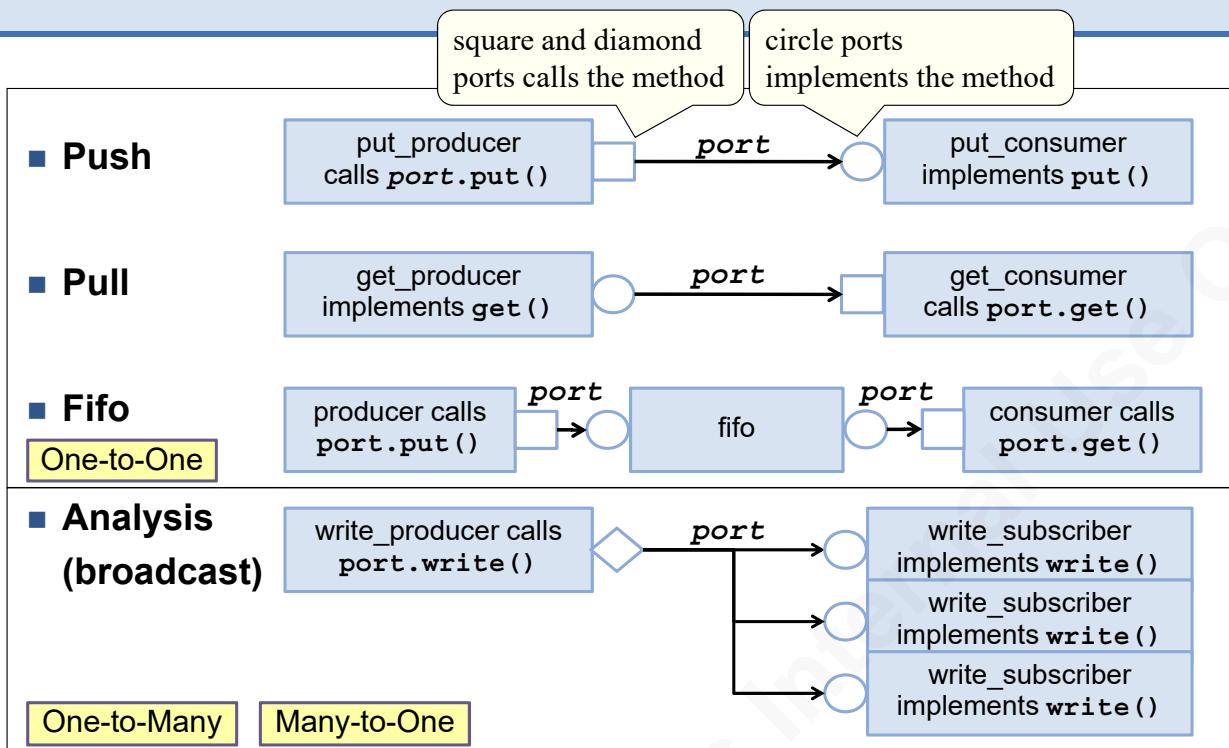
■ TLM Classes

- `uvm_*_export`
- `uvm_*_imp`
- `uvm_*_port`
- `uvm_*_fifo`
- `uvm_*_socket`



6-7

UVM TLM 1.0



Four types of One-to-One classes:

`*_port`

Used in component which calls the interface method (e.g. `put_producer`, `get_consumer` etc.)

`*_imp`

Used in component where the interface method is implemented (e.g. `put_consumer`, `get_producer` etc.)

`*_export`

Used as a pass-through port in components to connect port to implementation ports (on later slide)

`*_fifo` (on later slide)

Used to connect port to port (e.g. fifo)

Has built-in buffer

The put/get/fifo TLM ports requires that a producer is connected to a consumer.

With analysis port, a producer can be connected to any number of subscribers including 0 (no subscriber).

Push Mode

```
class producer extends uvm_component; ...
  uvm_blocking_put_port #(packet) put_port;
  function void build_phase(uvm_phase phase); ...
    put_port = new("put_port", this);
  endfunction
  virtual task initiate_tr(); ...
    put_port.put(tr);
  endtask
endclass
```

6-9

```
class consumer extends uvm_component; ...
  uvm_blocking_put_imp #(packet, consumer) put_export;
  function void build_phase(uvm_phase phase); ...
    put_export = new("put_export", this);
  endfunction
  virtual task put(packet tr);
    process_tr(tr);
  endtask
endclass
```

```
class environment extends uvm_env; producer p; consumer c; ...
  virtual function void connect_phase(uvm_phase phase); ...
    p.put_port.connect(c.put_export); // connection required!
  endfunction
endclass
```

```
graph LR
    subgraph Producer [producer]
        direction TB
        P[put_port] -- "calls port.put()" --> Port[port]
    end
    subgraph Consumer [consumer]
        direction TB
        C[put_consumer] -- "implements put()" --> Export[put_export]
    end
    subgraph Environment [environment]
        direction TB
        Env[connect_phase] -- "p.put_port.connect(c.put_export)" --> Connect[// connection required!]
    end
    Port --- Port
    Port --- Export
```

In the PUSH mode, the producer send transaction without request from consumer.

Two common names for TLM port handles

*_port:

TLM handle with this suffix designates the parent component as the component which calls the interface method

(typically in Run-Time phases)

This suffix is applied to TLM ports of *_port type

*_export

TLM handle with this suffix designates the parent component with the responsibility of exporting the interface method (i.e. must implement interface method) for external use

This suffix is applied to TLM ports of *_imp or *_export type

Pull Mode



In the PULL mode, the producer waits for the consumer to make the request call to send the transaction.

FIFO Mode



■ Connect producer to consumer via uvm_tlm_fifo

```
class environment extends uvm_env; ...
producer p;
consumer c;
uvm_tlm_fifo #(packet) tr_fifo;
virtual function void build_phase(uvm_phase phase); ...
    p = producer::type_id::create("p", this);
    c = consumer::type_id::create("c", this);
    tr_fifo = new("tr_fifo", this); // No proxy (type_id) for TLM ports
endfunction
virtual function void connect_phase(uvm_phase phase); ...
    p.put_port.connect(tr_fifo.put_export);
    c.get_port.connect(tr_fifo.get_export); endfunction
endclass
```

6-11

Please note that the fifo implementation follows the port name guideline: use *_export as the port name where the implementation of the communication method is done.

Analysis Port

- Can be left unconnected

```
class producer extends uvm_component;
  uvm_analysis_port #(packet) analysis_port;
  // other code left off
  virtual task assemble_tr(); ...
    analysis_port.write(tr);
  endtask
endclass

class subscriber extends uvm_component;
  uvm_analysis_imp #(packet, subscriber) analysis_export;
  // other code left off
  virtual function void write(packet tr); // cannot block
    process_transaction(tr);
  endfunction
endclass

class environment extends uvm_env; ...
  producer p; subscriber s0, s1; // other subscribers
  virtual function void connect_phase(uvm_phase phase);
    p.analysis_port.connect(s0.analysis_export);
    p.analysis_port.connect(s1.analysis_export);
  endfunction
endclass
```

The diagram illustrates the connection between a producer and two subscribers. A blue box labeled "write_producer calls port.write()" has a line pointing to a diamond-shaped connector labeled "port". From this connector, two lines branch out to two separate blue boxes, each labeled "write_subscriber implements write()".

6-12

Four analysis port class types:

* _analysis_port

Used in initiator (which calls the interface method, write_producer)

* _analysis_imp

Used in subscriber (where the interface method is implemented, write_subscriber)

* _analysis_export

Use as a pass-through port in subscriber

* _analysis_fifo

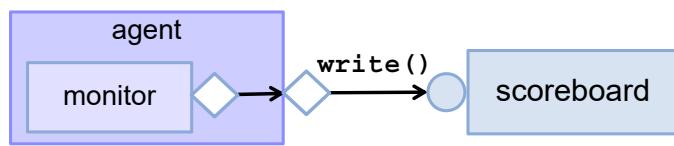
Port with embedded buffer to connect analysis port to get port

An analysis producer can connect to any number of subscribers, including 0.

Port Pass-Through

■ Connecting sub-component TLM ports

- Use same port type



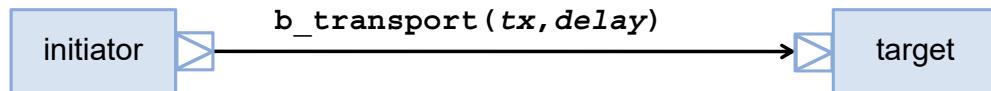
```
class monitor extends uvm_monitor; // other code not shown ...
  uvm_analysis_port #(packet) analysis_port;
  virtual function void build_phase(uvm_phase phase);
    this.analysis_port = new("analysis_port", this);
  endfunction
endclass

class agent extends uvm_agent; // other code not shown ...
  monitor mon;
  uvm_analysis_port #(packet) analysis_port;
  virtual function void build_phase(uvm_phase phase);
    this.analysis_port = new("analysis_port", this);
  endfunction
  virtual function void connect_phase(uvm_phase phase);
    mon.analysis_port.connect(this.analysis_port);
  endfunction
endclass
```

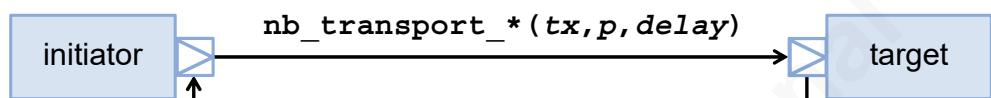
port
must be
same
type

6-13

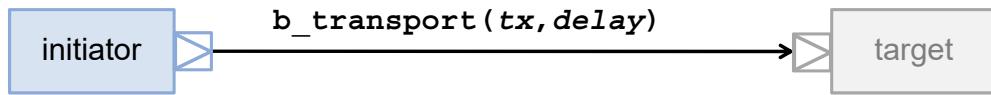
- **Blocking**



- **Non-Blocking**



Blocking Transport Initiator



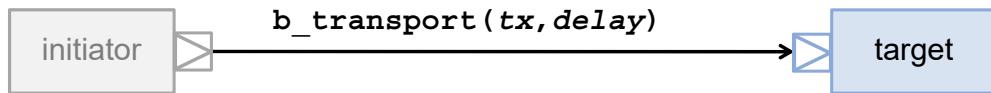
```
class initiator extends uvm_component;
  uvm_tlm_b_initiator_socket #(packet) i_socket;
  // other code not shown
  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    i_socket=new("i_socket", this);
  endfunction
  virtual task initiate_tr();
    packet tx = packet::type_id::create("tx", this);
    uvm_tlm_time delay = new();
    delay.set_abstime(1.5, 1e-9); // set delay to 1.5ns
    i_socket.b_transport(tx, delay);
  endtask
endclass
```

6-15

The **uvm_tlm_time** class is defined as follows:

```
class uvm_tlm_time;
  static local real m_resolution = 1.0e-12; // ps by default
  static function void set_time_resolution(real res);
  function new(string name = "uvm_tlm_time", real res = 0);
  function void reset();
  function real get_realtime(time scaled, real secs = 1.0e-9);
  function void incr(real t, time scaled, real secs = 1.0e-9);
  function void decr(real t, time scaled, real secs);
  function real get_abstime(real secs);
  function void set_abstime(real t, real secs);
endclass
```

Blocking Transport Target



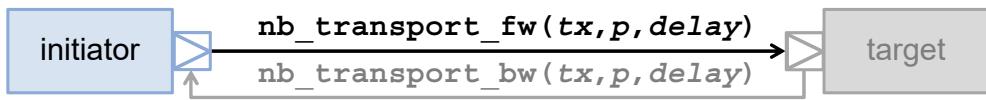
```
class target extends uvm_component; ...
  uvm_tlm_b_target_socket #(target, packet) t_socket;
  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    t_socket=new("t_socket", this);
  endfunction

  virtual task b_transport(packet tx, uvm_tlm_time delay);
    $display("realtime = %t", delay.get_realtime(1ns));
    ...
  endtask
endclass

class environment extends uvm_env;
  initiator intr;
  target trgt;
  // component_utils, constructor and build_phase not shown
  virtual function void connect_phase(uvm_phase phase);
    intr.i_socket.connect(trgt.t_socket);
  endfunction
endclass
```

6-16

Non-Blocking Transport Initiator



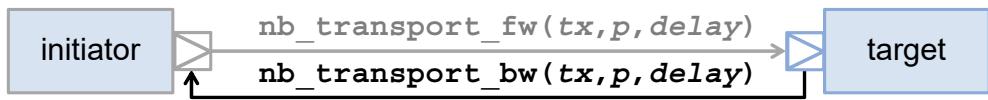
```
class initiator extends uvm_component;
    uvm_tlm_nb_initiator_socket #(initiator, packet) i_socket;
    // component_utils, constructor and build_phase not shown
    virtual task initiate_tr();
        uvm_tlm_sync_e sync; uvm_tlm_phase_e p; uvm_tlm_time delay = new;
        packet tx = packet::type_id::create("tx", this);
        ...
        sync = i_socket.nb_transport_fw(tx, p, delay);
    endtask
    virtual function uvm_tlm_sync_e nb_transport_bw(packet tx,
                                                    ref uvm_tlm_phase_e p, input uvm_tlm_time delay);
        // ... Process acknowledgement from target
        return (UVM_TLM_COMPLETED);
    endfunction
endclass
```

6-17

The `uvm_tlm_phase_e` and `uvm_tlm_sync_e` enums are defined as follows:

```
typedef enum
{
    UVM_TLM_ACCEPTED,
    UVM_TLM_UPDATED,
    UVM_TLM_COMPLETED
} uvm_tlm_sync_e;
typedef enum
{
    UNINITIALIZED_PHASE,
    BEGIN_REQ,
    END_REQ,
    BEGIN_RESP,
    END_RESP
} uvm_tlm_phase_e;
```

Non-Blocking Transport Target



```
class target extends uvm_component;
    uvm_tlm_nb_target_socket #(target, packet) t_socket;
    // component_utils, constructor and build_phase not shown
    virtual function uvm_tlm_sync_e nb_transport_fw(packet tx,
                                                    ref uvm_tlm_phase_e p, input uvm_tlm_time delay);
        tx.print();
        fork process_tr(tx); join_none // for delayed acknowledgement
        return (UVM_TLM_ACCEPTED);
    endfunction
    virtual task process_tr(packet tx);
        uvm_tlm_sync_e sync; uvm_tlm_phase_e p; uvm_tlm_time delay = new;
        // ... After completion of tx processing
        sync = t_socket.nb_transport_bw(tx, p, delay);
    endtask
endclass
```

6-18

Unit Objectives Review

You should now be able to:

- **Describe and implement TLM port/socket for communication between components**



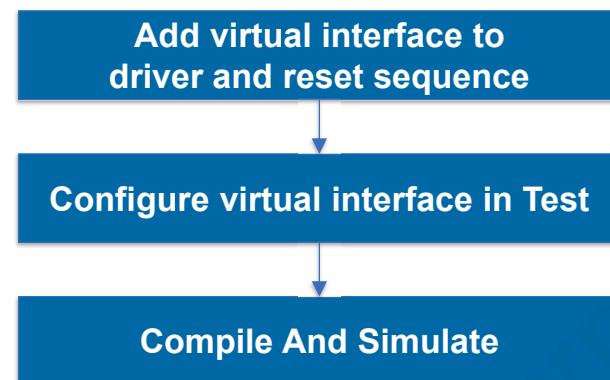
6-19

Lab 3: Driving the DUT



60 minutes

Implement & configure physical device drivers



6-20

Appendix

TLM 2.0 Generic Payload

Verdi UVM-Aware Debugging Features

6-21

TLM 2.0 Generic Payload (1/4)

- TLM 2.0 Generic Payload is a generic bus read/write access transaction
 - Used for cross-platform interoperability (e.g. SystemC)

```
typedef enum {
    UVM_TLM_READ_COMMAND,           // Bus read operation
    UVM_TLM_WRITE_COMMAND,          // Bus write operation
    UVM_TLM_IGNORE_COMMAND          // No bus operation
} uvm_tlm_command_e;

typedef enum {
    UVM_TLM_OK_RESPONSE = 1,        // Bus operation completed successfully
    UVM_TLM_INCOMPLETE_RESPONSE = 0, // Transaction was not delivered to target
    UVM_TLM_GENERIC_ERROR_RESPONSE = -1, // Bus operation had an error
    UVM_TLM_ADDRESS_ERROR_RESPONSE = -2, // Invalid address specified
    UVM_TLM_COMMAND_ERROR_RESPONSE = -3, // Invalid command specified
    UVM_TLM_BURST_ERROR_RESPONSE = -4, // Invalid burst specified
    UVM_TLM_BYTE_ENABLE_ERROR_RESPONSE = -5 // Invalid byte enabling specified
} uvm_tlm_response_status_e;
```

6-22

TLM 2.0 Generic Payload (2/4)

```
class uvm_tlm_generic_payload extends uvm_sequence_item;
  rand bit [63:0] m_address;
  rand uvm_tlm_command_e m_command;
  rand byte unsigned m_data[];
  rand int unsigned m_length; // Number of bytes to be copied to or from the m_data array
  rand uvm_tlm_response_status_e m_response_status;
  rand byte unsigned m_byte_enable[];
  rand int unsigned m_byte_enable_length; // Number of elements in m_byte_enable array
  rand int unsigned m_streaming_width; // Number of bytes transferred on each beat
`uvm_object_utils_begin(uvm_tlm_generic_payload)
  `uvm_field_int(m_address, UVM_ALL_ON);
  `uvm_field_enum(uvm_tlm_command_e, m_command, UVM_ALL_ON);
  `uvm_field_array_int(m_data, UVM_ALL_ON);
  `uvm_field_int(m_length, UVM_ALL_ON);
  `uvm_field_enum(uvm_tlm_response_status_e, m_response_status, UVM_ALL_ON);
  `uvm_field_array_int(m_byte_enable, UVM_ALL_ON);
  `uvm_field_int(m_streaming_width, UVM_ALL_ON);
`uvm_object_utils_end
... Continued on next slide
```

6-23

TLM 2.0 Generic Payload (3/4)

```
function new(string name="");
virtual function uvm_tlm_command_e get_command();
virtual function void set_command(uvm_tlm_command_e command);
virtual function bit is_read();
virtual function void set_read();
virtual function bit is_write();
virtual function void set_write();
virtual function void set_address(bit [63:0] addr);
virtual function bit [63:0] get_address();
virtual function void get_data (output byte unsigned p []);
virtual function void set_data(ref byte unsigned p []);
virtual function int unsigned get_data_length();
virtual function void set_data_length(int unsigned length);
virtual function int unsigned get_streaming_width();
virtual function void set_streaming_width(int unsigned width);
virtual function void get_byte_enable(output byte unsigned p[]);
virtual function void set_byte_enable(ref byte unsigned p[]);
virtual function int unsigned get_byte_enable_length();
virtual function void set_byte_enable_length(int unsigned length);
virtual function uvm_tlm_response_status_e get_response_status();
virtual function void set_response_status(uvm_tlm_response_status_e status);
virtual function bit is_response_ok();
virtual function bit is_response_error();
virtual function string get_response_string(); // Continued on next page
```

6-24

TLM 2.0 Generic Payload (4/4)

- **TLM 2.0 Generic Payload can be extended to add additional members**

- Requires implementation of `uvm_tlm_extension` class

```
// Continued from previous page
local uvm_tlm_extension_base m_extensions [uvm_tlm_extension_base];
function uvm_tlm_extension_base set_extension(uvm_tlm_extension_base ext);
function int get_num_extensions();
function uvm_tlm_extension_base get_extension(uvm_tlm_extension_base ext_handle);
function void clear_extension(uvm_tlm_extension_base ext_handle);
function void clear_extensions();
endclass
```

```
class uvm_tlm_extension #(type T=int) extends uvm_tlm_extension_base;
  typedef uvm_tlm_extension#(T) this_type;
  local static this_type m_my_tlm_ext_type = ID();
  function new(string name="");
    static function this_type ID();
    virtual function uvm_tlm_extension_base get_type_handle();
    virtual function string get_type_handle_name();
  endclass
```

6-25

Appendix

TLM 2.0 Generic Payload

Verdi UVM-Aware Debugging Features

6-26

Verdi UVM Debug Switches

For VCS 2017.12 and Verdi 2017.12 onwards

■ **Compile-time switches:**

- UVM Debug requires `-debug_access+all`
- Reverse debugging Requires `-debug_access+reverse`

■ **Post Simulation run-time switches:**

```
simv +UVM_VERDI_TRACE +UVM_TR_RECORD +UVM_LOG_RECORD \
      +UVM_TESTNAME=test_base
verdi -ssf novas.fsdb -nologo &
```

■ **Interactive Simulation run-time switches**

```
simv +UVM_VERDI_TRACE +UVM_TR_RECORD +UVM_LOG_RECORD \
      +UVM_TESTNAME=test_base -gui=verdi
```

6-27

If custom UVM source is used instead of the version shipped with vcs, the `VCS_UVM_HOME` environment variable must be set to point to the UVM source code directory.

For uvm-1.2, you will need to add the `uvm_custom_install_VCS_recorder.sv` file as part of the compile list to enable UVM transaction recording.

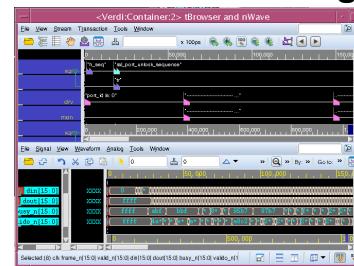
```
% setenv VCS_UVM_HOME <path to UVM/src>
% vcs -sverilog test.sv +incdir+${VCS_HOME}/etc/uvm-1.2/vcs\
${VCS_HOME}/etc/uvm-1.2/vcs/uvm_custom_install_vcs_recorder.sv
```

For IEEE UVM source code, the transaction recording is not yet implemented:

UVM-Aware Features in Verdi

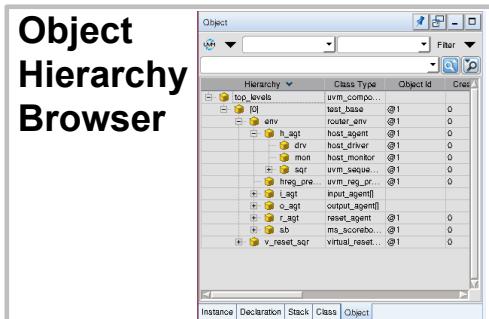
■ Post simulation debug

Transaction Recording

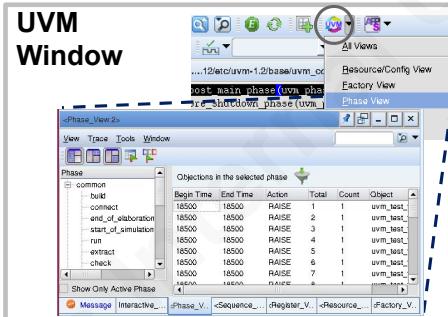


■ Interactive debug

Object Hierarchy Browser



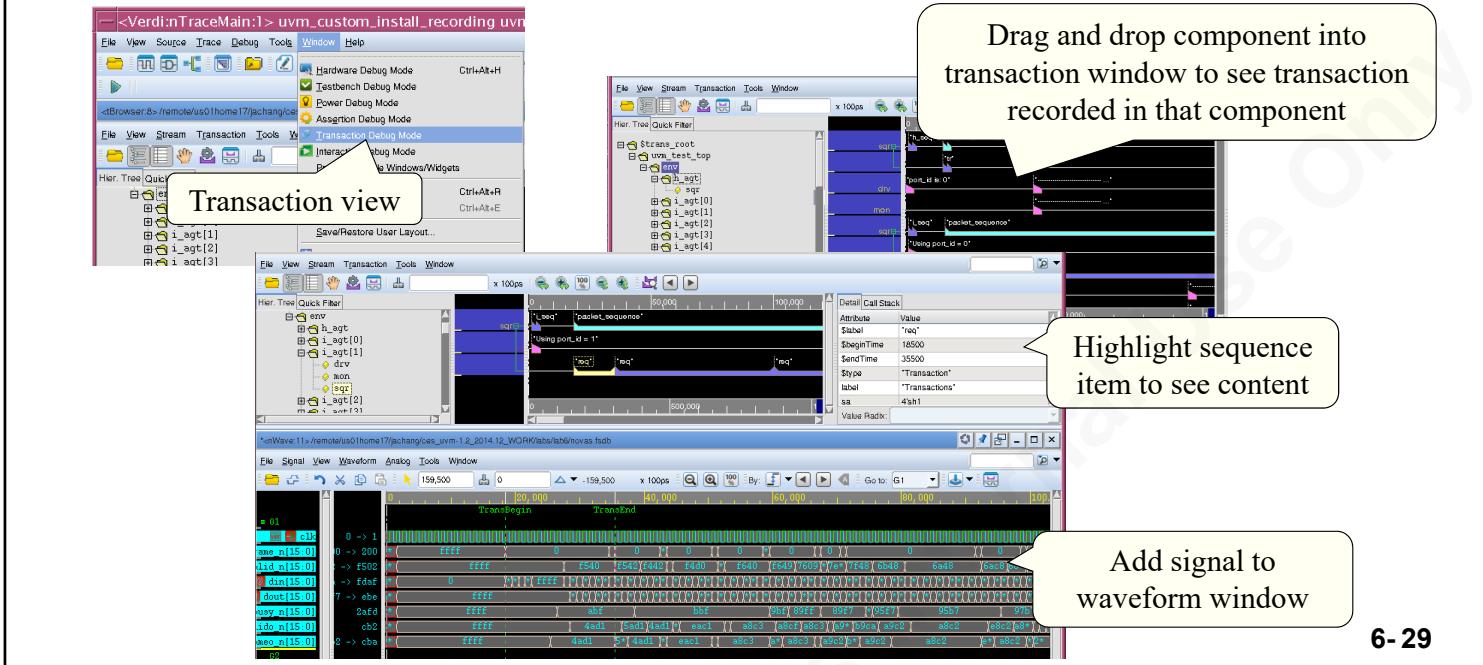
UVM Window



6-28

UVM Transaction and Log Debug

■ Available for both post and interactive simulation



6-29

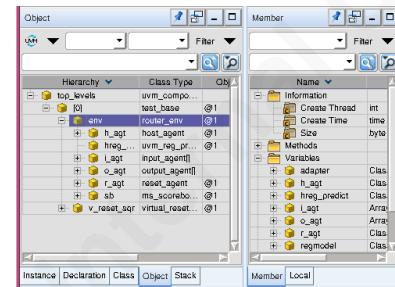
Object Hierarchy Browser

■ General mode and methodology-aware mode

- All Objects
- UVM (Objects/Components)
- OVM (Objects/Components)
- VMM (Objects/Components)

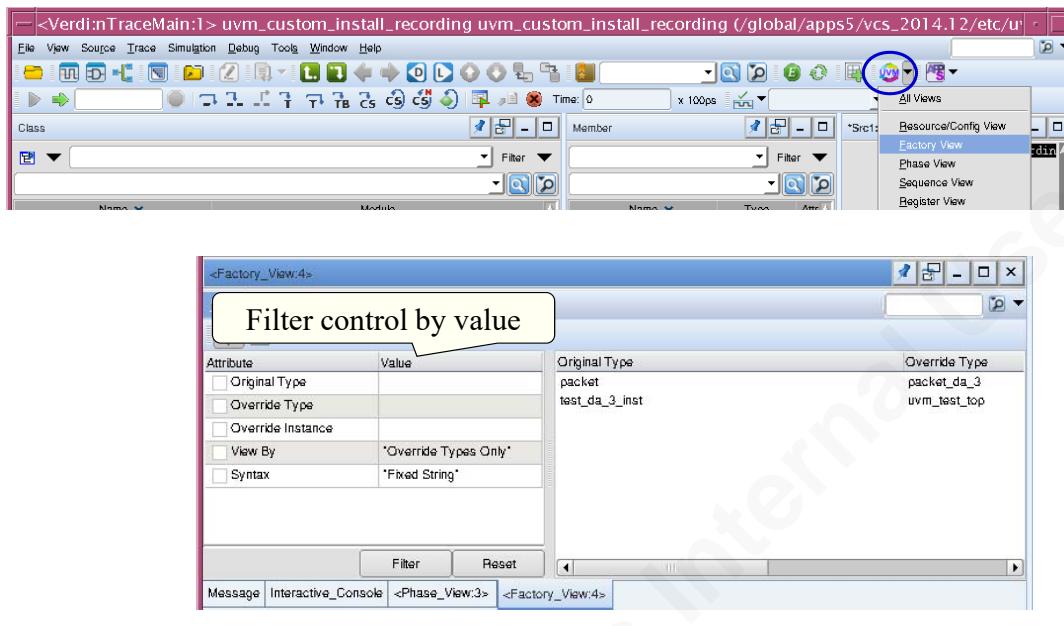
■ Display and navigate all dynamic variables

- Testbench hierarchy view
- Object creation time
- Object thread ids
- Object references
- Dynamic memory profiling
- Linked with Member Pane value annotation



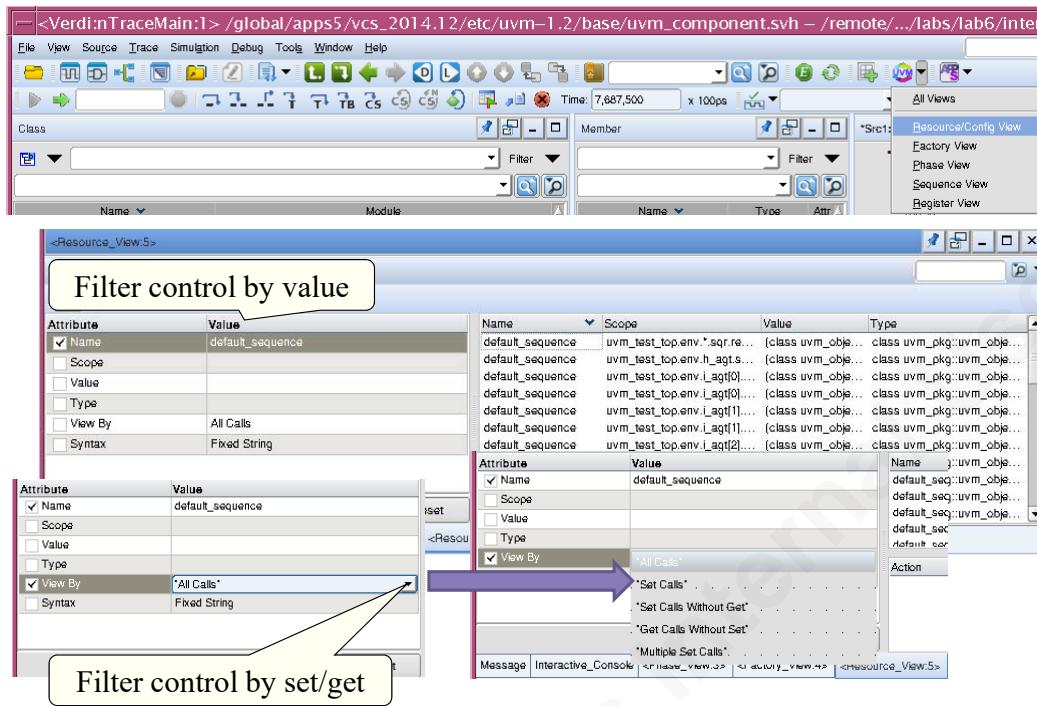
6-30

UVM Factory Debug



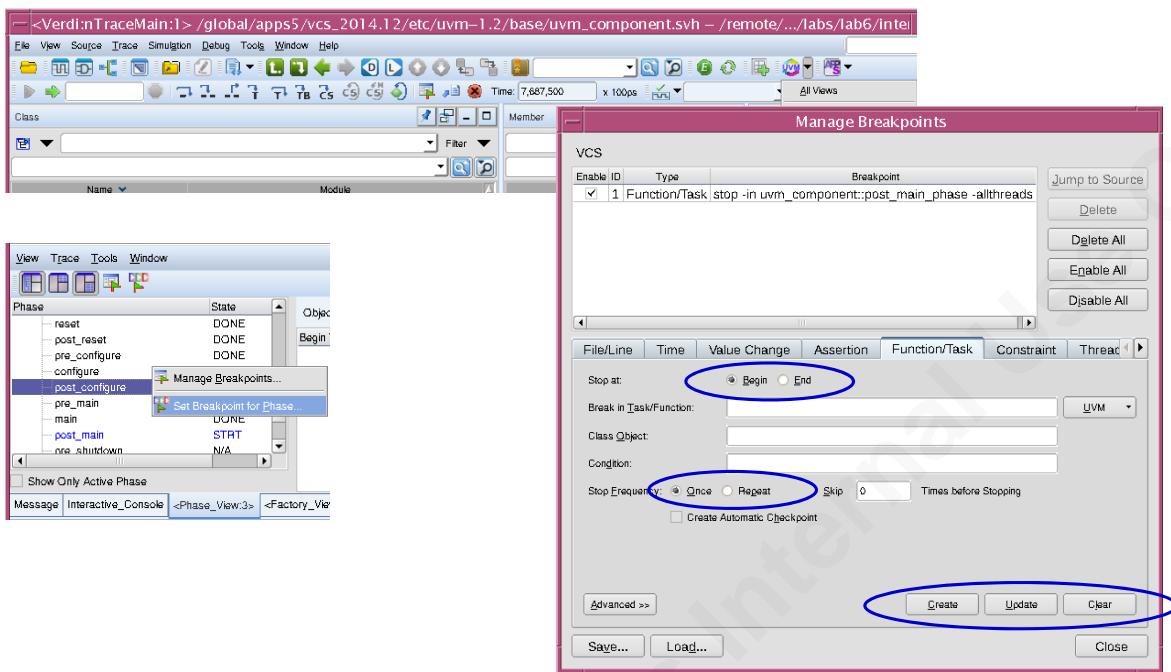
6-31

UVM Resource Debug



6-32

UVM Phase-Based Breakpoint



6-33

UVM Phase Objection Debug

Phase History

The screenshot shows the UVM Phase History window. The Phase History table lists various simulation phases and their states. The Objection History table shows objections raised during the selected phase (main).

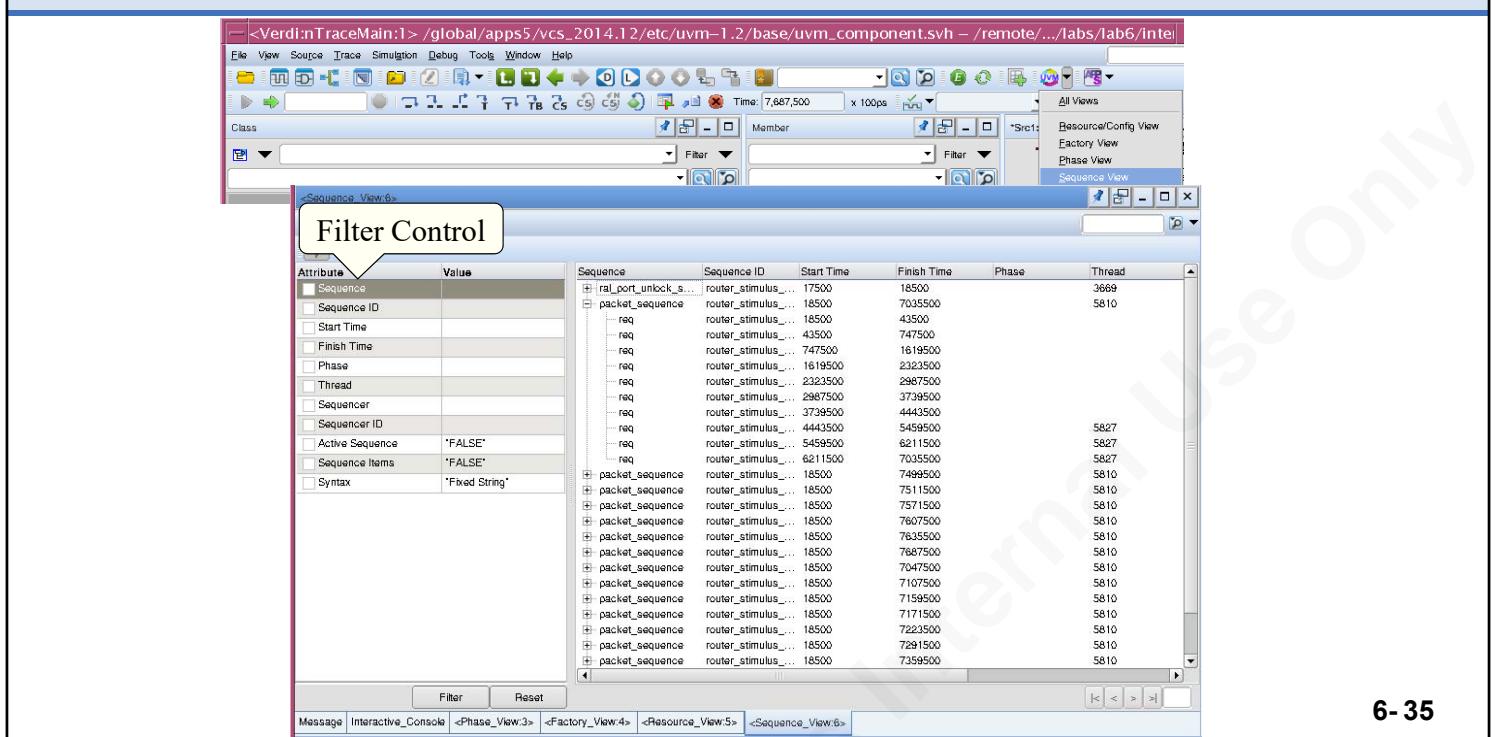
Phase	State	Time
common		
build		
connect		
end_of_elabs		
start_of_simulation	DONE	
run	SKIP	
extract	N/A	
check	N/A	
report	N/A	
final	N/A	
uvm		
pre_reset	DONE	
reset	DONE	
post_reset	DONE	
pre_configure	DONE	
configure	DONE	
post_configure	DONE	
pre_main	DONE	
main	DONE	
post_main	STRT	
pre_shutdown	N/A	
shutdown	N/A	
post_shutdown	N/A	

Domain	State	Time	
uvm	DONE	18500	
uvm	SCHEDULED	18500	
pre_main	STRT	18500	
pre_main	SKIP	18500	
pre_main	DONE	18500	
main	uvm	SCHEDULED	18500
main	uvm	STRT	18500
main	uvm	DONE	7687500
post_main	uvm	SCHEDULED	7687500
post_main	uvm	STRT	7687500

Objections in the selected phase					
Begin Time	End Time	Action	Total	Count	Object
18500	18500	RAISE	1	1	uvm_test_top.env.i.a...
18500	18500	RAISE	2	1	uvm_test_top.env.i.a...
18500	18500	RAISE	3	1	uvm_test_top.env.i.a...
18500	18500	RAISE	4	1	uvm_test_top.env.i.a...
18500	18500	RAISE	5	1	uvm_test_top.env.i.a...
18500	18500	RAISE	6	1	uvm_test_top.env.i.a...
18500	18500	RAISE	7	1	uvm_test_top.env.i.a...
18500	18500	RAISE	8	1	uvm_test_top.env.i.a...
18500	18500	RAISE	9	1	uvm_test_top.env.i.a...
18500	18500	RAISE	10	1	uvm_test_top.env.i.a...
18500	18500	RAISE	11	1	uvm_test_top.env.i.a...

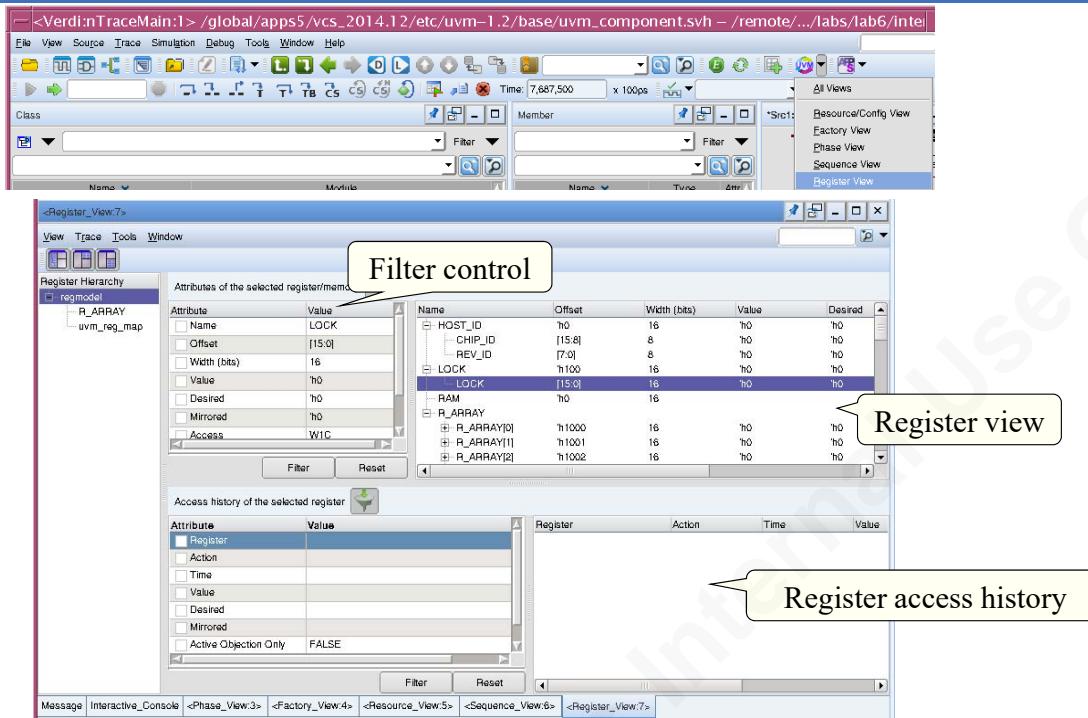
6-34

UVM Sequence Debug



6-35

UVM Register Debug



6-36

Agenda

DAY

2

5 UVM Configuration & Factory

6 UVM Component Communication



7 UVM Scoreboard & Coverage

8 UVM Callback



Unit Objectives



After completing this unit, you should be able to:

- Build re-usable self checking scoreboards by using the in-built UVM comparator classes
- Implement functional coverage

7-2

Scoreboard - Introduction

■ Today's challenges

- Self checking testbenches need scoreboards
- Develop a scoreboard once, re-use many times in different testbenches
- Need different scoreboarding mechanisms for different applications
- Must be aware of DUT's data transformation

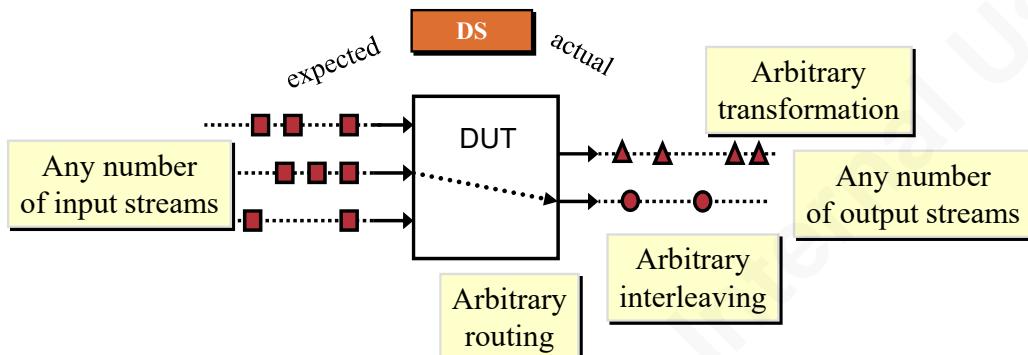
Solution: UVM scoreboard class extension with tailored functions for matching expected & observed data:

- In-order expects
- Built in Analysis Exports in the Comparator classes

Scoreboard – Data Streams

Any ordered data sequence. Not just packets.

Application	Streams
Networking	Packets in, packets out
DSP	Samples in, samples out
Modems, codecs	Frames in, code samples out
Busses, controllers	Requests in, responses out



7-4

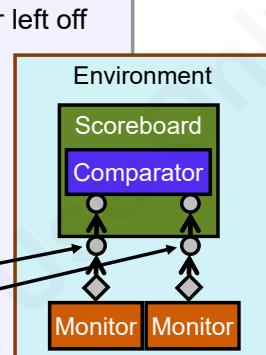
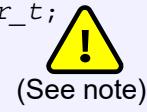
Scoreboard Implementation

- UVM provides `uvm_in_order_class_comparator` for data checking

```
class scoreboard extends uvm_scoreboard; // utils macro and constructor left off
  typedef uvm_in_order_class_comparator #(packet) cmp_r_t;
  cmp_r_t cmp_r;
  uvm_analysis_export #(packet) before_export;
  uvm_analysis_export #(packet) after_export;

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    cmp_r = cmp_r_t::type_id::create("cmp_r", this);
    before_export = new("before_export", this);
    after_export = new("after_export", this);
  endfunction

  virtual function void connect_phase(uvm_phase phase);
    before_export.connect(cmp_r.before_export);
    after_export.connect(cmp_r.after_export);
  endfunction
endclass
```



7-5

The `uvm_scoreboard` base class does not implement any scoreboarding mechanism. It is user's responsibility to develop the entire content of the scoreboard.

The `uvm_in_order_class_comparator` is part of the existing Accellera UVM source code release. However, it is not part of the IEEE UVM standard. Use with care!

It is being used here to illustrate how a pass-through port for an analysis port works.

Scoreboard Transaction Source: Monitor

■ Monitor embeds analysis port

```
class iMonitor extends uvm_monitor; ...
    virtual router_io vif;
    uvm_analysis_port #(packet) analysis_port;
    // uvm_component_utils macro and constructor
    virtual function void build_phase(...); ...
        analysis_port = new("analysis_port", this);
        if (!uvm_config_db#(virtual router_io)::get(this, "", "vif", vif))
            `uvm_fatal("CFGERR", ...);
    endfunction
    virtual task run_phase(uvm_phase phase);
        forever begin
            packet tr = packet::type_id::create("tr");
            get_packet(tr);
            analysis_port.write(tr);
        end
    endtask
    virtual task get_packet(packet tr); ...
endclass
```

The diagram illustrates the internal structure of a Monitor component. It consists of three main parts: an Analysis port (represented by a green box), a Scoreboard (represented by a green box), and a Comparator (represented by a purple box). The Analysis port is connected to both the Scoreboard and the Comparator. The Scoreboard and Comparator are connected to the Monitor (represented by an orange box). A callout labeled 'Get DUT interface' points to the 'vif' connection between the Monitor and the Scoreboard. Another callout labeled 'Pass observed transaction to collector components via TLM analysis port' points to the line 'analysis_port.write(tr)' in the code.

7-6

The example above omitted the following declarations: utility macros, and constructor:

```
`uvm_component_utils(iMonitor)

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

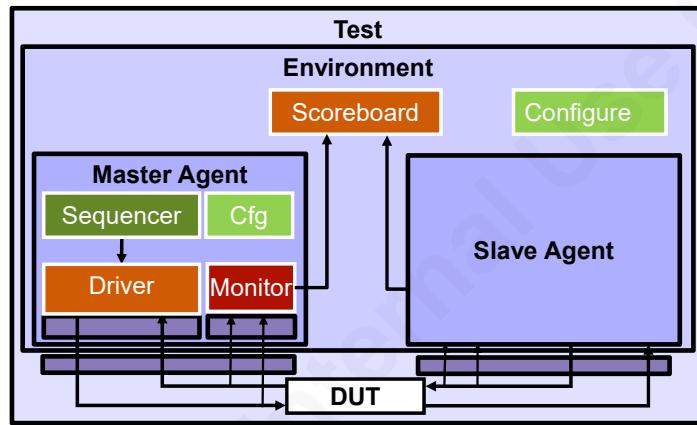
Embed Monitor in Agent

- **Agent extends from uvm_agent class**

- Contains a driver, a sequencer and a monitor
- Contains configuration and other parameters

- **Two operating modes:**

- Active:
 - ◆ Emulates a device in the system interacting with DUT
 - ◆ Instantiates a driver, sequencer and monitor
- Passive:
 - ◆ Operates passively
 - ◆ Only monitor instantiated and configured



7-7

An agent is a group of UVM components implemented to support execution of a protocol, such as AHB, PCI, etc.

An agent contains:

A Sequencer to pass a stream of transactions to the driver.

A Driver that gets transactions from the sequencer and drives them into the DUT

A Monitor that watches the DUT and creates an observed stream of transactions

A Configuration that holds agent-specific control fields

UVM Agent Example

```
class master_agent extends uvm_agent;
  uvm_analysis_port #(packet) analysis_port;
  sequencer sqr;
  driver   drv;    Sequencer, Driver and Monitor
  iMonitor mon;
  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    analysis_port = new("analysis_port", this);
    if (is_active == UVM_ACTIVE) begin
      sqr = sequencer::type_id::create("sqr",this);
      drv = driver::type_id::create("drv",this);
    end
    mon   = iMonitor::type_id::create("mon",this);
  endfunction: build_phase
  function void connect_phase(uvm_phase phase);
    mon.analysis_port.connect(this.analysis_port);
    if(is_active == UVM_ACTIVE)
      drv.seq_item_port.connect(sqr.seq_item_export);
  endfunction: connect_phase
endclass
```

Pass-through analysis port

Sequencer, Driver and Monitor

is_active flag is built-in

Create sequencer and driver if active

Connect Pass-through port

Connect sequencer to driver

7-8

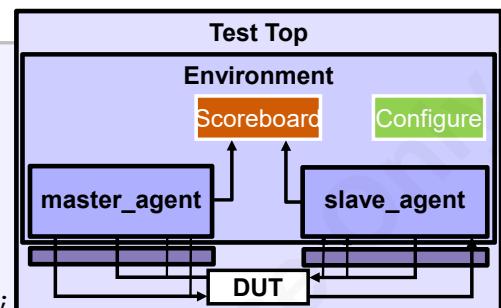
The example above omitted the following declarations: utility macros, and constructor:

```
`uvm_component_utils(master_agent)

function new(input string name, uvm_component parent);
  super.new(name, parent);
endfunction
```

Using UVM Agent in Environment

```
class router_env extends uvm_env;
    master_agent m_agt;
    slave_agent s_agt;
    scoreboard sb;
    // utils and constructor not shown
    virtual function void build_phase(...);
        super.build_phase(phase);
        m_agt = master_agent::type_id::create("m_agt", this);
        s_agt = slave_agent::type_id::create("s_agt", this);
        sb = scoreboard::type_id::create("sb", this);
        uvm_config_db#(uvm_active_passive_enum)::set(this, "m_agt", "is_active", UVM_ACTIVE);
        uvm_config_db#(uvm_active_passive_enum)::set(this, "s_agt", "is_active", UVM_ACTIVE);
    endfunction
    virtual function void connect_phase(uvm_phase phase);
        m_agt.analysis_port.connect(sb.before_export);
        s_agt.analysis_port.connect(sb.after_export);
    endfunction
endclass
```



Environment sets agent's mode of operation

7-9

The example above omitted the following declarations: utility macros, and constructor:

```
`uvm_component_utils(router_env)

function new(input string name, uvm_component parent);
    super.new(name, parent);
endfunction
```

Scoreboard Can Be Parameterized

■ Create a base class without parameter

```
virtual class sb_base extends uvm_scoreboard;  
    pure virtual task wait_for_expected_q_empty();  
        virtual function void clear_expected_q(); endfunction  
    endclass
```

Embed common methods

■ Then create parameterized class with required members

```
class packet_sb#(type in_type = packet, expected_type = in_type) extends sb_base;  
    typedef packet_sb #(T) this_type;  
    `uvm_component_param_utils(this_type)  
    const static string type_name = $sformatf("packet_sb#(%s)", T::type_name);  
    virtual function string get_type_name();  
        return type_name;  
    endfunction  
    virtual task wait_for_expected_q_empty(); ... endtask  
    virtual function void clear_expected_q(); ... endfunction  
endclass
```

Don't forget parameterized class requirements!

Implement common methods

7-10

The classes above is missing the constructor declaration. Make your classes do include the following:

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

Scoreboard: User Implementation (1/2)

■ Need multiple analysis implementation ports

```
class scoreboard #(type in_type = uvm_object, out_type = in_type) extends sb_base;
  // factory macro and constructor left off
  typedef scoreboard #(in_type, out_type) this_type;
  `uvm_analysis_imp_decl(_in)
  `uvm_analysis_imp_decl(_out)
  uvm_analysis_imp_in #(in_type, this_type) in_export;
  uvm_analysis_imp_out #(out_type, this_type) out_export;
  int m_matches = 0, m_mismatches = 0;
  out_type expected_q[$];
  virtual function void build_phase(uvm_phase phase);
    super.build_phase();
    in_export = new("in_export", this);
    out_export = new("out_export", this);
  endfunction
  virtual task wait_for_expected_q_empty(); ... endtask
  virtual function void clear_expected_q(); ... endfunction
// continued on next slide
```

Use macro to create
distinct **write()**
methods for analysis port

7-11

The above code left off the **uvm_component_param_utils** macro, **type_name**, and **get_type_name()**, **wait_for_expected_q_empty()**, **clear_expected_q()** method implementations. These are required, but are left off the slide in the interest of slide real estate.

Scoreboard: User Implementation (2/2)

■ Narrow down potential matches with a tag search

```
// continued from previous slide
virtual function void write_in(in_type in_tr);
    out_type expected;
    expected = transform(in_tr); // transform here-code not shown
    expected_q.push_back(expected);
endfunction

virtual function void write_out(out_type out_tr);
    int index[$];
    index = expected_q.find_index() with (item.quick_match(out_tr));
    foreach(index[i]) begin
        if (out_tr.compare(expected_q[index[i]])) begin
            expected_q.delete(index[i]);
            m_matches++;
            return;
        end
    end
    `uvm_warning("SB_ERR", "No match found");
    m_mismatches++;
endfunction

endclass
```

Distinct analysis port `write()` methods

Distinct analysis port `write()` methods

Do a quick search first

Do full compare on results of quick search

7-12

The heart of this scoreboard is the queue of expected packets, `expected`, and the SystemVerilog search function `find_index()` that searches for all element that satisfies the search expression. This function returns a queue of indices for all matching entries or an empty queue if there was no match.

The search expression just compares the `da` variable, for a fast search.

If the index queue size is ≥ 1 , a match was found. Now do a full `compare()` which is slower, but checks all fields. Print the appropriate message, and delete the entry from `pkt_list`.

If no matching packet was found, tell the user.

In this example, a mismatch only generates a warning. Your testbench may want to generate error messages instead.

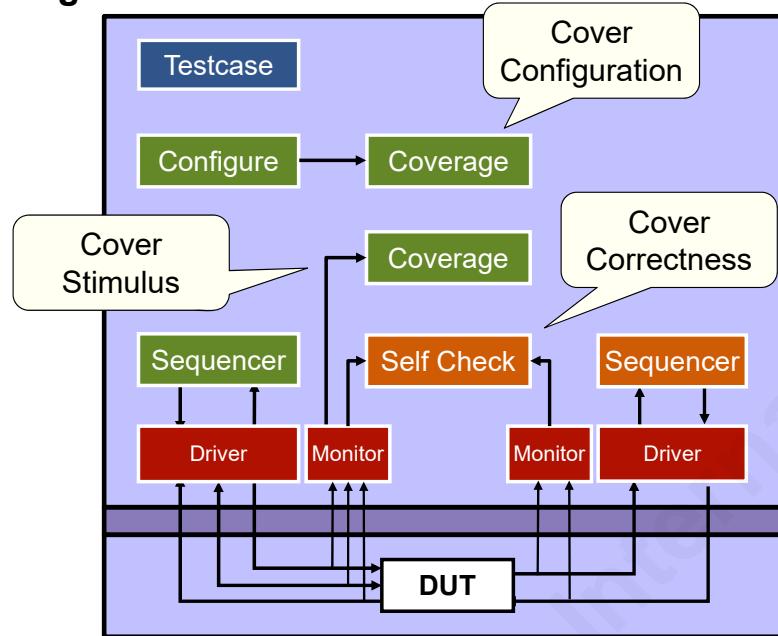
Functional Coverage

- **Measure the random stimulus to track progress towards verification goal**
- **What to measure?**
 - Configuration: Has testbench tried all legal environment possibilities?
 - ◆ N drivers, M Slaves, bus addresses, etc.
 - Stimulus: Has testbench generated all representative transactions, including errors?
 - ◆ Reads, writes, interrupts, long packets, short bursts, overlapping operations
 - Correctness: Has DUT responded correctly to the stimulus?
 - ◆ Reads, writes, interrupts, long packets, short bursts, overlapping operations

7-13

Connecting Coverage to Testbench

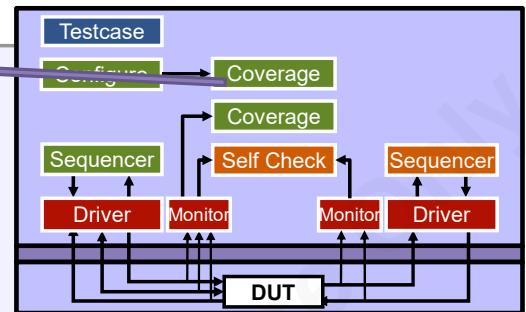
■ SystemVerilog Testbench Structure



7-14

Component Configuration Coverage (1/2)

```
covergroup cfg_cg() with function sample(env_cfg cfg);  
  ...  
endgroup  
class config_coverage extends uvm_component;  
  bit coverage_enable = 0;  
  env_cfg cfg; cfg_cg cg;  
  virtual function void build_phase(uvm_phase phase);  
    super.build_phase(phase);  
    uvm_config_db#(int)::get(this, "", "coverage_enable", coverage_enable);  
    if (coverage_enable) begin  
      uvm_config_db #(env_cfg)::get(this, "", "cfg", cfg);  
      cg = new();  
    end  
  endfunction  
  virtual function void start_of_simulation_phase(uvm_phase phase);  
    if (coverage_enable)  
      cg.sample(cfg);  
  endfunction  
endclass
```



Sample method called in
start_of_simulation_phase

7-15

Here is the cover group for this test. The sample function and argument num_of_active_ports are arraigned so the value can be sampled during simulation.

```
covergroup cfg_cg() with function sample(router_cfg cfg);  
  coverpoint cfg.num_of_active_ports;  
endgroup
```

Define the covergroup outside the class. The LRM requires that an embedded covergroup must be constructed in the class's constructor. The problem is that configuration variables such as *coverage_enable* and *cfg* are not assigned until the build phase, so you will need to call *uvm_config_db#(...)::get(...)* manually inside *new()*. Stick with non-embedded groups and the build phase.

The example above omitted the following declarations: utility macros, and constructor:

```
`uvm_component_utils(config_coverage)  
  
function new(string name, uvm_component parent);  
  super.new(name, parent);  
endfunction
```

Component Configuration Coverage (2/2)

■ Build configuration coverage component in test

```
class test_ports extends test_base;
    env_cfg cfg;
    config_coverage cfg_cov;
    virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        cfg = env_cfg::type_id::create("cfg", this);
        if (!cfg.randomize()) begin
            `uvm_fatal(...);
        end
        cfg_cov = config_coverage::type_id::create("cfg_cov", this);
        uvm_config_db #(env_cfg)::set(this, "env", "cfg", cfg);
        uvm_config_db #(env_cfg)::set(this, "cfg_cov", "cfg", cfg);
        uvm_config_db #(int)::set(this, "cfg_cov", "coverage_enable", 1);
    endfunction
endclass
```

Control coverage in test

7-16

The example above omitted the following declarations: utility macros, and constructor:

```
`uvm_component_utils(test_ports)

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

Stimulus Coverage

```
covergroup pkt_cg with function sample(packet pkt);  
  coverpoint pkt.sa;          Class with built-in analysis port  
endgroup  
class packet_coverage extends uvm_subscriber #(packet);  
  pkt_cg cov; bit coverage_enable;  
  virtual function void build_phase(uvm_phase phase); .  
    if (coverage_enable) cov = new();  
  endfunction  
  virtual function void write(T t);  
    if (coverage_enable) cov.sample(t);  
  endfunction  
endclass  
  
class test_stimulus_coverage extends test_base; ...;  
  packet_coverage cov_comp;  
  virtual function void build_phase(uvm_phase phase); ...;  
    cov_comp = packet_coverage::type_id::create("cov_comp", this);  
  endfunction  
  virtual function void connect_phase(uvm_phase phase); ...;  
    env.agt.analysis_port.connect(cov_comp.analysis_export);  
  endfunction  
endclass
```

Diagram illustrating the UVM coverage architecture:

- Testcase**: Contains **Configure**, **Coverage**, **Sequencer**, **Driver**, **Monitor**, and **DUT**.
- Configure** connects to **Coverage**.
- Sequencer** connects to **Driver** and **Monitor**.
- Driver** and **Monitor** both connect to the **DUT**.
- Coverage** connects to **Self Check**.
- Self Check** connects to **Sequencer**.
- Sample method called with monitored packet**: A callout points to the `cov.sample(t)` line in the code.
- Connect analysis ports**: A callout points to the `env.agt.analysis_port.connect(cov_comp.analysis_export);` line in the code.

7-17

The example above omitted the following declarations: utility macros, and constructor:

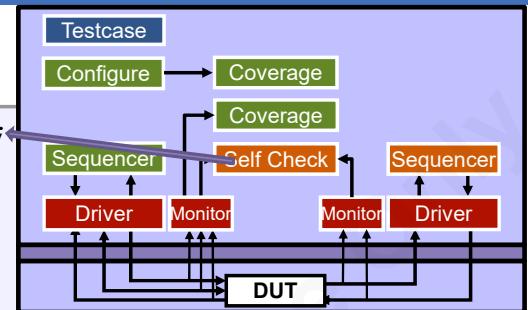
```
`uvm_component_utils(packet_coverage)  
  
function new(string name, uvm_component parent);  
  super.new(name, parent);  
endfunction  
  
`uvm_component_utils(test_stimulus)  
  
function new(string name, uvm_component parent);  
  super.new(name, parent);  
endfunction
```

Correctness Coverage

■ Cover verified transaction in scoreboard

```
covergroup sb_pkt_cg with function sample(packet pkt);  
  coverpoint pkt.sa;  
  coverpoint pkt.da;  
  cross pkt.sa, pkt.da;  
endgroup
```

```
class scoreboard #(type T = packet) extends scoreboard_base;  
  // component_utils and other code not shown  
  bit coverage_enable = 0;  
  virtual function void write_after(T pkt);  
    if (pkt.compare(pkt_ref)) begin  
      m_matches++;  
      if (coverage_enable) sb_pkt_cg.sample(pkt_ref);  
    end else begin  
      m_mismatches++;  
    end  
  endfunction  
endclass
```



Sample transaction once
match is confirmed

7-18

Unit Objectives Review

You should now be able to:

- Build re-usable self checking scoreboards by using the in-built UVM comparator classes
- Implement functional coverage



7-19

Appendix

Multi-Stream Scoreboard

7-20

Scoreboard: Multi-Stream

```
class ms_scoreboard extends uvm_scoreboard;
  uvm_analysis_imp_before #(packet, scoreboard) before_export;
  uvm_analysis_imp_after #(packet, scoreboard) after_export;
  typedef uvm_in_order_class_comparator #(packet) cmpr_t;
  cmpr_t cmpr[16];
  `uvm_component_utils(ms_scoreboard)
  function new(string name, uvm_component parent);
    super.new(name, parent);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
  endfunction
  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    before_export = new("before_export", this);
    after_export = new("after_export", this);
    for (int i=0; i < 16; i++) begin
      cmpr[i] = cmpr_t::type_id::create($sformatf("cmpr_%0d", i), this);
    end
  endfunction
  ... // Continued on next page
```

7-21

Scoreboard: Multi-Stream

```
virtual function void write_before(packet pkt);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    cmp[packet].before_export.write(pkt);
endfunction

virtual function void write_after(packet pkt);
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    cmp[packet].after_export.write(pkt);
endfunction

virtual function void report();
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    foreach (cmp[i]) begin
        `uvm_info("Scoreboard_Report",
            $sformatf("Comparator[%0d] Matches = %0d, Mismatches = %0d",
            i, cmp[i].m_matches, cmp[i].m_mismatches), UVM_MEDIUM);
    end
endfunction

endclass
```

7-22

Agenda

DAY

2

5 UVM Configuration & Factory

6 UVM Component Communication



7 UVM Scoreboard & Coverage

8 UVM Callback



Unit Objectives



After completing this unit, you should be able to:

- Embed UVM callback methods
- Build *façade* UVM callback classes
- Implement UVM callback to inject errors
- Implement UVM callback to implement coverage

8-2

Changing Behavior of Components

- How to enable adding/modifying operation of a component?
- One method: embed simple callbacks

```
class driver extends uvm_driver #(packet);  
  // utils macro and constructor not shown  
  virtual task run_phase(uvm_phase phase);  
    forever begin  
      seq_item_port.get_next_item(req);  
      pre_send(req); // simple callback  
      send(req);  
      post_send(req); // simple callback  
      seq_item_port.item_done();  
    end  
  endtask  
  virtual task send(packet tr); ...; endtask  
  virtual task pre_send(packet tr); endtask // required for simple callback  
  virtual task post_send(packet tr); endtask // required for simple callback  
endclass
```

Embed simple no-op methods
before and after major operation

Simple callback method are no-op methods of the class

8-3

The example above omitted the following declarations: utility macros, and constructor:

```
`uvm_component_utils(driver)  
  
function new(string name, uvm_component parent);  
  super.new(name, parent);  
endfunction
```

Implementing Simple Callback Operations

- Simple callback requires one to extend from existing component class

```
class driver_new extends driver;
    virtual task pre_send(...); ...
    virtual task post_send(...); ...
endclass
```

In the derived class, one can implement the callback methods

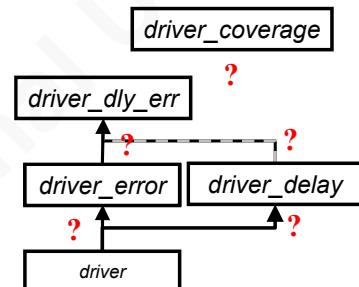
- This works well for making same change for all tests

- But, causes problems for testcase only changes

- Multiple extensions can cause unstable OOP hierarchy
 - ◆ How many versions of drivers to maintain?
 - ◆ How to add multiple extensions?

```
class driver_error extends driver;
class driver_delay extends driver;
class driver_dly_err extends driver_error;
```

What to extend from if multiple requirements for a test?



8-4

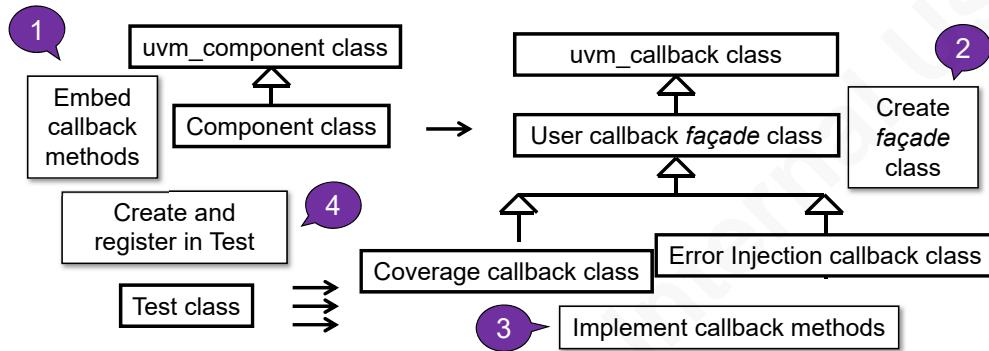
You can add new behavior to an existing component by extending the class. For example, **driver** can be extended to inject errors by extending the class to create **driver_error** that has a new **send()** task.

But, what if you want to make further modifications, such as adding a delay to the transmission? If your new class, **driver_delay**, extends the base driver, you won't be able to write a test that injects both delays and errors. If instead your new **driver_delay** class extends from the **driver_error** class, then all delay tests will also have to accommodate the error injection.

As your test needs grows, this type of callback implementation will lead to an unmanageable explosion of driver classes. There is a better way. UVM provides an alternative technique of callbacks, where each class is independent and thus can be combined in any combination.

Implementing UVM Callbacks

- Use UVM callbacks to add flexibility, without creating huge OOP hierarchy
- Four steps:
 - Embed UVM callback methods in components
 - Create a façade UVM callback class
 - Develop UVM callback classes extending from façade callback class
 - Create and register UVM callback objects in environment



8-5

A *façade* class has empty virtual methods. Thus the class's default behavior is to do nothing. You can create a class that does real work by extending these methods with ones that have bodies.

This is similar the architectural façade on saloons in the US Old West that had a 2-story front on a 1-story building. The building could later be extended and the front would not have to be changed.

Step 1: Embed Callback Methods

- Typically before and/or after major operation

```
class driver extends uvm_driver #(packet);
  `uvm_register_cb(driver, driver_callback)
  // utils macro and constructor not shown
  virtual task run_phase(uvm_phase phase);
    forever begin
      seq_item_port.get_next_item(req);

      `uvm_do_callbacks(driver, driver_callback, pre_send(this, req));
      send(req);

      `uvm_do_callbacks(driver, driver_callback, post_send(this, req));
      seq_item_port.item_done();
    end
  endtask
endclass
```

1a. Register UVM callback with component

1b. Embed UVM callback methods with **uvm_do_callbacks** macro

Component class name

UVM callback class name
User must create (see next slide)

UVM callback method
User must embed in callback class (see next slide)

8-6

The example above omitted the following declarations: utility macros, and constructor:

```
`uvm_component_utils(iMonitor)

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

Step 2: Declare the façade Class

■ Create façade class called in `uvm_do_callbacks` macro

- Typically declared in same file as the component
- All methods must be declared as virtual
- Leave the body of methods empty

2. Create callback façade class

```
typedef class driver;
class driver_callback extends uvm_callback; // utils macro not needed
  function new(string name = "driver_callback");
    super.new(name);
  endfunction
  virtual task pre_send(driver drv, packet tr); endtask
  virtual task post_send(driver drv, packet tr); endtask
endclass
```

Empty body: noop

Argument types must match types
in `uvm_do_callbacks` macro

Step 3: Implement Callback: Error

- Create error class by extending from façade class
 - Embed error in callback method

3. Implement error callback

```
class driver_err_callback extends driver_callback;
    function new(string name = "driver_err_callback");
        super.new(name);
    endfunction
    virtual task pre_send(driver drv, packet tr);
        tr.payload.delete();
    endtask
endclass
```

8-8

Step 4: Create and Register Callback Objects

- Instantiate the callback object in test
- Construct and register callback object

```
class test_driver_err extends test_base;
  // utils macro and constructor not shown
  driver_err_callback drv_err_cb;           4a. Create callback objects

  virtual function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
   drv_err_cb = new();
    uvm_callbacks #(driver, driver_callback)::add(env.agt.drv, drv_err_cb);
    uvm_callbacks #(driver, driver_callback)::display();
  endfunction
endclass
```

4b. Register callback objects

4c. Visually verify the registration (optional)

8-9

The example above omitted the following declarations: utility macros, and constructor:

```
`uvm_component_utils(test_driver_err)

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

Driver Coverage Example

■ If there are no analysis ports in driver

- Callbacks can be the hooks for coverage also

```
class driver_callback extends uvm_callback;
    virtual task pre_send(driver drv, packet tr); endtask
    virtual task post_send(driver drv, packet tr); endtask
endclass
class driver extends uvm_driver #(packet);
    `uvm_register_cb(driver, driver_callback)
    virtual task run_phase(uvm_phase phase);
        forever begin
            seq_item_port.get_next_item(req);
            `uvm_do_callbacks(driver, driver_callback, pre_send(this, req));
            send(req);
            `uvm_do_callbacks(driver, driver_callback, post_send(this, req));
            seq_item_port.item_done();
        end
    endtask
endclass
```

8-10

The example above omitted the following declarations for driver_callback: constructor:

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

The example above omitted the following declarations for driver: utility macros, and constructor:

```
`uvm_component_utils(driver)

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

Implement Coverage via Callback

■ Create coverage class by extending from façade class

- Define covergroup in coverage class
- Construct covergroup in class constructor
- Sample coverage in callback method

3a. Extend *façade* class

```
class driver_cov_callback extends driver_callback;
  covergroup drv_cov with function sample(packet pkt);
    coverpoint pkt.sa; coverpoint pkt.da;
    cross pkt.sa, pkt.da;
  endgroup
  function new(string name = "driver_cov_callback");
    super.new(name);
    drv_cov = new();
  endfunction
  virtual task post_send(driver drv, packet tr);
    drv_cov.sample(tr);
  endtask
endclass
```

3b. Implement coverage

8-11

Create and Register Callback Objects

- Instantiate the callback in Environment
- Construct and register callback object in connect phase

```
class test_driver_cov extends test_base;
  // utils macro and constructor not shown
  driver_cov_callback drv_cov_cb;           4a. Create callback objects
  virtual function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
   drv_cov_cb = new();
    uvm_callbacks #(driver, driver_callback)::add(env.agt.drv, drv_cov_cb);
  // uvm_callbacks #(driver, driver_callback)::add(null, drv_cov_cb);
  endfunction
endclass
```

Alternative: Register callback objects to all driver objects

```
4b. Register callback objects
```

8-12

The example above omitted the following declarations: utility macros, and constructor:

```
`uvm_component_utils(test_driver_cov)

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

Sequence Simple Callback Methods

- **uvm_sequence::pre_start() (task)**
 - called at the beginning of start() execution
- **uvm_sequence::pre_body() (task)**
 - Called before sequence body execution
- **uvm_sequence::pre_do() (task)**
 - called after sequencer::wait_for_grant() call and after sequencer has selected this sequence, but before the item is randomized
- **uvm_sequence::mid_do() (function)**
 - called after sequence item randomized, but before it is sent to driver
- **uvm_sequence::post_do() (function)**
 - called after the driver indicates item completion, using item_done/put
- **uvm_sequence::post_body() (task)**
 - Called after sequence body execution
- **uvm_sequence::post_start() (task)**
 - called at the end of start() execution

User should not call these methods directly. Instead, override in sequence definition

8-13

To avoid confusion in implementation, the use of pre_body() and post_body() methods should be avoided.

Unit Objectives Review

You should now be able to:

- Embed UVM callback methods
- Build *façade* UVM callback classes
- Implement UVM callback to inject errors
- Implement UVM callback to implement coverage



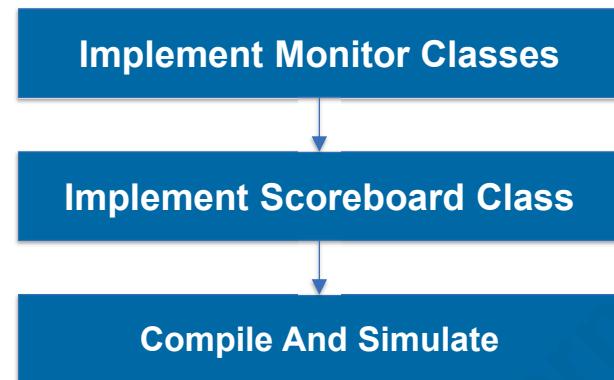
8-14

Lab 4: Full Testbench with Monitors and Scoreboard



60 minutes

- Implement monitors and scoreboard



8-15

This page was intentionally left blank

Agenda

DAY

3

9 UVM Advanced Sequence/Sequencer

10 UVM Phasing and Objections



11 UVM Register Abstraction Layer (RAL)

12 Summary



Unit Objectives



After completing this unit, you should be able to:

- Control execution order of sequences within a phase with a Top Sequence
- Manage synchronization of concurrent sequence executions within a phase with uvm_event

9-2

Managing Sequence Execution

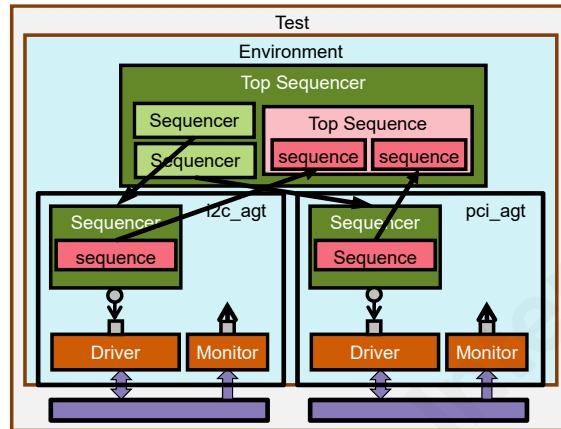
- **How can one coordinate sequences running across multiple agents?**
 - e.g. Reset of one part of the DUT must be done before another part of the DUT can go through a reset sequence
- **Solution: Top Sequence (Sequence Execution Manager)**
 - Explicitly manage the execution of sequences across multiple agents within a phase
 - Allows fine grained control of the sequence execution order
 - Doesn't have its own sequence item, only used to manage the execution of other sequences
 - Resources required by the sequence execution manager resides in a support sequencer

9-3

There is another common name used to described these classes: virtual sequence and virtual sequencer. However, the Accellera committee has found these names to be confusing to users. So, the recommendation from the committee is to no longer call them "virtual" sequences and sequencers. In this unit, we will calling them top sequences and top sequencers.

Managing Sequence Execution

- Typically, a sequence only interacts with a single agent to manage the activities of a single DUT interface
- When multiple DUT interfaces need to be synchronized, a upper layer sequence and sequencer are required

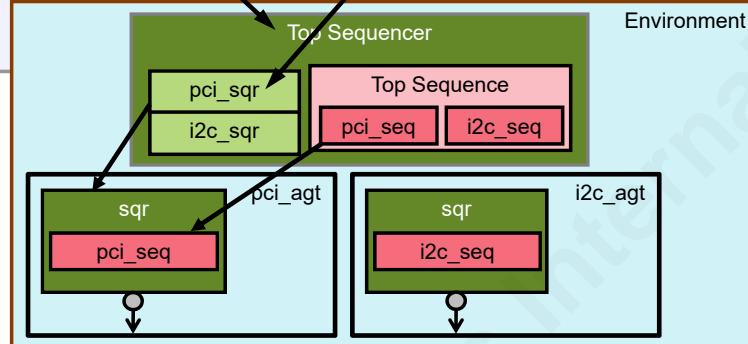


9-4

Top Sequence

- Embed sequences to be managed
- In body() method, manage these sequence execution

```
class top_sequence extends uvm_sequence;
  `uvm_declare_p_sequencer(top_sequencer)
  pci_sequence pci_seq;
  i2c_sequence i2c_seq;
  virtual task body();
    `uvm_do_on(pci_seq, p_sequencer.pci_sqr);
    `uvm_do_on(i2c_seq, p_sequencer.i2c_sqr);
  endtask
endclass
```



Derived from
uvm_sequence class

Macro creates a handle called
p_sequencer to access
top sequencer content

See note

9-5

The top sequence class needs to have access to sequencer handles it needs to control. There are two ways to accomplish this.

One – Create a top sequencer where the handles of the sequencers to be managed resides. This is shown above and the next slide. This is the most common approach. (Can be parameterized)

Two – Pass the handles of sequencer to be controlled via `uvm_config_db`. This approach is more flexible in that no sequencer class is needed to support the execution of the top sequence. The downside to this approach is that it is harder to debug and can be confusing for someone who is not entirely comfortable with UVM. (Can also be parameterized)

Recommendation: Choose one approach and stick with that approach for the project. Do not implement both approaches within the same project.

Change in IEEE UVM: `~uvm_do` is expanded to support sequencer, priority and constraint

(`~uvm_do_on` is not needed nor supported with IEEE UVM):

```
~uvm_do(SEQ_OR_ITEM, SEQR=get_sequencer(), PRIORITY=-1, CONSTRAINTS={})
```

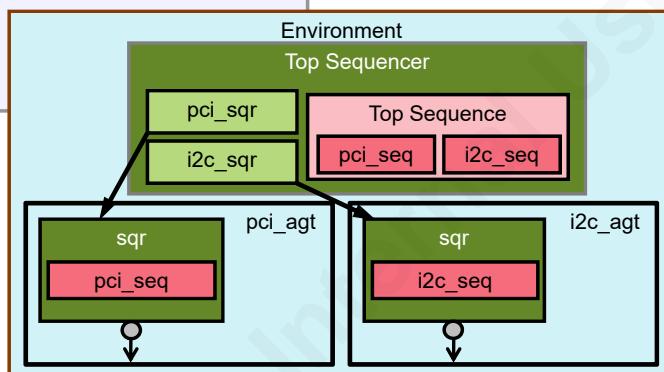
The following code is left off the slide:

```
~uvm_object_utils(top_sequence)
function new(string name = "top_sequence");
  super.new(name);
endfunction
```

Top Sequencer

- Not associated with any agent
 - Contains resources needed by the top Sequence
 - Example: sequencer handles

```
class top_sequencer extends uvm_sequencer;
  `uvm_component_utils(top_sequencer)
  // Constructor not shown
  pci_sequencer pci_sqr;
  i2c_sequencer i2c_sqr;
endclass
```



9-6

The top_sequencer acts as a service mechanism for the execution of top sequence it's tied to. In this case, the top sequencer contains handles to the sequencers needed by the top sequence.

For other examples of how the sequencer can serve as a service mechanism to support sequence execution, please take a look at the appendix.

The following is the constructor for the top sequencer class:

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

Executing Top Sequence

- Create the top sequencer in build phase
- Set managed sequencer's "default_sequence" to null
- Configure the top sequencer to execute the top sequence

```
class test_new extends test_base; // other code not shown
    top_sequencer top_sqr;
    virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        // creation of other objects not shown
        top_sqr = top_sequencer::type_id::create("top_sqr", this);
        uvm_config_db#(uvm_object_wrapper)::set(this,
            "env.pci_agt.sqr.main_phase", "default_sequence", null);
        uvm_config_db#(uvm_object_wrapper)::set(this,
            "env.i2c_agt.sqr.main_phase", "default_sequence", null);
        uvm_config_db#(uvm_object_wrapper)::set(this, "top_sqr.main_phase",
            "default_sequence", top_sequence::get_type());
    endfunction
// continued on the next page
```

Disable managed sequencers

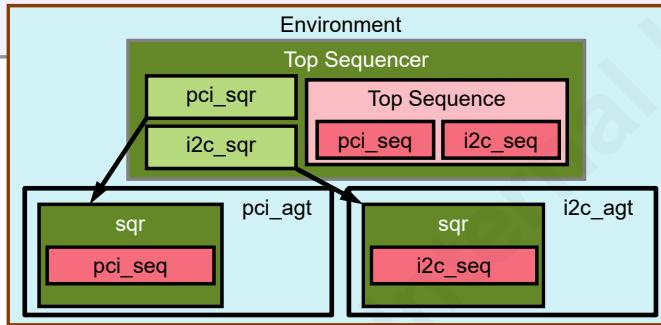
Set top sequencer to execute top sequence

9-7

Set Top Sequencer Content

- In connect phase, set top sequencer's sequencer handles to the sequencers to be managed

```
// Continued from previous page
virtual function void connect_phase(uvm_phase phase);
    top_sqr.pci_sqr = env.pci_agt.sqr;
    top_sqr.i2c_sqr = env.i2c_agt.sqr
endfunction
endclass
```



9-8

In the connect phase, point the sequencer handles in the top sequencer to the actual location of the sequencers in the environment.

Sequence Execution Management

- What if sequence needs to start in middle of another sequence?

```
class top_sequence extends uvm_sequence;
  // other code not shown
  virtual task body();
    fork
      `uvm_do_on(pci_seq, p_sequencer.pci_sqr)
      begin
        wait(...); // wait for some event before processing
        `uvm_do_on(i2c_seq, p_sequencer.i2c_sqr)
      end
    join
  endtask
endclass
```

- Instead of SystemVerilog wait, use the more powerful `uvm_event` via `uvm_pool`

9-9

Synchronization Mechanism: uvm_event

■ Wait for one trigger to releases all waits

- Emulates SV event

Triggers uvm_event

UVM-1.1

```
class uvm_event extends uvm_object;
    virtual function void trigger(uvm_object data=null); // data optional
    virtual function void reset(bit wakeup=0);
    virtual function bit is_on();
    virtual function bit is_off();
    virtual task wait_on(bit delta=0);
    virtual task wait_off(bit delta=0);
    virtual task wait_trigger(); // equivalent to @(event);
    virtual task wait_trigger_data(output uvm_object data); // with data
    virtual task wait_ptrigger(); // equivalent to wait(event.triggered)
    virtual task wait_ptrigger_data(output uvm_object data); // with data
    virtual function void cancel(); Cancels wait
endclass
```

After trigger, uvm_event stays on until reset is called

Query state of uvm_event

Level sensitive wait for state. Does not block if true.

Edge sensitive wait

9-10

There are two distinction operational modes with the uvm_event class: edge sensitive and level sensitive.

Edge Sensitive:

For this mode, the methods involved are: **trigger()**, **wait_trigger()**, **wait_ptrigger()** and the **_data()** methods. Do NOT use **reset()**, **is_on()**, **is_off()**, **wait_on()** and **wait_off()** methods.

The **wait_*** methods will block and released when the **trigger()** is called.

Level Sensitive:

For this mode, the methods involved are: **trigger()**, **reset()**, **is_on()**, **is_off()**, **wait_on()**, and **wait_off()**. Do NOT use any of the **wait_*trigger** methods!

The **trigger()** method sets the event state to ON and releases all existing **wait_trigger()** and **wait_on()** for the event.

The **reset()** method sets the event state to OFF and releases all **wait_off()** for the event. If used for edge sensitive wait, the **reset()** call will disarm the trigger afterwards! You should NOT call **reset()** if the event is used as an edge sensitive synchronization mechanism!

Synchronization Mechanism: uvm_event

- Essentially the same, except allows user to specify default transaction type

```
class uvm_event_base extends uvm_object;
    virtual function void reset(bit wakeup=0);
    virtual function bit is_on();
    virtual function bit is_off();
    virtual task wait_on(bit delta=0);
    virtual task wait_off(bit delta=0);
    virtual task wait_trigger(); // equivalent to @(event);
    virtual task wait_ptrigger(); // equivalent to wait(event.triggered)
    virtual function void cancel();
endclass
```

UVM-1.2 &
IEEE UVM



```
class uvm_event #(type T=uvm_object) extends uvm_event_base;
    virtual function void trigger(T data=null);
    virtual task wait_trigger_data(output T data);
    virtual task wait_ptrigger_data(output T data);
    virtual function T get_trigger_data();
    virtual function void cancel();
endclass
```

9-11

Specialized Pools for Synchronization

- Simplify usage of `uvm_event` and `uvm_barrier` through two specialized resource pools:

```
typedef uvm_object_string_pool #(uvm_event) uvm_event_pool;
typedef uvm_object_string_pool #(uvm_event#(uvm_object)) uvm_event_pool;
```

UVM-1.1

UVM-1.2

See note for IEEE



- With pools, events are globally accessible

- Commonly used members of the `uvm_object_string_pool` class:

```
class uvm_object_string_pool#(type T=uvm_object) extends uvm_pool#(string,T);
    typedef uvm_object_string_pool #(T) this_type;
    function new (string name="");
        static function T get_global (string key);
        virtual function T get (string key);
        virtual function void delete (string key);
    endclass
```

class is deprecated in IEEE
please see note

9-12

get methods create keyed resource
if it does not already exist

In IEEE UVM, the `uvm_object_string_pool` class will be deprecated.



The event pool will become a class: (Note: these changes do not impact user code for the next few slides!)

```
class uvm_event_pool extends uvm_pool#(string,uvm_event#(uvm_object));
    function new (string name="");
        static function uvm_event_pool get_global_pool();
        static function uvm_event get_global (string key);
        virtual function uvm_event get (string key);
        virtual function void delete (string key);
    endclass
```

Trigger Global Reset Event (Level Sensitive)

■ Triggering a globally accessible reset event

```
class reset_monitor extends uvm_monitor; // other code left off
  uvm_event reset_event = uvm_event_pool::get_global("reset");
  virtual task run_phase(uvm_phase phase);
    forever begin
      reset_tr tr = reset_tr::type_id::create("tr", this);
      observe(tr);
    end
  endtask: run_phase
  virtual task observe(reset_tr tr);
    @(vif.reset_n);
    if (vif.reset_n == 1'b0) begin
      tr.kind = reset_tr::ASSERT;
      reset_event.trigger();           Set "reset" uvm_event to on state
    end else begin
      tr.kind = reset_tr::DEASSERT;
      reset_event.reset();           Set "reset" uvm_event to off state
    end
  endtask
endclass
```

Get "reset" uvm_event from **uvm_event_pool**

Set "reset" uvm_event to on state

Set "reset" uvm_event to off state

Caution see note !

9-13

You must choose to use uvm_event as either a level sensitive or edge sensitive event. You must NOT mix the two usage model.

For level sensitive synchronization, use **trigger()** to turn on the event. Use **reset()** to turn off the event.

For edge sensitive synchronization, only use **trigger()** to trigger the edge. Do NOT call the **reset()** method.

Calling **reset()** method while using the event as an edge sensitive synchronization will disarm the trigger mechanism. The consequence is that afterwards, the event can no longer be released by any **trigger()** call. Most likely causing a locked thread.

uvm_event_pool Example (Level Sensitive)

```
class top_reset_sequence extends uvm_sequence; // other code left off
`uvm_declare_p_sequencer(top_reset_sequencer)
uvm_event reset_event = uvm_event_pool::get_global("reset");
reset_sequence r_seq;
router_input_port_reset_sequence i_seq;
virtual task body();
    fork
        `uvm_do_on(r_seq, p_sequencer.r_sqr);
        foreach (p_sequencer(pkt_sqr[i]) begin
            int j = i;
            fork
                begin
                    reset_event.wait_on(); // Wait for "reset" uvm_event to be on
                    `uvm_do_on(i_seq, p_sequencer(pkt_sqr[j]));
                end
            join_none // Execute response to the reset event
        end
    join
endtask
endclass
```

Get "reset" **uvm_event** from **uvm_event_pool**

Execute reset sequence

Wait for "reset" **uvm_event** to be on

Execute response to the reset event

9-14

Exclusive Sequencer Access (lock/unlock)

■ Sequence can request exclusive access to sequencer

- Place sequence request behind other requests already in the queue
- Once granted, stays at top of the request queue blocking other sequence access requests

```
class simple_sequence extends uvm_sequence #(packet);  
  // utils macro and constructor not shown  
  virtual task body();  
    lock();  
    repeat(10) begin  
      `uvm_do_with(req, {sa == 6; da == 6;});  
    end  
    unlock();  
  endtask  
endclass
```

lock(); When granted, sequence has exclusive access to sequencer

unlock(); Removes sequence from top of sequencer queue

Exclusive Sequencer Access (grab/ungrab)

■ Sequence can request for exclusive access to sequencer

- Placed at the top of the sequence request queue
 - ◆ Granted access when no other grabs or locks are currently blocking access
- Has exclusive access to sequencer until explicitly released
- Typically needed for high priority interrupt sequence execution

```
class simple_sequence extends uvm_sequence #(packet);
    // utils macro and constructor not shown
    virtual task body();
        grab();           // Sequence requesting grab()
        repeat(10) begin // reserves sequencer for exclusive use
            `uvm_do_with(req, {sa == 6; da == 6});
        end
        ungrab();         // Removes sequence from
    endtask             // top of sequencer queue
endclass
```

9-16

Unit Objectives Review

You should now be able to:

- Control execution order of sequences within a phase with a Top Sequence
- Manage synchronization of concurrent sequence executions within a phase with uvm_event



9-17

Appendix

Sequence Arbitration & Priority

Reactive Sequence

Interrupt Sequence

`uvm_barrier`

9-18

Sequence Arbitration and Priority (1/2)

- Once a sequence is started on a sequencer, it must be arbitrated for access to the sequencer resources
 - Such as the `seq_item_port`
- By default, all sequences have priority of 100
 - Child sequence defaults to that of the parent sequence
- The higher the value, the higher the priority
- The priority can be set via macro
 - ``uvm_do_pri(seq, pri)` // UVM-1.1 and UVM-1.2
 - ``uvm_do(seq,,pri)` // IEEE UVM
- This priority can change dynamically through the course of simulation
 - `seq.set_priority(300);`

9-19

Sequence Arbitration and Priority (2/2)

- **Sequencers use one of several arbitration schemes**
 - SEQ_ARB_FIFO - Requests are granted in FIFO order (default)
 - SEQ_ARB_WEIGHTED - Requests are granted randomly by weight
 - SEQ_ARB_RANDOM - Requests are granted randomly
 - SEQ_ARB_STRICT_FIFO - Requests at highest priority granted in fifo order
 - SEQ_ARB_STRICT_RANDOM - Requests at highest priority granted randomly
 - SEQ_ARB_USER Arbitration is delegated to the user-defined function,
`user_priority_arbitration()`
- **The default arbitration scheme (SEQ_ARB_FIFO) is unaffected by sequence priority**

9-20

Code Example

■ Set interrupt sequence to the highest priority

```
task my_sequence::body(uvm_phase phase);
    get_sequencer().set_arbitration(SEQ_ARB_STRICT_FIFO)
    fork
        `uvm_do(burst_seq);
        `uvm_do(noise_seq);
        `uvm_do_pri(inter_seq, 3000); // non-IEEE
    join
endtask

task my_test::run_phase(uvm_phase phase);
    phase.raise_objection(this);
    env.agt.sqr.set_arbitration(SEQ_ARB_STRICT_FIFO);
    fork
        burst_seq.start(env.agt.sqr);
        noise_seq.start(env.agt.sqr);
        inter_seq.start(env.agt.sqr, 3000); // all UVM version
    join
    phase.drop_objection(this);
endtask
```

Set the priority
when started

9-21

Appendix

Sequence Arbitration & Priority

Reactive Sequence

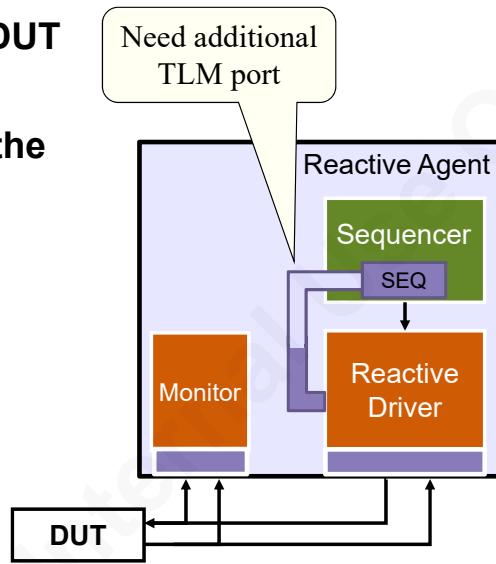
Interrupt Sequence

`uvm_barrier`

9-22

Reactive Sequences

- Activity is initiated by the reactive driver when it receives a transaction from the DUT
 - Not the Sequencer
- The Partial Transaction must be sent to the reactive sequence
- The sequence needs to formulate a response and send it back to the driver
- These requirements make reactive sequences different



9-23

Reactive TLM Port Setup

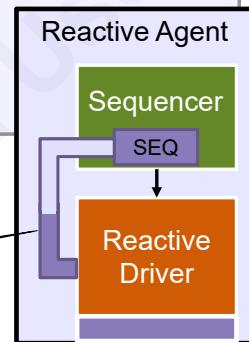
```
class reactive_driver extends uvm_driver#(trans);
  uvm_blocking_get_imp#(trans, reactive_driver) get_export;
  virtual task get(output trans tr);
    // Shown later
  endtask
endclass

class reactive_sqr extends uvm_sequence_base #(trans);
  uvm_blocking_get_port#(trans) get_port;
  virtual task wait_for_req(uvm_sequence_base seq, output trans req);
    wait_for_grant(seq);
    get_port.get(req);
  endtask
endclass

class reactive_agent extends uvm_agent;
  virtual function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    sqr.get_port.connect(drv.get_export);
  endfunction
endclass
```

Declare a TLM “get” port and associated get method

Connect reactive TLM port



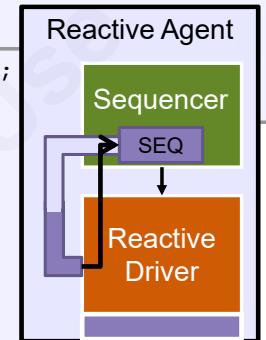
9-24

Reactive Sequence Request

```
class reactive_sequence extends uvm_sequence #(trans); // other code left off
`uvm_declare_p_sequencer(reactive_sequencer)
virtual task body();
  forever begin
    p_sequencer.wait_for_req(this, req); Wait for request from driver
    // ...
  end
endtask
endclass

class reactive_sequencer extends uvm_sequencer #(trans); // code left off
  virtual task wait_for_req(uvm_sequence_base seq, output trans req);
    wait_for_grant(seq);
    get_port.get(req); Get request from driver
  endtask
endclass

class reactive_driver extends uvm_driver#(trans);
  trans m_tr;
  virtual task get(output trans tr);
    wait(m_tr != null); tr = m_tr; m_tr = null;
  endtask
  virtual task run_phase(uvm_phase phase);
    forever begin
      get_tr(m_tr); Get DUT output
      // ... get and send response from sequence
    end
  endtask
endclass
```



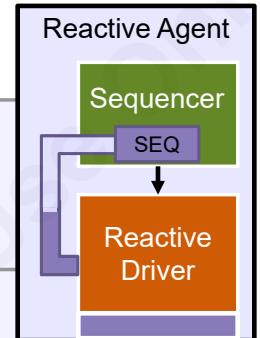
9-25

Reactive Sequence Response

```
class reactive_sequence extends uvm_sequence #(trans);
  `uvm_declare_p_sequencer(reactive_sequencer)
  virtual task body();
    forever begin
      p_sequencer.wait_for_req(this, req);
      // generate response to req
      p_sequencer.send_rsp(this, rsp); Give response to driver through sequencer
    end
  endtask
endclass

class reactive_sequencer extends uvm_sequencer #(trans);
  virtual task send_rsp(uvm_sequence_base seq, trans rsp);
    rsp.set_item_context(seq);
    seq.finish_item(rsp); Give response to driver
  endtask
endclass

class reactive_driver extends uvm_driver#(trans);
  forever begin
    get_tr(); // get DUT output
    seq_item_port.get_next_item(rsp);
    drive_rsp(rsp);
    seq_item_port.item_done();
  end
endclass
```



9-26

Appendix

Sequence Arbitration & Priority

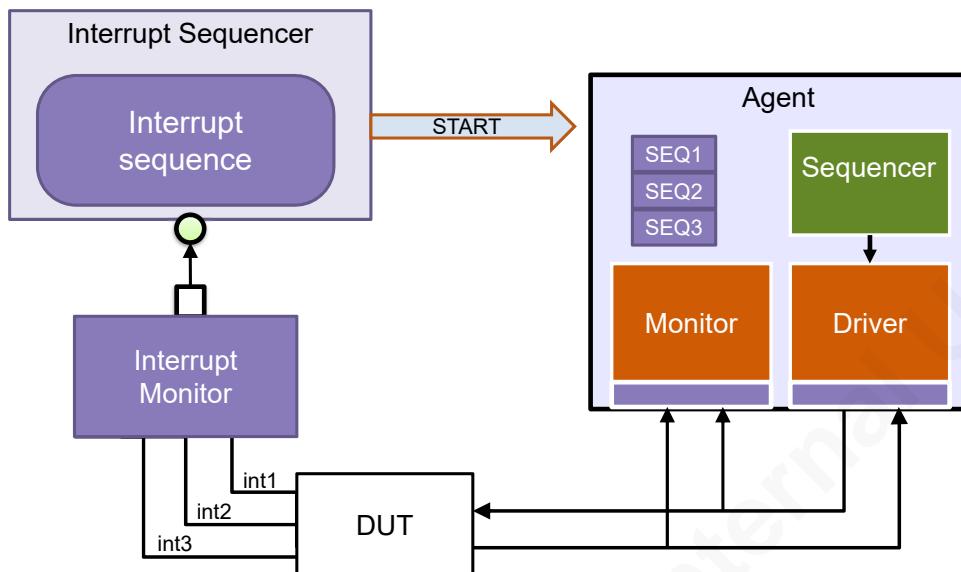
Reactive Sequence

Interrupt Sequence

`uvm_barrier`

9-27

Interrupt Sequences



9-28

Detect Interrupt

```
task intr_monitor::tx_monitor(); // simplified code
    intr_trans tr = intr_trans::type_id::create("tr", this);
    bit [3:0] interrupts = vif.monClk.interrupts;
    @(vif.monClk iff (vif.monClk.interrupts != interrupts));
    tr.interrupts = vif.mon.interrupts;
    analysis_port.write(tr);
endtask

class intr_sequencer extends uvm_sequencer;
    my_sequencer sqr;
    intr_trans tr; event intr_event;
    uvm_analysis_imp#(intr_trans,intr_sequencer) analysis_export;
    virtual function void write(intr_trans tr);
        if(tr.interrupts) begin // at least one interrupt set
            this.tr = tr;
            ->intr_event;
        end
    endfunction
endclass

function void environment::connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    intr_sqr.sqr = agt.sqr;
    intr_mon.analysis_port.connect(intr_sqr.analysis_export);
endfunction
```

Monitor interrupts and send out via analysis port

Needs sequencer to pass interrupt responses to driver

Connects to the intr_monitor's analysis port

9-29

React to Interrupt

```
class interrupt_sequence extends uvm_sequence; // other code left off
  intr_sequence_1 seq1;
  intr_sequence_2 seq2;
  intr_sequence_3 seq3;
  virtual task body();
    p_sequencer.set_arbitration(SEQ_ARB_STRICT_FIFO);
    forever begin
      @(p_sequencer.interrupt_event);
      case(p_sequencer.tr.interrupts)
        4'b0001: `uvm_do_on_pri(seq1, p_sequencer.sqr, 500); // non-IEEE
        4'b0010: `uvm_do_on(seq2, p_sequencer.sqr);           // non-IEEE
        4'b0100: `uvm_do_on(seq3, p_sequencer.sqr);           // non-IEEE
      endcase
    end
  endtask
endclass
```

Set arbitration

Wait for the interrupt event from the sequencer

Priority

```
function void intr_test::build_phase(uvm_phase phase);
  super.build_phase(phase);
  uvm_config_db#(uvm_object_wrapper)::set(this, "env.intr_sqr.run_phase",
                                             "default_sequence", interrupt_sequence::get_type());
endfunction
```

9-30

Appendix

Sequence Arbitration & Priority

Reactive Sequence

Interrupt Sequence

`uvm_barrier`

9-31

Component Synchronization: uvm_barrier

- Wait for number of waiters to reach threshold to release all waits
- Essential methods of the class:

```
class uvm_barrier extends uvm_object;
    function new (string name="", int threshold=0);
        virtual function void set_threshold (int threshold);

    virtual task wait_for();
        Sets threshold level (n number of waiters) to release all waits
    endtask

    virtual function void reset (bit wakeup=1);
        Caller blocked until threshold of waiters has been reached. Increments waiter count.
    endfunction

    virtual function void set_auto_reset (bit value=1);
        Reset numbers of waiters to 0. If wakeup set, release all current waiters.
    endfunction

    virtual function void cancel ();
        If value is 1, when threshold is hit, count resets to 0. If 0, count does not reset.
    endfunction

endclass
```

Cancels wait. Decrements waiter count.

9-32

Agenda

DAY

3

9 UVM Advanced Sequence/Sequencer

10 UVM Phasing and Objections



11 UVM Register Abstraction Layer (RAL)

12 Summary



Unit Objectives

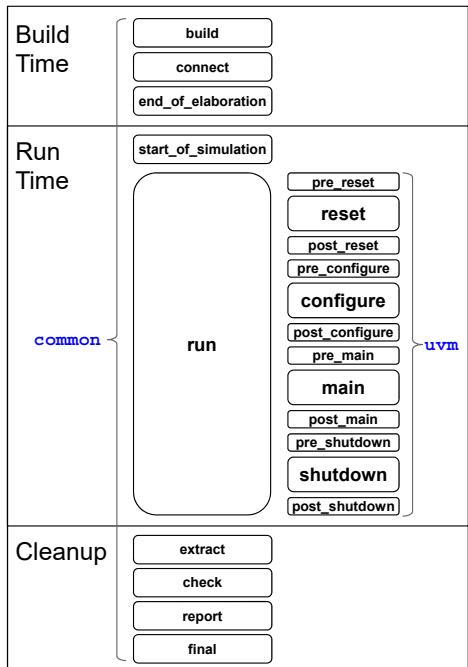


After completing this unit, you should be able to:

- Control phase objections
- Control phase timeout
- Create user phases
- Create phase domains
- Implement phase callbacks

10-2

UVM Component Phasing



- **Synchronized phase execution**
- **Two predefined domains**
 - **common** - Environment components (environment/agent/driver/monitor)
 - ◆ run
 - **uvm** - Sequence execution control (test only)
 - ◆ reset -> shutdown
 - With pre_*/post_* phases
- **Task phases terminate when objection count reaches zero**
- **Enough flexibility for basic to intermediate user**

10-3

Common Phases

build	Create and configure testbench components
connect	Establish cross-component connections
end_of_elaboration	Check for correctness of testbench structure
start_of_simulation	Print configuration for components
run	Stimulate the DUT
extract	Extract data from different points of the verification environment
check	Check for any unexpected conditions in the verification environment
report	Report results of the test
final	Tie up loose ends

For individual component type recommendation see appendix

10-4

Run-Time Task Phases

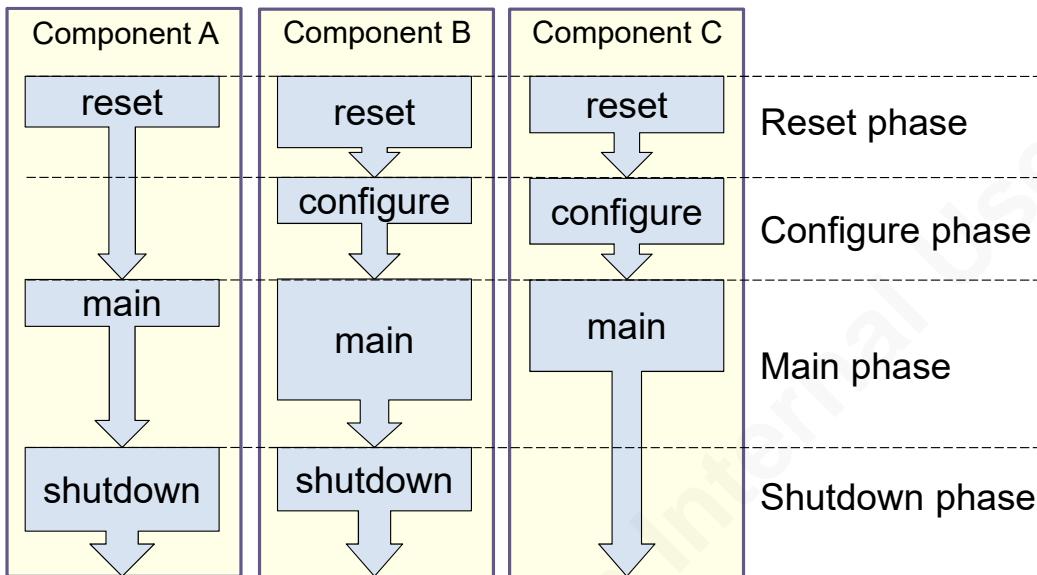
pre_reset	Setup/Wait for conditions to reset DUT
reset	Reset DUT/De-assert control signals
post_reset	Wait for DUT to be at a known state
pre_configure	Setup/Wait for conditions to configure DUT
configure	Configure the DUT
post_configure	Wait for DUT to be at a known configured state
pre_main	Setup/Wait for conditions to start testing DUT
main	Test DUT
post_main	Typically a no-op
pre_shutdown	Typically a no-op
shutdown	Wait for data in DUT to be drained
post_shutdown	Perform final checks that consume simulation time

For individual component type recommendation see appendix

10-5

Task Phase Synchronization

- Run-Time task phases for all components will move on to the next phase only when all objections for that phase are dropped



10-6

Objections are like students who raise their hand during lecture. Instructor cannot move on to next slide (phase) until all questions are answered (objections dropped).

Phase Objection – Device Drivers (1/6)

- Do not raise/drop objection in component's run_phase
 - Impacts simulation with excessive objection count

```
class driver extends ...;
  virtual task run_phase(uvm_phase phase);
    forever begin
      seq_item_port.get_next_item(req);
      phase.raise_objection(this);
      send(req);
      seq_item_port.item_done();
      phase.drop_objection(this);
    end
  endtask
endclass
```

```
class monitor extends ...;
  virtual task run_phase(uvm_phase phase);
    forever begin
      @(negedge vif.sig]);
      phase.raise_objection(this);
      get_packet(tr);
      phase.drop_objection(this);
      ...
    end
  endtask
endclass
```

10-7

It is generally not possible for the driver and monitor classes to be able to determine when a test is done. Therefore, it does not make sense for the drivers and the monitor to raise or drop objections in the run_phase. Doing so will only slow down simulation.

In general, a few objections by a component in a phase is fine. But, if all the drivers and monitors raise and drop objections for every transaction processed, performance will suffer. Use the necessary objections, and no more.

Phase Objection – Sequence (2/6)

- Objections can be raised/dropped in sequence

```
class packet_sequence ...; // other code not shown
    function packet_sequence::new(string name = "packet_sequence");
        super.new(name);
        set_automatic_phase_objection(1);           // UVM-1.2 & IEEE UVM Only!
    endfunction
    virtual task pre_start();
        if (get_parent_sequence() == null && starting_phase != null)
            starting_phase.raise_objection(this); // UVM-1.1 ONLY!
    endtask
    virtual task post_start();
        if (get_parent_sequence() == null && starting_phase != null)
            starting_phase.drop_objection(this); // UVM-1.1 ONLY!
    endtask
endclass
```

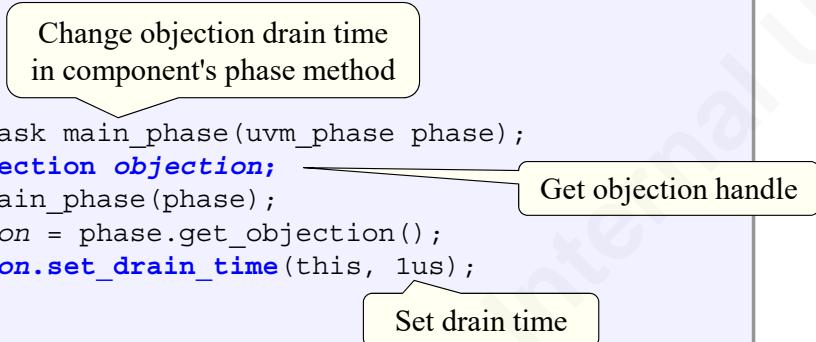
10-8

See Unit 4 – Creating Stimulus Sequences, for explanation of the differences between UVM-1.1, 1.2 and IEEE.

Phase Objection – Drain Time (3/6)

- **Objections in sequence may not be enough**
 - There may be latency within the DUT
- **Known latency can be taken care of with drain time**
 - `set_drain_time()` method extends the phase for the specified amount of simulation time after objection count reaches 0

```
class test_drain extends test_base; // other code not shown  
  
    Change objection drain time  
    in component's phase method  
  
    virtual task main_phase(uvm_phase phase);  
        uvm_objection objection;  
        super.main_phase(phase);  
        objection = phase.get_objection();  
        objection.set_drain_time(this, 1us);  
    endtask  
endclass
```



10-9

Phase Objection – Scoreboard (4/6)

- A common way of handling unknown latency of DUT is to wait for the scoreboard's expect queue to be empty

```
class scoreboard#(type T=packet) extends scoreboard_base; // other code left off  
  T expected_queue[$]; // queue of expected transactions  
  virtual task wait_for_empty();  
    wait (expected_queue.size() == 0);  
  endtask  
endclass  
  
class test_shutdown extends test_base; // other code left off  
  virtual task shutdown_phase(uvm_phase phase); // phase specific  
    uvm_component comps[$]; scoreboard_base sb;  
    phase.raise_objection(this, "Wait for scoreboard to empty");  
    uvm_root::get().find_all("*", comps);  
    foreach (comps[i]) begin  
      if ($cast(sb, comps[i])) begin  
        sb.wait_for_empty(); Wait for scoreboard  
queue to empty  
      end  
    end  
    phase.drop_objection(this, "Scoreboard queues emptied");  
  endtask  
endclass
```

10-10

Phase Objection – Test (5/6)

- Test can extend phase via phase callback

- `phase_ready_to_end()`
 - ◆ Called when all objections are dropped for a phase
- Requires the test writer to create and manage a `BUSY` flag

```
class my_test extends test_base;                                // other code left off
    virtual function void phase_ready_to_end(uvm_phase phase); // all phases
        // BUSY flag must be set and clear by test
        if (BUSY) begin
            phase.raise_objection(this, "In middle of protocol");
            fork
                begin
                    wait(BUSY == 0);
                    phase.drop_objection(this, "Completed protocol");
                end
            join_none
        end
    endfunction
endclass
```

10-11

The code demonstrates how to handle a `phase_ready_to_end` callback. It checks if a `BUSY` flag is set. If set, it raises an objection and enters a forked block. Inside the fork, it waits for the `BUSY` flag to clear, then drops the objection. The code ends with a `join_none` statement.

Phase Objection - Debug (6/6)

```
virtual function void phase_started(uvm_phase phase);
  if (uvm_report_enabled(UVM_FULL, UVM_INFO, "OBJECTORS")) begin
    if (phase.get_objection() == null) return; ————— Only debug if objection exists
    fork begin
      phase.wait_for_state(UVM_PHASE_EXECUTING); ————— Wait for component to have a
      fork ————— chance to raise objection
        uvm_phase this_phase = phase;
        uvm_objection objection = phase.get_objection();
        uvm_object objectors[$];
        forever begin
          objection.get_objectors(objectors); ————— Print objectors
          foreach (objectors[i]) $display("@%t - objectors[%0d]: %s",
                                         $realtime, i, objectors[i].get_full_name());
          #1us; // Objection sample period
        end
        begin this_phase.wait_for_state(UVM_PHASE_ENDED); end
      join_any
      disable fork;
    end join_none
  end
endfunction
```

For objection debugging using objection callbacks see appendix

10-12

Phase when executing goes through the following normal transition:

DORMANT -> SCHED -> SYNC -> START -> EXEC -> READY -> END -> CLEAN -> DONE

```
typedef enum {
  UVM_PHASE_DORMANT      = 1,
  UVM_PHASE_SCHEDULED    = 2,
  UVM_PHASE_SYNCING       = 4,
  UVM_PHASE_STARTED       = 8,
  UVM_PHASE_EXECUTING     = 16,
  UVM_PHASE_READY_TO_END = 32,
  UVM_PHASE_ENDED         = 64,
  UVM_PHASE_CLEANUP       = 128,
  UVM_PHASE_DONE           = 256,
  UVM_PHASE_JUMPING       = 512
} uvm_phase_state;
```

UVM Timeout

■ Run-Time switch:

- +UVM_TIMEOUT
- Maximum absolute time before fatal is called
- Defaults to 9,200 sec
 - ◆ Causes fatal message if reached



Caution: Not in IEEE P1800.2

■ Embed in test:

```
uvm_root::get().set_timeout(.timeout(1ms))
```

- Overridden by +UVM_TIMEOUT if called in new()
 - ◆ Recommended
- Overrides +UVM_TIMEOUT if called in phase method
 - ◆ Not recommended



Caution: Not in IEEE P1800.2

10-13

Advanced Features

■ Phase Domains

- UVM has two default phase domains: common and uvm
- Phases within same domain are synchronized
- Inter-domain phases can be synchronized
 - ◆ User can set full, partial or no synchronization

■ User Defined Phases

- Can be mixed with predefined phases

■ Phase Jumping

- Forwards and Backwards



Should only be implemented in test

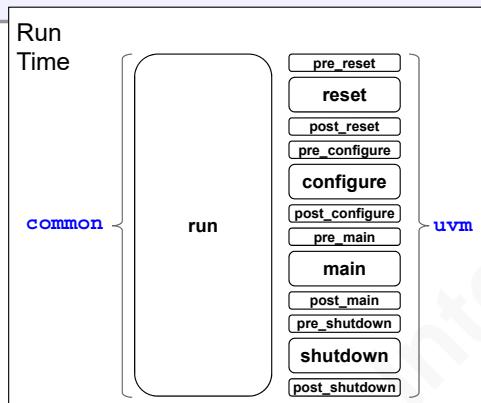
10-14

Phase Domains (1/2)

■ UVM phases are scheduled within UVM domains

- There are two domains: **common** and **uvm**
 - ◆ Executing concurrently as two synchronized threads

```
virtual function void phase_started(uvm_phase phase);
    $display("%s Phase is in %s domain", phase.get_name(), phase.get_domain_name());
endfunction
```



10-15

Phase Domains (2/2)

- User can create customized domains

```
class domain_test extends test_base;
    gpu_env g_env; mp3_env m_env;
    uvm_domain domain0=new("domain0"), domain1=new("domain1");
    virtual function void build_phase(uvm_phase phase);
        g_env.set_domain(domain0);
        m_env.set_domain(domain1);

        domain1.sync(.target(this.get_domain()));
        // domain0.sync(.target(this.get_domain()),
        //              .phase(uvm_shutdown_phase::get()));
        ...
    endfunction
endclass
```

Created two user domains

***g_env*'s phases are independent of all other domains**

***m_env*'s phases are synchronized with test domain**

***g_env*'s shutdown phase is sync'ed with test domain**

10-16

User typically do not need to create customized domains.

The only reason that customized domain is required is when logic blocks within the DUT operates independently of each other. That is, each has its own reset and configuration timing requirements. Without separate domains, there can be gaps within the functional spec that never get tested.

Note: sync() method only synchronize the starting time for the specified phase. The starting time for the following phase are not synchronized.

CAUTION: Currently buggy in implementation for .sync() method.

User Defined Phase

- User can create customized phases

```
`uvm_user_task_phase(new_cfg, test_base, my_)
```

Component where phase will execute

Macro creates a new phase `class` called `my_new_cfg_phase`
`test_base` must implement `new_cfg_phase(uvm_phase phase)`

```
class test_base extends uvm_test; // utils left off
    virtual function void build_phase(uvm_phase phase);
        ... // all environment are constructed here
        begin
            uvm_domain env_domain = env.get_domain();
            env_domain.add(my_new_cfg_phase::get(),
                .after_phase(uvm_configure_phase::get()));
        end
    endfunction
endclass
```

Add phase into domain

10-17

The following are user phase creation macros:

```
`define uvm_user_task_phase(PHASE, COMP, PREFIX) \
`m_uvm_task_phase(PHASE, COMP, PREFIX)

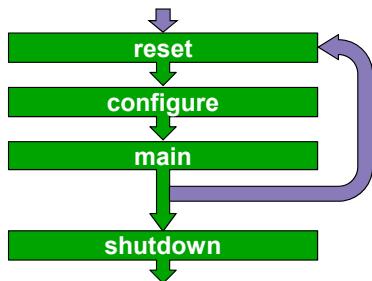
`define uvm_user_topdown_phase(PHASE, COMP, PREFIX) \
`m_uvm_topdown_phase(PHASE, COMP, PREFIX)

`define uvm_user_bottomup_phase(PHASE, COMP, PREFIX) \
`m_uvm_bottomup_phase(PHASE, COMP, PREFIX)
```

Phase Jump: Backward

- Phases can be rolled back
- Mid-simulation reset can be accomplished by jumping back to reset phase
- Environment must be designed to support jumping
 - All active components must return control signals to de-asserted state
 - All residual properties must be deleted
 - All residual processes must be killed

⚠ Make sure clean up is done!



```
class test_jump2reset extends test_base;
  // code not shown
  virtual task main_phase(...);
    // detect some condition
    if (phase.get_run_count() < 5)
      phase.jump(uvm_reset_phase::get());
    endtask
```

10-18

`jump()` method will cause an immediate jump without regards to outstanding objections.

`jump()` only affects the phase execution of the domain in which the jump was executed.

Drivers, monitors and scoreboards all can be in middle of some operation when a jump happens. All thread spun off by these components must be killed. All transactions accumulated by these components but not yet processed also must be deleted.

An alternative to `jump()` is the `set_jump_phase()` method.

The `set_jump_phase()` method specifies which phase to transition to when this phase completes. Unlike `jump()`, `set_jump_phase()` does not terminate the phase immediately. Since the phase transition is objection based, the amount of clean necessary is minimal.

CAUTION: Avoid if you can. Very difficult to manage correctly.

Phase Jump: Forward

- **Can be used for a test to skip a behavior**
 - Example: skip rest of main if coverage is met
- **Environment must be designed to support jumping**
 - May want to wait for components to be in idle state before calling jump

```
class my_test extends uvm_test;  
  ...  
  virtual task main_phase(...);  
    forever begin  
      #1us; // periodic coverage check  
      if ($get_coverage() == 100.0) begin  
        phase.jump(uvm_post_main_phase::get());  
      end  
    end  
  endtask  
endclass
```

10-19

Phase Jumping Cleanup

- **phase_ended() callback**

- Called before each phase terminates

```
virtual function void phase_ended(uvm_phase phase);
    uvm_phase jump_phase = phase.get_jump_target();
    if (jump_phase != null) begin
        if (jump_phase.is_before(phase)) begin
            ...
        end else begin
            if (jump_phase.is_after(phase)) begin
                ...
            end
        end
    endfunction: phase_ended
```

10-20

Get Phase Execution Count

- **get_run_count()**
 - Returns the number of times this phase has executed

```
virtual function void phase_started(uvm_phase phase);
    if (phase.is(uvm_reset_phase::get())) begin
        int count = phase.get_run_count();
        if (count > 1) begin
            ...
        end
    end
endfunction : phase_started
```

10-21

Unit Objectives Review

You should now be able to:

- Control phase objections
- Control phase timeout
- Create user phases
- Create phase domains
- Implement phase callbacks



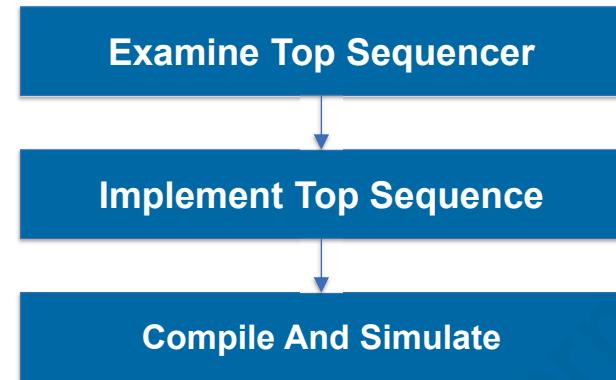
10-22

Lab 5: Top Sequencer and Sequence



30 minutes

- Implement sequence control



10-23

Appendix

uvm_phase Class Key Members

Debugging Objections with Callbacks

Component Phasing Guidelines

Reacting to Reset

10-24

uvm_phase Class Key Methods

- **uvm_phase class key methods are as follows:**

```
class uvm_phase extends uvm_object;
  function bit is(uvm_phase phase);
  function bit is_before(uvm_phase phase);
  function bit is_after(uvm_phase phase);
  function uvm_objection get_objection();
  virtual function void raise_objection(uvm_object obj, string description="", int count=1);
  virtual function void drop_objection(uvm_object obj, string description="", int count=1);
  function uvm_phase find(uvm_phase phase, bit stay_in_scope=1);
  function uvm_phase find_by_name(string name, bit stay_in_scope=1);
  function uvm_phase get_schedule(bit hier=0);
  function string get_schedule_name(bit hier=0);
  function uvm_domain get_domain();
  function string get_domain_name();
  function void add(uvm_phase phase, with_phase=null, after_phase=null, before_phase=null);
  function void sync(uvm_domain target, uvm_phase phase=null, uvm_phase with_phase=null);
  function void unsync(uvm_domain target, uvm_phase phase=null, uvm_phase with_phase=null);
  function void jump(uvm_phase phase);
  function void set_jump_phase(uvm_phase phase);
  static function void jump_all(uvm_phase phase);
  function uvm_phase get_jump_target();
  function int get_run_count();
  function int unsigned get_ready_to_end_count();
  function uvm_phase_state get_state();
  task wait_for_state(uvm_phase_state state, uvm_wait_op op=UVM_EQ);
    // see next slide for possible states
    // possible op: UVM_EQ, UVM_NE, UVM_LT,
    // UVM_LTE, UVM_GT, UVM_GTE
endclass
```

10-25

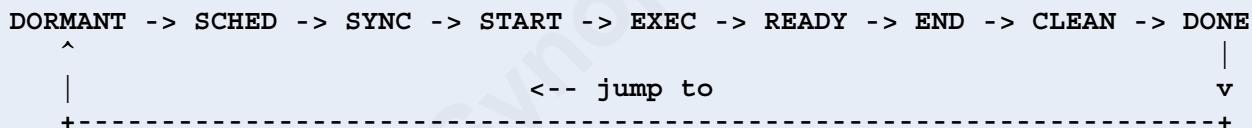
uvm_phase Class States

- The `get_state()` and `wait_for_state()` methods operates on the following possible states of a phase

Phase State	Description
UVM_PHASE_DORMANT	Domain inactive
UVM_PHASE_SCHEDULED	Phase scheduled waiting for preceding phases to complete
UVM_PHASE_SYNCING	All preceding phases completed
UVM_PHASE_STARTED	Phase ready, <code>phase_started()</code> executes
UVM_PHASE_EXECUTING	Phase method executes
UVM_PHASE_READY_TO_END	No objections, <code>phase_ready_to_end()</code> executes
UVM_PHASE_ENDED	Phase completed, <code>phase_ended()</code> executes
UVM_PHASE_CLEANUP	Phase related threads killed
UVM_PHASE_DONE	Done, execute succeeding phase

10-26

Phase when executing goes through the following transition:



```
typedef enum {
    UVM_PHASE_DORMANT      = 1,
    UVM_PHASE_SCHEDULED    = 2,
    UVM_PHASE_SYNCING       = 4,
    UVM_PHASE_STARTED        = 8,
    UVM_PHASE_EXECUTING     = 16,
    UVM_PHASE_READY_TO_END  = 32,
    UVM_PHASE_ENDED          = 64,
    UVM_PHASE_CLEANUP        = 128,
    UVM_PHASE_DONE            = 256,
    UVM_PHASE_JUMPING        = 512
} uvm_phase_state;
```

Appendix

uvm_phase Class Key Members

Debugging Objections with Objection Callbacks

Component Phasing Guidelines

Reacting to Reset

10-27

Debugging Objections with Callbacks

```
class my_objection_cb extends uvm_objection_callback; ...; // constructor not shown
  uvm_object objectors[$];
  virtual function void raised(uvm_objection objection, uvm_object obj, uvm_object source_obj, string description, int count);
    if (obj.get_full_name() != "") begin
      `uvm_info("OBJECTION",
        $sformatf("%t: %s: Raised for %s, total is now %0d", $realtime, objection.get_name(), obj.get_full_name(),
          objection.get_objection_total()+1),
      UVM_MEDIUM);
      objection.get_objectors(objectors);
      foreach (objectors[i]) $display("%t: OBJECTION - objectors[%0d]: %s", $realtime, i, objectors[i].get_full_name());
    end
  endfunction
  virtual function void dropped (uvm_objection objection, uvm_object obj, uvm_object source_obj, string description, int count);
    if (obj.get_full_name() != "") begin
      `uvm_info("OBJECTION",
        $sformatf("%t: %s: dropped for %s, total is now %0d", $realtime, objection.get_name(), obj.get_full_name(),
          objection.get_objection_total()-1),
      UVM_MEDIUM);
      objection.get_objectors(objectors);
      foreach (objectors[i]) $display("%t: OBJECTION - objectors[%0d]: %s", $realtime, i, objectors[i].get_full_name());
    end
  endfunction
endclass
class test_base extends uvm_test; ...; // other code left off
  virtual function void build_phase(uvm_phase phase); ...; // other code left off
    if (uvm_report_enabled(UVM_FULL, UVM_INFO, "OBJECTIONS")) begin
      my_objection_cb cb = new("cb");
      uvm_objection_cbs_t::add(null, cb); //typewide callback
    end
  endfunction
endclass
```

10-28

Appendix

uvm_phase Class Key Members

Debugging Objections with Objection Callbacks

Component Phasing Guidelines

Reacting to Reset

10-29

Driver Guideline

- Emulates launch and capture registers
- Build time phases (emulates external module compile time)
 - `build_phase()` // retrieve configuration
 - `end_of_elaboration_phase()` // configuration and connectivity error check
- Run time phases (emulates external module simulation run time)
 - `start_of_simulation_phase()` // print configuration
 - `run_phase()` // get and process item – emulates `always`
- Phase callbacks
 - Generally not needed



Do NOT implement `pre_reset` through `post_shutdown` phases!
(see note)

10-30

Typically, only the phase methods listed above are implemented in the driver class.

The other phase methods, especially the uvm domain phase methods, should not be implemented in the driver class code. The uvm domain phase methods should be implemented in the test class code to control the sequence execution order.

The typical reason why someone would implement the `reset_phase()` in the driver code is to handle the reset condition. But, this is a wrong approach. By embedding the `reset_phase()` in the driver, you are taking away the ability of the driver to react to a reset in the middle of simulation.

A better approach for handling reset is described in the “Reacting to Reset in Driver, Monitor and Scoreboards” section of the appendix.

Monitor Guideline

- Emulates capture registers
- Build time phases (emulates external module compile time)
 - `build_phase()` // retrieve configuration
 - `end_of_elaboration_phase()` // configuration and connectivity error check
- Run time phases (emulates external module simulation run time)
 - `start_of_simulation_phase()` // print configuration
 - `run_phase()` // report observed transaction – emulates `always`
- Phase callbacks
 - Generally not needed



Do NOT implement pre_reset through post_shutdown phases!
(see note)

10-31

Typically, only the phase methods listed above are implemented in the monitor class.

The other phase methods, especially the uvm domain phase methods, should not be implemented in the monitor class code. The uvm domain phase methods should be implemented in the test class code to control the sequence execution order.

The typical reason why someone would implement the `reset_phase()` in the monitor code is to handle the reset condition. But, this is a wrong approach. By embedding the `reset_phase()` in the monitor, you are taking away the ability of the monitor to react to a reset in the middle of simulation.

A better approach for handling reset is described in the “Reacting to Reset in Driver, Monitor and Scoreboards” section of the appendix.

Agent Guideline

- Emulates external block-level module
- Build time phases (emulates external block-level module compile time)
 - `build_phase()` // construct sub-components
 - // retrieve and set sub-component configuration
 - `connect_phase()` // connect sub-components
 - `end_of_elaboration_phase()` // configuration and connectivity error check
- Run time phases (no module behavior emulation)
 - `start_of_simulation_phase()` // cover & display configuration
- Phase callbacks – not needed



Do NOT implement pre_reset through post_shutdown phases!
(see note)

10-32

Agent class is only a container for behavior component objects. Only the phase methods listed above should be implemented in the agent class.

The other phase methods, especially the uvm domain phase methods and the run phase method, should not be implemented in the agent class code. The uvm domain phase methods and the run phase method should be implemented in the test class code to control the sequence execution order. There should be no reason to implement the uvm domain phase methods in the agent class.

Scoreboard Guideline

- **Tracks correctness of operation**

- **Build time phases**

- `build_phase()` // retrieve configuration if needed
- // construct sub-components if needed
- `connect_phase()` // connect pass-through analysis ports if needed
- `end_of_elaboration_phase()` // configuration error check

- **Run time phases**

- `start_of_simulation_phase()` // optional – cover & print configuration
- `run_phase()` // can be used to support reset detection



Do NOT implement pre_reset through post_shutdown phases!
(see note)

10-33

Typically, only the phase methods listed above are be implemented in the scoreboard class.

The other phase methods, especially the uvm domain phase methods and the run phase method, should not be implemented in the scoreboard class code. The uvm domain phase methods and the run phase method should be implemented in the test class code to control the sequence execution order.

The typical reason why someone would implement the `reset_phase()` in the scoreboard code is to handle the reset condition. But, this is a wrong approach. By embedding the `reset_phase()` in the scoreboard, you are taking away the ability of the scoreboard to react to a reset in the middle of simulation.

A better approach for handling reset is described in the “Reacting to Reset in Driver, Monitor and Scoreboards” section of the appendix.

For determining end of test, the `shutdown_phase()` method should be implemented in the test rather than in the scoreboard.

Environment Phase Guideline

- Emulates external system-level module

- Build time phases

- `build_phase()` // construct sub-components
- // retrieve and set sub-component configuration
- `connect_phase()` // connect sub-components
- `end_of_elaboration_phase()` // configuration and connectivity error check

- Run time phases

- `start_of_simulation_phase()` // cover & display configuration

- Phase callbacks – not needed



Do NOT implement pre_reset through post_shutdown phases!
(see note)

10-34

Just like the agent class, the environment class is only a container class. Only the phase methods listed above should be implemented.

The other phase methods, especially the uvm domain phase methods and the run phase method, should not be implemented in the environment class code. The uvm domain phase methods and the run phase method should be implemented in the test class code to control the sequence execution order. There should be no reason to implement the uvm domain phase methods in the environment class.

Test Phase Guideline

- No restriction/limitation of phases
- Build time phases
 - `build_phase()` // construct and configure environment
 - `connect_phase()` // make changes to environment connections
 - `end_of_elaboration_phase()` // configuration and connectivity error check
- Run time phases
 - `start_of_simulation_phase()` // display configuration
 - `run_phase()` // control sequence execution – emulates Verilog initial
 - `reset_phase()` // execute reset sequence (optional)
 - `configure_phase()` // execute configure sequence (optional)
 - `main_phase()` // execute stimulus sequence (optional)
 - `shutdown_phase()` // additional end of test condition (optional)
- Cleanup phases and callbacks - as needed by test

10-35

Appendix

uvm_phase Class Key Members

Debugging Objections with Objection Callbacks

Component Phasing Guidelines

Reacting to Reset

10-36

Driver Reset Guideline (1/3)

```
class driver extends uvm_driver #(transaction);
  uvm_event  reset_event;
  process    process_aa[process];
  `uvm_component_utils_begin(driver)
    `uvm_field_object(reset_event, UVM_ALL_ON) // field auto retrieved
  `uvm_component_utils_end
  virtual task run_phase(uvm_phase phase);
    interface_reset(); // see next slide
    forever begin
      fork
        begin
          process_aa[process::self] = process::self;
          seq_item_port.get_next_item(req);
          // driver operation
          seq_item_port.item_done();
          process_aa.delete(process::self);
        end
      join
    end
  endtask: run_phase
// Continued on next slide
```

Must store each process in flight

When done each process must remove itself from array

10-37

The following code is left off the slide:

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

virtual function void end_of_elaboration_phase(uvm_phase phase);
  super.end_of_elaboration_phase(phase);
  if (reset_event == null) begin
    `uvm_fatal("CFG_ERR", "reset_event is not available");
  end
endfunction: end_of_elaboration_phase
```

Driver Reset Guideline (2/3)

```
// Continued from previous slide
virtual task interface_reset();
fork
    forever begin
        reset_event.wait_trigger();

        foreach (process_aa[i]) begin
            process_aa[i].kill(); 
        end
        process_aa.delete(); 
        // reset all control signals & all class fields to reset state
    end
    join_none
endtask: interface_reset

// Continued on next slide
```

Must kill process in flight
when reset is detected

Delete process array when done

10-38

Driver Reset Guideline (3/3)

```
// Continued from previous slide
virtual function void start_of_simulation_phase(uvm_phase phase);
    super.start_of_simulation_phase(phase); // other code left off

    // You must turn off the auto transaction recording.
    // If left turn on, you will get a lot of Verdi recording warnings.
    // If interested in transaction recording, you must do it yourself.
    // Make sure to end recording when reset is detected.
`ifndef UVM_VERSION_1_1
    seq_item_port.disable_auto_item_recording();
`else
    // For UVM-1.1, set +UVM_DISABLE_AUTO_ITEM_RECORDING compile-time switch.
    // Unfortunately, that will turn all auto transaction recordings.
    // You need to migrate to UVM-1.2/IEEE UVM to avoid this problem.
`endif
endfunction: start_of_simulation_phase
endclass: driver
```



Should be the start_of_simulation_phase. If coded in build_phase will result in run-time error due to race in when components are constructed

10-39

Transaction Monitor Reset Guideline (1/2)

```
class monitor extends uvm_monitor;
  uvm_event  reset_event;
  process    process_aa[process];
  `uvm_component_utils_begin(monitor)
  `uvm_field_object(reset_event, UVM_ALL_ON) // field auto retrieved
  `uvm_component_utils_end
  virtual task run_phase(uvm_phase phase);
    interface_reset();
    forever begin
      fork
        begin
          process_aa[process::self] = process::self;
          // monitor operational code
          process_aa.delete(process::self);
        end
      join
    end
    endtask: run_phase
// Continued on next slide
```

10-40

The following code is left off the slide:

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

virtual function void end_of_elaboration_phase(uvm_phase phase);
  super.end_of_elaboration_phase(phase);
  if (reset_event == null) begin
    `uvm_fatal("CFG_ERR", "reset_event is not available");
  end
endfunction: end_of_elaboration_phase
```

Transaction Monitor Reset Guideline (2/2)

```
// Continued from previous slide

virtual task interface_reset();
  fork
    forever begin
      reset_event.wait_trigger();
      foreach (process_aa[i]) begin
        process_aa[i].kill();
      end
      process_aa.delete();
      // reset all class fields to reset state
    end
    join_none
  endtask: interface_reset

endclass: monitor
```

10- 41

Reset Monitor Guideline

```
class reset_monitor extends uvm_monitor; // other code left off
  uvm_event                      reset_event;
  `uvm_component_utils_begin(reset_monitor)
    `uvm_field_object(reset_event, UVM_ALL_ON) // field auto retrieved
  `uvm_component_utils_end

  virtual task run_phase(uvm_phase phase);
    forever begin
      observe();
    end
  endtask: run_phase

  virtual task observe();
    @(vif.reset_n);
    if (vif.reset_n == 1'b0) begin
      reset_event.trigger();
    end
  endtask: observe
endclass: reset_monitor
```

10-42

The following code is left off the slide:

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

virtual function void end_of_elaboration_phase(uvm_phase phase);
  super.end_of_elaboration_phase(phase);
  if (reset_event == null) begin
    `uvm_fatal("CFG_ERR", "reset_event is not available");
  end
endfunction: end_of_elaboration_phase
```

Agent Reset Guideline

```
class any_agent extends uvm_agent; // Other content left off
  uvm_event      reset_event;

  `uvm_component_utils_begin(any_agent)
    `uvm_field_object(reset_event, UVM_ALL_ON) // field auto retrieved
  `uvm_component_utils_end

  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    // pass event to agent's children
    uvm_config_db#(uvm_object)::set(this, "", "reset_event", reset_event);
  endfunction: build_phase
endclass: any_agent
```



For auto config_db retrieval, the data type of the object must be **uvm_object**!

10-43

The following code is left off the slide:

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

virtual function void end_of_elaboration_phase(uvm_phase phase);
  super.end_of_elaboration_phase(phase);
  if (reset_event == null) begin
    `uvm_fatal("CFG_ERR", "reset_event is not available");
  end
endfunction: end_of_elaboration_phase
```

Scoreboard Reset Guideline

```
class scoreboard extends uvm_scoreboard;
    Transaction      expected_QUE[$];
    uvm_event        reset_event;

    `uvm_component_utils_begin(scoreboard)
        `uvm_field_object(reset_event, UVM_ALL_ON) // field auto retrieved
    `uvm_component_utils_end

    virtual task run_phase(uvm_phase phase);
        forever begin
            reset_event.wait_trigger();
            uvm_wait_for_nba_region(); // need to allow analysis calls to flush
            expected_QUE.delete();
        end
    endtask: run_phase

    // Scoreboard behavioral code left off
endclass: scoreboard
```

10-44

The following code is left off the slide:

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

virtual function void end_of_elaboration_phase(uvm_phase phase);
    super.end_of_elaboration_phase(phase);
    if (reset_event == null) begin
        `uvm_fatal("CFG_ERR", "reset_event is not available");
    end
endfunction: end_of_elaboration_phase
```

Environment Reset Guideline

```
class environment extends uvm_env;
    reset_agent r_agt;
    other_agent other_agt;
    scoreboard sb;
    uvm_event reset_event;
    `uvm_component_utils_begin(environment)
        `uvm_field_object(reset_event, UVM_ALL_ON) // field auto retrieved
    `uvm_component_utils_end
    virtual function void build_phase(uvm_phase phase); // other code left off
        super.build_phase(phase);
        if (reset_event == null) begin
            `uvm_info("CFG", "Using local reset event", UVM_MEDIUM);
            reset_event = new("reset_event");
        end
        uvm_config_db #(uvm_object)::set(this, "r_agt", "reset_event", reset_event);
        uvm_config_db #(uvm_object)::set(this, "other_agt", "reset_event", reset_event);
        uvm_config_db #(uvm_object)::set(this, "sb", "reset_event", reset_event);
    endfunction: build_phase
endclass: environment
```

Environment distributes the reset event

For auto config_db retrieval, the data type of the object must be **uvm_object**!



10-45

The following code is left off the slide:

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

Sequence Reset Guideline (1/2)

```
class my_sequence extends uvm_sequence #(transaction);
  uvm_object event_obj;
  uvm_event reset_event;
  process process_aa[process];
  virtual task pre_start();
    uvm_config_db#(uvm_object)::get(this.get_sequencer(), "", "reset_event", event_obj);
    if (!$cast(reset_event, event_obj)) begin
      `uvm_fatal("CFG_ERR", "reset_event is not configured correctly");
    end
    interface_reset(); // thread to detect reset and kill sequence
  endtask
  virtual task body()
    fork begin
      process_aa[process::self] = process::self;
      // body code left off
      process_aa.delete(process::self);
    end join
  endtask: body
// continued on next slide
```

For config_db retrieval, the data type
of the object must be **uvm_object**!



10-46

The following code is left off the slide:

```
`uvm_object_utils(my_sequence)

function new(string name = "my_sequence");
  super.new(name, parent);
endfunction: new
```

Sequence Reset Guideline (2/2)

```
// continued from previous slide
virtual function void interface_reset();
  fork
    forever begin
      reset_event.wait_trigger();
      foreach (process_aa[i]) begin
        process_aa[i].kill();
      end
      process_aa.delete();
    end
  join_none
endfunction: interface_reset
endclass: my_sequence
```

10- 47

Sequencer Reset Guideline

```
class my_sequencer #(type T = uvm_sequence_item) extends uvm_sequencer #(T);
  typedef my_sequencer#(T) this_type;
  uvm_event reset_event;
  `uvm_component_param_utils_begin(this_type)
    `uvm_field_object(reset_event, UVM_ALL_ON) // field auto retrieved
  `uvm_component_utils_end
  virtual task run_phase(uvm_phase phase);
    interface_reset(); // reset thread
  endtask: run_phase
  virtual task interface_reset();
    fork
      forever begin
        reset_event.wait_trigger();
        this.stop_sequences();
      end
    join_none
  endtask: interface_reset
endclass: my_sequencer
```

Tells the sequencer to kill all sequences and child sequences currently operating on the sequencer, and remove all requests, locks, and responses that are currently queued. This essentially resets the sequencer to an idle state. (LRM)

10-48

The following code is left off the slide

```
const static string type_name = $sformatf("my_sequencer#(%s)", T::type_name);
virtual function string get_type_name();
  return type_name;
endfunction

function new(string name, uvm_component parent);
  super.new(name, parent);
endfunction

virtual function void end_of_elaboration_phase(uvm_phase phase);
  super.end_of_elaboration_phase(phase);
  if (reset_event == null) begin
    `uvm_fatal("CFG_ERR", "reset_event is not available");
  end
endfunction: end_of_elaboration_phase
```

Agenda

DAY

3

9 UVM Advanced Sequence/Sequencer

10 UVM Phasing and Objections



11 UVM Register Abstraction Layer (RAL)

12 Summary



Unit Objectives



After completing this unit, you should be able to:

- Create ralf file to represent DUT registers
- Use ralgen to create UVM register classes
- Use UVM register in sequences
- Implement adapter to pass UVM register content to drivers
- Run built-in UVM register tests

11-2

Register & Memories

- Every DUT has them
- First to be verified
 - Reset value
 - Bit(s) behavior
- High maintenance
 - Modify tests
 - Modify firmware model

3

Registers

3.1 MODER (Mode Register)

Bit #	Access	Description
31-17		Reserved
16	RW	RECSMALL – Receive Small Packets 0 = Packets smaller than MINFL are ignored. 1 = Packets smaller than MINFL are accepted.
15	RW	PAD – Padding enabled 0 = Do not add pads to short frames. 1 = Add pads to short frames (until the minimum frame length is equal to MINFL).
14	RW	HUGEN – Huge Packets Enable 0 = The maximum frame length is MAXFL. All additional bytes are discarded. 1 = Frames up 64 KB are transmitted.
13	RW	CRCEN – CRC Enable 0 = Tx MAC does not append the CRC (passed frames already contain the CRC). 1 = Tx MAC appends the CRC to every frame.
12	RW	DLYCRCEN – Delayed CRC Enabled 0 = Normal operation (CRC calculation starts immediately after the SFD).

11-3

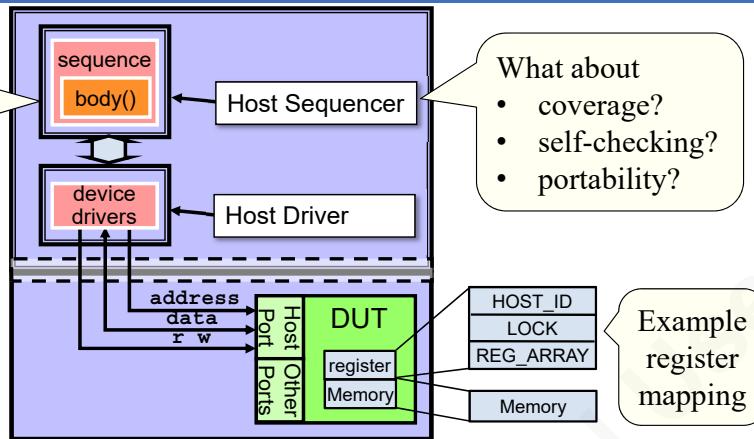
The register and memory space of a design brings a lot of overhead to verification. Any change to the register/memory space in the specification has an avalanche effect on the DUT and TB. Every test and component affected must be changed. This can be time consuming as well as a maintenance nightmare (likely to miss some and have odd test results).

RAL reduces the impact of these spec changes, by abstracting the address out of the register read and write calls and automating the generation of register classes. Because of this, address space changes are made to one file (machine consumable spec) and automatically propagated to the rest of the TB environment

Testbench without UVM Register Abstraction

- A typical test
- check reset value
 - write value
 - check value

- What about
- coverage?
 - self-checking?
 - portability?

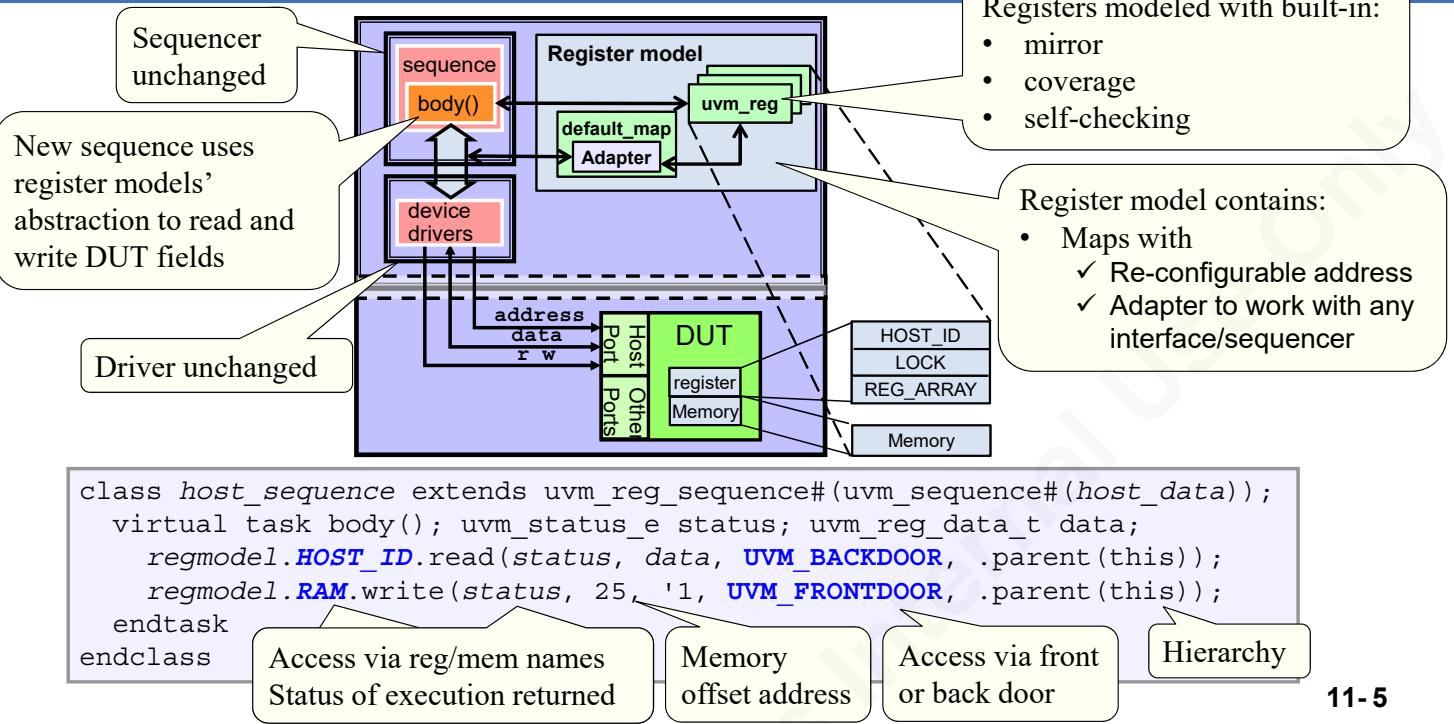


```
class host_sequence extends uvm_sequence #(host_data); ...;
virtual task body();
`uvm_do_with(req, {addr=='h100; data=='1; kind==UVM_WRITE;});
`uvm_do_with(req, {addr=='h1025; kind==UVM_READ;}); ...
endtask
endclass
```

Address hardcoded! Field name unknown!
Status of execution unknown! Front door access only!

11-4

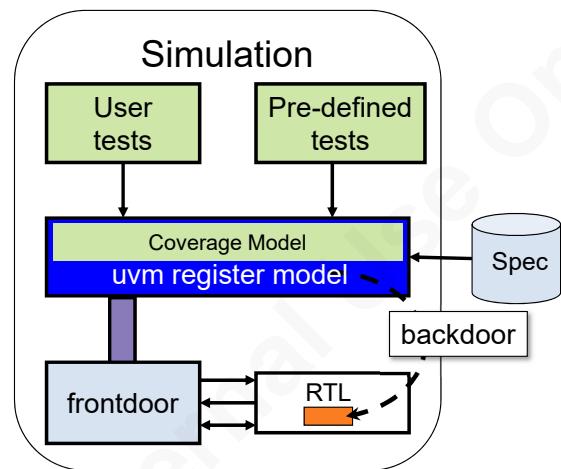
Testbench with UVM Register Abstraction



11-5

UVM Register Abstraction

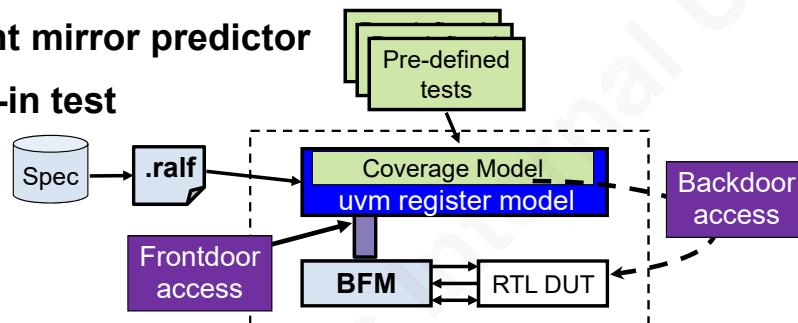
- Abstracts reading/writing to configuration fields and memories
- Supports both front door and back door access
- Mirrors register data
- Built-in functional coverage
- Hierarchical model for ease of reuse
- Pre-defined tests exercise registers and memories



11-6

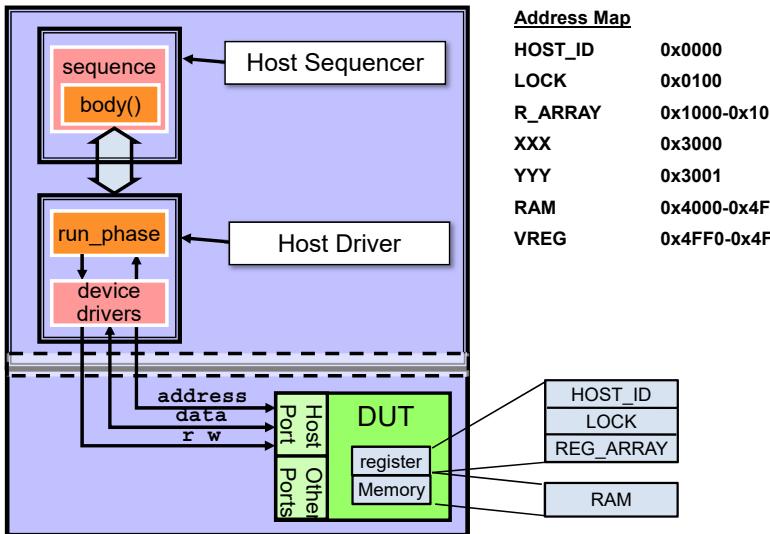
Implement UVM Register Abstraction

- Step 1: Verify frontdoor without UVM register abstraction
- Step 2: Describe register fields in .ralf file
- Step 3: Use ralgen to create UVM register abstraction
- Step 4: Create UVM register abstraction adapter
- Step 5: Add UVM register abstraction and adapter in environment
- Step 6: Write and run UVM register abstraction sequence
- Optional: Implement mirror predictor
- Optional: Run built-in test



11-7

Example Specification



Address Map

HOST_ID	0x0000
LOCK	0x0100
R_ARRAY	0x1000-0x10FF
XXX	0x3000
YYY	0x3001
RAM	0x4000-0x4FFF
VREG	0x4FF0-0x4FFF

HOST_ID Register

Field	CHIP	REV
Bits	15-8	7-0
Mode	ro	ro
Reset	0x5A	0x03

LOCK Register

Field	LOCK
Bits	15-0
Mode	w1c
Reset	0xffff

R_ARRAY[256] Registers

Field	H_REG
Bits	15-0
Mode	rw
Reset	0x0000

Register XXX

HOST_ID	0x0000
LOCK	0x0100
REG_ARRAY	0x1000-0x10FF

Register YYY

HOST_ID	0x0000
LOCK	0x0100
REG_ARRAY	0x1000-0x10FF

RAM (4K)

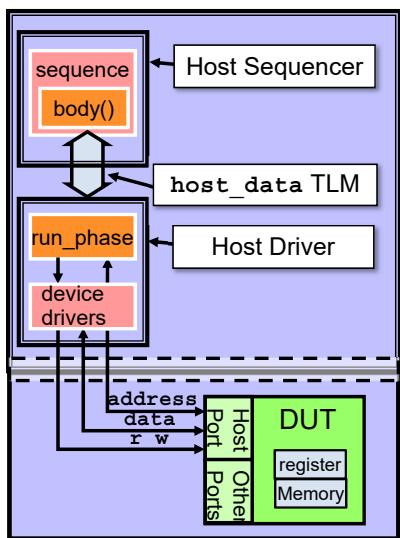
Bits	15-0
Mode	rw

VREG[16]

Bits	15-0
Mode	rw

11-8

Step 1: Create Host Data & Driver



```
class host_data extends uvm_sequence_item;
// constructor and utils macro not shown
  rand uvm_access_e kind;
  uvm_status_e status;
  rand bit[15:0] addr, data;
endclass
```

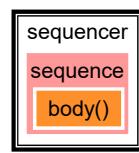
Transaction specifies operation,
address, data and status

```
class host_driver extends uvm_driver #(host_data);
// constructor and utils macro not shown
task run_phase(uvm_phase phase);
  forever begin
    seq_item_port.get_next_item(req);
    data_rw(req); // call device driver
    seq_item_port.item_done();
  end
endtask
// device drivers not shown;
endclass
```

11-9

Step 1: Create Host Sequence

■ Implement host sequence



```
class host_bfm_sequence extends uvm_sequence #(host_data);
  // utils macro, constructor, pre/post_start not shown
  task body();
    `uvm_do_with(req, {addr=='h000; kind==UVM_READ;});
    `uvm_do_with(req, {addr=='h100; data=='1; kind==UVM_WRITE;});
    `uvm_do_with(req, {addr=='h1025; kind==UVM_READ;});
  endtask
endclass
```

Sequence hardcodes register addresses
Access DUT through front door

11-10

Create a simple host sequence without RAL to verify the front door operation of the DUT.

Verify Frontdoor Host is Working

- Follow UVM guideline and complete test structure
- Then, compile and run simulation

```
class host_agent extends uvm_agent; // support code not shown
  typedef uvm_sequencer #(host_data) host_sequencer;
  host_driver drv; host_monitor mon; host_sequencer sgr;
endclass

class host_env extends uvm_env;
  host_agent h_agt; // other support code not shown
endclass

class test_base extends uvm_test;
  host_env h_env; // other support code not shown
  function void build_phase(uvm_phase phase);
    // super.build_phase and construction of components not shown
    uvm_config_db#(virtual host_io)::set(this, "h_env.h_agt", "vif",
                                             router_test_top.host_io);
    uvm_config_db#(uvm_object_wrapper)::set(this, "h_agt.configure_phase",
                                             "default_sequence", host_bfm_sequence::get_type());
  endfunction
endclass
```

11-11

Create a simple host env and test it via a simple test without RAL to verify the front door operation of the DUT.

Step 2: Create .ralf File Based on Spec

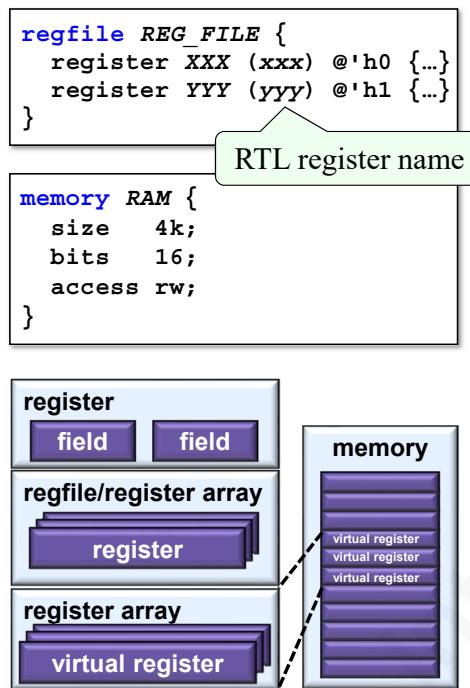
```

register HOST_ID {
    field REV_ID {
        bits 8;
        access ro;
        reset 'h03;
    }
    field CHIP_ID {
        bits 8;
        access ro;
        reset 'h5A;
    }
}

register LOCK {
    field LOCK {
        bits 16;
        access wlc;
        reset 'hffff;
    }
}

register R_ARRAY {
    field H_REG {
        bits 16;
        access rw;
        reset 'h0000;
    }
}

```



HOST_ID Register		
Field	CHIP_ID	REV_ID
Bits	15-8	7-0
Mode	ro	ro
Reset	0x5A	0x03

LOCK Register	
Field	LOCK
Bits	15-0
Mode	w1c
Reset	0xfffff

R_ARRAY[256] Registers	
Field	H_REG
Bits	15-0
Mode	rw
Reset	0x0000

Register XXX
Register YYY
RAM (4K)
Bits 15-0
Mode rw
VREG[16]
Bits 15-0

11-12

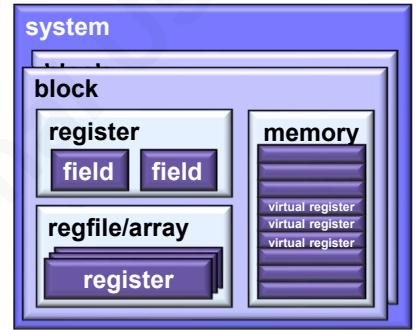
Step 2: Create .ralf File Based on Spec

.ralf block emulates RTL block module level
Specifies all content of block

```
block host_regmodel {
    bytes 2;
    register HOST_ID      (chip_id)      @'h0000;
    register LOCK          (lock)         @'h0100;
    register R_ARRAY[256]  (host_reg[%d]) @'h1000;
    register nested        (subblk.nested) @'h2000;
    regfile REG_FILE       @'h3000;
    memory RAM             (ram)          @'h4000;
    virtual register VREG[16]      RAM @'hOFF0 {
        field VREG { bits 16; access rw; }
    }
}

system dut_regmodel {
    bytes 2;
    block host_regmodel
    block other_regmodel
}
```

Address Map	
HOST_ID	0x0000
LOCK	0x0100
R_ARRAY	0x1000-0x10FF
XXX	0x3000
YYY	0x3001
RAM	0x4000-0x4FFF
VREG	0x4FF0-0x4FFF



If system has multiple instances of the same block, each instance must be named like the following:

```
system dut_regmodel {
    bytes 2;
    block host_regmodel=BLK0 (blk0) @'h0000;
    block host_regmodel=BLK1 (blk1) @'h8000;
}
```

Each instance of the block must be named

UVM Register Abstraction File (.ralf) Syntax

■ Field

- Contains
 - ◆ Bits
 - ◆ Access
 - ◆ Reset
 - ◆ Constraints
 - ◆ Coverage

```
field field_name {
    bits n;
    access rw|ro|wo|w1|w0c|w1c|rc|...;
    reset value;
    [constraint name { <expressions> }]
    [enum { <name[=val],> }]
    [cover <+|- b|f>]
    [coverpoint {<bins name[[n]] = {<n| [n:n],>} | default>}]
}
```

```
field REV {
    bits 8;
    access ro;
    reset 'h03;
}
```



11-14

For detailed RALF file syntax and description, please consult UVM Register Abstraction Layer Generator User Guide in SolvNet.

UVM Register Abstraction: Field

RAL field can have any of the following specifications

bits	Number of bits in the field
access	See next slide
reset	Specify the hard reset value for the field
constraint	Constraint to be used when randomizing field
enum	Define symbolic name for field values
cover	Specifies bits in fields are to be included (+b) in or excluded (-b) from the register-bit coverage model. Specifies coverpoint is a goal (+f). If not (-f) coverpoint weight will be zero.
coverpoint	Explicitly specifies the bins in coverpoint for this field

11-15

For detailed RALF file syntax and description, please consult UVM Register Abstraction Layer Generator User Guide in SolvNet.

Field Access Types

RW	read-write
RO	read-only
WO	write-only; read error
W1	write-once
WO1	write-once; read error
W0C/S/T	write a 0 to bitwise-clear/set/toggle matching bits
W1C/S/T	write a 1 to bitwise-clear/set/toggle matching bits
RC/S	read clear /set all bits
WC/S	write clear /set all bits
WOC/S	write-only clear/set matching bits; read error
WRC/S	read clear/set all bits
WSRC [WCRS]	write sets all bits; read clears all bits [inverse]
W1SRC [W1CRS]	write one set matching bits; read clears all bits [inverse]
W0SRC [W0CRS]	write zero set matching bits; read clears all bits [inverse]
NOACCESS	no effect for write and read

11-16

For detailed RALF file syntax and description, please consult UVM Register Abstraction Layer Generator User Guide in SolvNet.

UVM Register Abstraction File (.ralf) Syntax

■ Register contains fields

```
register reg_name {  
    field name[=rename] [[n]] [@bit_offset[+incr]];  
    field name[[n]] [(hdl_path)] [@bit_offset] {<properties>}  
    [bytes n;]  
    [left_to_right;]  
    [<constraint name {<expression>}>]  
    [shared [(hdl_path)];]  
    [cover <+|- a|b|f>]  
    [cross <cross_item1> ... <cross_itemN>] [{label <cross_label_name>}]  
}
```

```
register HOST_ID {  
    field REV_ID;  
    field CHIP_ID;  
}
```

```
register LOCK {  
    bytes 2;  
    field LOCK {  
        access wlc;  
        reset 'hffff;  
    }  
}
```



11-17

For detailed RALF file syntax and description, please consult UVM Register Abstraction Layer Generator User Guide in SolvNet.

UVM Register Abstraction: Register

RAL registers can have any of the following specifications

bytes	Number of bytes in the register
left_to_right	If specified, fields are concatenated starting from the most significant side of the register but justified to the least-significant side
[n]	Array of fields
bit_offset	Bit offset from the least-significant bit
incr	Array offset bit increment
hdl_path	Specify the module instance containing the register (backdoor access)
shared	All blocks instancing this register share the space
cover	Specifies if address of register should be excluded (-a) from the block's address map coverage model.
cross	Specifies cross coverage of two or more fields of coverpoints. The cross coverage can have a label.

11-18

For detailed RALF file syntax and description, please consult UVM Register Abstraction Layer Generator User Guide in SolvNet.

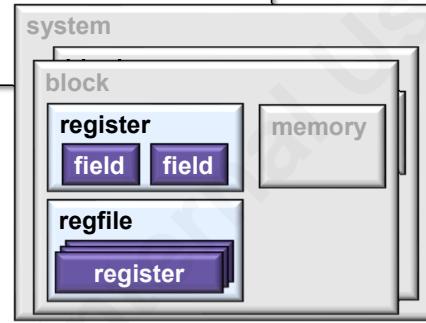
UVM Register Abstraction File (.ralf) Syntax

- Register File contains register(s)

```
regfile regfile_name {
    register name[=rename] [[n]] [(hdl_path)] [@offset];
    register name[[n]] [(hdl_path)] [@offset] {<property>}
    [<constraint name {<expression>}>]
    [shared [(hdl_path)]]
    [cover <+|- a|b|f>]
```

```
}
```

```
regfile REG_FILE {
    register XXX (xxx) @'h0 {
        field xxx {
            bits 16;
            access rw;
            reset 'h0000;
        }
        register YYY (yyy) @'h1 {
            ...
        }
    }
```



11-19

For detailed RALF file syntax and description, please consult UVM Register Abstraction Layer Generator User Guide in SolvNet.

UVM Register Abstraction File (.ralf) Syntax

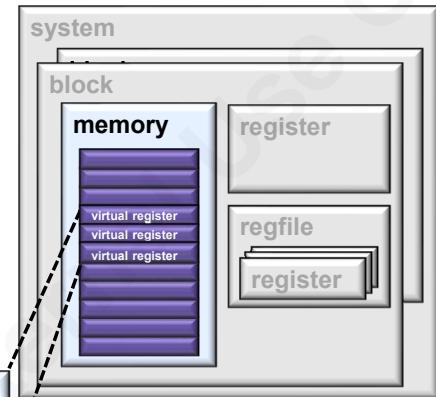
■ Memory

```
memory mem_name {
    size m[k|M|G];
    bits n;
    access rw|ro;
    [addr|literal[++|--]];
    [shared [(hdl_path)]];
}
virtual register reg_name {
    field field_name {
        bits n; access rw|ro;
    }
}
```

```
memory RAM {
    size 4k;
    bits 16;
    access rw;
}
```

• Emulated registers in memory

```
virtual register VREG[16] RAM @'h0FF0 {
    field VREG { bits 16; access rw; }
}
```



11-20

For detailed RALF file syntax and description, please consult UVM Register Abstraction Layer Generator User Guide in SolvNet.

UVM Register Abstraction File (.ralf) Syntax

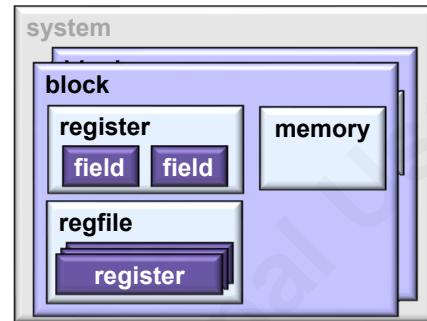
■ Blocks

- Defines content of block layer module
- Can contain domains that define multiple interfaces accessing the same register (minimum of two if specified)
- Can be instantiated in system

```
block blk_name {  
    <property>  
}
```



```
block blk_name {  
    domain PCI {  
        <property>  
    }  
    domain AMBA {  
        <property>  
    }  
}
```



11-21

For detailed RALF file syntax and description, please consult UVM Register Abstraction Layer Generator User Guide in SolvNet.

UVM Register Abstraction: Block

■ Contains

- Register
- Memory
- Block

```
block blk_name {
    bytes n;
    [endian no|little|big|fifo_ls fifo_ms;]
    [<register name[=rename] [[n]] [(hdl_path)] [@offset];>]
    [<register name[[n]] [(hdl_path)] [@offset] {<property>}]
    [<regfile name[=rename] [[n]] [(hdl_path)] [@offset] [+incr];>]
    [<regfile name[[n]] [(hdl_path)] [@offset] [+incr] {<property>}]
    [<memory name[=rename] [(hdl_path)] [@offset];>]
    [<memory name [(hdl_path)] [@offset] {<property>}>]
    [<constraint name {<expression>}>]
}

block host_regmodel {
    bytes 2;
    register HOST_ID
    register LOCK          (lock)      @'h0000;
    register R_ARRAY[256]   (host_reg[%d]) @'h0100;
    regfile REG_FILE       @'h1000;
    memory RAM             (ram)      @'h3000;
    virtual register VREG[16]        RAM @'h4000;
    field VREG { bits 16; access rw; }
}
```

Register array

Must add index

11-22

For detailed RALF file syntax and description, please consult UVM Register Abstraction Layer Generator User Guide in SolvNet.

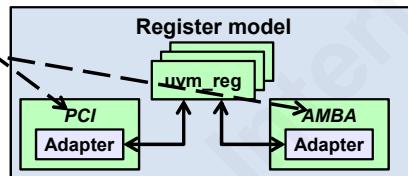
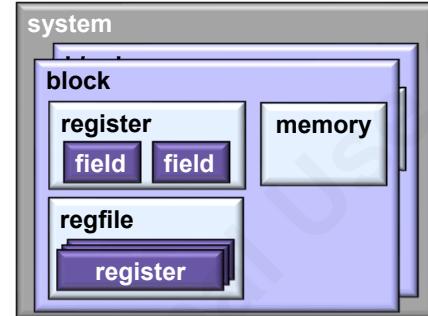
UVM Register Abstraction File (.ralf) Syntax

■ Top Level or subsystem

- Can contain domains that define multiple interfaces accessing the same register (minimum of two if specified)

```
system sys_name {  
    <property>  
}
```

```
system sys_name {  
    domain PCI {  
        <property>  
    }  
    domain AMBA {  
        <property>  
    }  
}
```



11-23

For detailed RALF file syntax and description, please consult UVM Register Abstraction Layer Generator User Guide in SolvNet.

UVM Register Abstraction: System

■ Top Level or subsystem

- Contains other systems or blocks

```
system sys_name {
    bytes n;
    endian no|little|big|fifo_ls fifo_ms;
    [<block name[.domain]=rename] [[n]] [(hdl_path)] @offset [+incr] ; >
    [<block name[[n]] [(hdl_path)] @offset [+incr] {<property>} >]
    [<system name[.domain]=rename] [[n]] [(hdl_path)] @offset [+incr] ; >
    [<system name[[n]] [(hdl_path)] @offset [+incr] {<property>} >]
    [<constraint name {<expression>} >]
}
```

```
system dut_regmodel {
    bytes 2;
    block host_regmodel=HOST0 (blk0) @'h0000;
    block host_regmodel=HOST1 (blk1) @'h8000;
}
```

11-24

For detailed RALF file syntax and description, please consult UVM Register Abstraction Layer Generator User Guide in SolvNet.

Step 3: Create UVM Register Abstraction Model

```
ralgen -uvm -t dut_regmodel host.ralf
```

```
// host.ralf
register HOST_ID {
    field REV_ID {...}
    field CHIP_ID {...}
}
register LOCK {
    field LOCK {...}
}
register R_ARRAY {
    field H_REG {...}
}
regfile REG_FILE {...}
memory RAM {...}
block host_regmodel {...}
system dut_regmodel {...}
```

```
// ral_dut_regmodel.sv
class ral_reg_HOST_ID extends uvm_reg;
    uvm_reg_field REV_ID;
    uvm_reg_field CHIP_ID;
    ...
endclass : ral_reg_HOST_ID
class ral_reg_LOCK extends uvm_reg;
class ral_reg_R_ARRAY extends uvm_reg;
class ral_regfile_REG_FILE extends uvm_regfile;
class ral_mem_RAM extends uvm_mem;
class ral_block_host_regmodel extends uvm_reg_block;
    rand ral_reg_HOST_ID      HOST_ID;
    rand ral_reg_LOCK        LOCK;
    rand ral_reg_R_ARRAY    R_ARRAY[256];
    rand ral_regfile_REG_FILE REG_FILE;
    rand ral_mem_RAM        RAM;
    ...
endclass : ral_block_host_regmodel
class ral_sys_dut_regmodel extends uvm_reg_block;
    rand ral_block_host_regmodel HOST0;
    rand ral_block_host_regmodel HOST1;
    ...
endclass : ral_sys_dut_regmodel
```

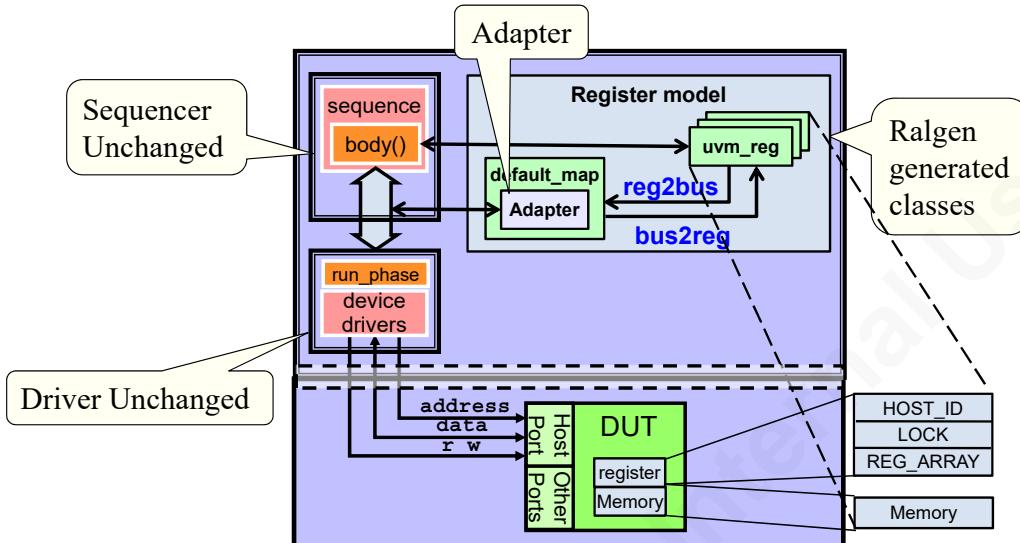
UVM
RAL
Classes

11-25

For detailed ralgen usage, please consult UVM Register Abstraction Layer Generator User Guide in SolvNet.

Step 4: Create UVM Register Adapter

- Environment needs adapters to convert UVM register data to bus transactions for each agent/sequencer

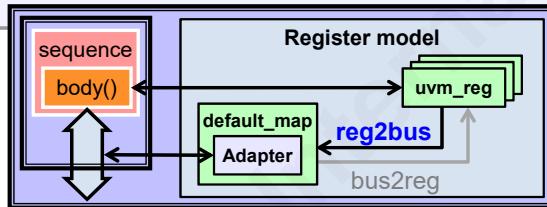


11-26

Sequencer Adapter Class (1/2)

- UVM abstracted registers need to be translated to what the driver can understand (`reg2bus`)

```
class reg_adapter extends uvm_reg_adapter;  
  virtual function uvm_sequence_item reg2bus(const ref uvm_reg_bus_op rw);  
    host_data tr;  
    tr = host_data::type_id::create("tr");  
    tr.addr = rw.addr;  
    tr.data = rw.data;  
    tr.kind = rw.kind;  
    return tr;  
  endfunction  
// Continued on next slide
```

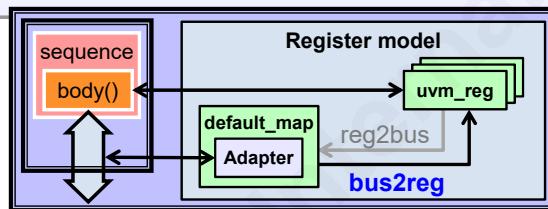


11-27

Sequencer Adapter Class (2/2)

- **bus2reg** translates bus transaction back to RAL

```
// Continued from previous slide
virtual function void bus2reg(uvm_sequence_item bus_item, ref uvm_reg_bus_op rw);
    host_data tr;
    if (!$cast(tr, bus_item)) `uvm_fatal(...);
    rw.addr = tr.addr;
    rw.data = tr.data;
    rw.kind = tr.kind;
    rw.status = tr.status;
endfunction
endclass
```



11-28

Optional Feature in Adapter Class

- There is an optional field in the access methods (write, read, peak, poke) called extension

- Used to pass additional information to the driver

```
task write(..., input uvm_object extension = null, ...)
```

Example:

```
typedef enum { by_byte, by_word } host_access_t;
class host_ext extends uvm_object; // code left off
    rand host_access_t access_mode;
    constraint mode { soft access_mode == by_word; }
endclass
```

```
task ral_sequence::body(); host_ext ext = new();
    ext.randomize() with { access_mode == by_byte; };
    regmodel.R0.write(status, data, .parent(this), .extension(ext));
endtask
```

```
function uvm_sequence_item ral_adapter::reg2bus(...);
    host_ext extension = host_ext'(get_item().extension);
    if (extension != null)
        host_data.access_mode = extension.access_mode;
endfunction
```

11-29

Step 5: Instantiating UVM Register Model

- Can be self-constructed or passed in via configuration database

```
class host_env extends uvm_env; // Other code not shown
host_agent          h_agt;
ral_block_host_regmodel regmodel;
virtual function void build_phase(uvm_phase phase); // Code simplified
uvm_config_db#(ral_block_host_regmodel)::get(this, "", "regmodel", regmodel);
if (regmodel == null) begin
  regmodel = ral_block_host_regmodel::type_id::create("regmodel", this);
  regmodel.build();
  regmodel.lock_model();
```

Lock register hierarchy
and create address map

```
  Create UVM  
  register hierarchy.  
  Not build_phase()!
```

```
  Pass register model to agent  
  for sequence to pick up
```

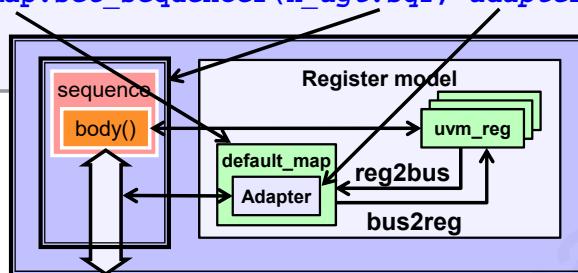
```
  uvm_config_db#(ral_block_host_regmodel)::set(this, "h_agt", "regmodel", regmodel);
end
endfunction // Continued on next slide
```

11-30

Tie Sequencer Adapter to Register Map

- Register each sequencer with the associated adapter

```
virtual function void connect_phase(uvm_phase phase);  
    reg_adapter adapter = reg_adapter::type_id::create("adapter", this);  
    super.connect_phase(phase);  
    regmodel.default_map.set_sequencer(h_agt.sqr, adapter);  
endfunction  
endclass
```



- A map within a register model represents an interface
 - Model with only one interface uses default_map
 - Model with multiple interfaces use map names specified by user
 - ◆ Domain name in .ralf file (for ralgen)

11-31

UVM Register Abstraction Sequence

```
class bfm_sequence extends uvm_sequence#(host_data);
    // other code not shown
    virtual task body();
        `uvm_do_with(req, {addr=='h0; kind==UVM_READ;});
        `uvm_do_with(req, {addr=='h4009; data=='1; kind==UVM_WRITE;});
    endtask
endclass
```

Becomes

```
class ral_sequence extends uvm_reg_sequence #(uvm_sequence#(host_data));
    ral_block host_regmodel regmodel; // other code not shown
    virtual task pre_start(); super.pre_start();
        uvm_config_db#(ral_block_host_regmodel)::get(
            get_sequencer().get_parent(), "", "regmodel", regmodel)
    endtask
    virtual task body(); uvm_status_e status; uvm_reg_data_t data;
        regmodel.HOST_ID.read(status, data, .parent(this)); // can specify .path
        regmodel.RAM.write(status, 9, '1, .parent(this)); // defaults to frontdoor
    endtask
endclass
```

Retrieve regmodel from database via agent

Abstracted and self-documenting code

11-32

When you parameterize a uvm_reg_sequence with a sequence class, the uvm_reg_sequence is a derivative of that class. In the above example, this means that in host_ral_sequence class, you don't need to raise or drop objection because it is already done in the host_sequence_base class.

Run RAL Sequence Implicitly or Explicitly

```
class test_ral_implicit extends test_ral_base; // Support code not shown  
virtual function void build_phase(uvm_phase phase); // Other code  
    uvm_config_db#(uvm_object_wrapper)::set(this,"*.sqr.configure_phase",  
        "default_sequence", host_ral_sequence::get_type());  
endfunction  
endclass
```

Execute RAL sequence implicitly

```
class test_ral_explicit extends test_ral_base; // Support code not shown  
// Not shown-in start_of_simulation_phase::set default_sequence to null  
virtual task configure_phase(uvm_phase phase);  
    host_ral_sequence h_seq; super.configure_phase(phase);  
    phase.raise_objection(this);  
    h_seq = host_ral_sequence::type_id::create("h_seq", this);  
    h_seq.start(env.h_agt.sqr);  
    phase.drop_objection(this);  
endtask  
endclass
```

Or, execute RAL sequence explicitly

11-33

As have already been explained in Unit 4, a sequence can be executed either implicitly or explicitly. Both will accomplish the same thing.

The reason to chose implicit sequence execution is for ease of re-use. Implicit sequence execution can be turned off by setting the "default_sequence" configuration of the sequencer to null. And, implicit sequence execution can be replaced in the test by simply setting the "default_sequence" configuration to another sequence.

The reason to chose explicit sequence execution is that coding is very straight forward. You construct and execute the sequence manually at the time of your choosing. However, once implemented, it is very difficult to change the behavior. You cannot easily disable it. You cannot easily replace the execution with something else.

UVM Register Test Sequences

- Some of test sequences depend on mirror to be updated when backdoor is used to access DUT registers
 - You need to call `set_auto_predict()` in test or implement explicit predictor

Sequence Name	Description
uvm_reg_hw_reset_seq	Test the hard reset values of register
uvm_reg_bit_bash_seq	Bit bash all bits of registers
uvm_reg_access_seq	Verify accessibility of all registers
uvm_mem_walk_seq	Verify memory with walking ones algorithm
uvm_mem_access_seq	Verify access by using front and back door
uvm_reg_mem_builtin_seq	Run all reg and memory tests
uvm_reg_mem_hdl_paths_seq	Verify hdl_path for reg and memory
uvm_reg_mem_shared_access_seq	Verify accessibility of shared reg and memory

11-34

Execute RAL Test Sequence

- Make sure RAL model is set to the correct prediction mode

```
class test_ral extends test_base; // support code left off
  // code identical to auto predict test is left off
  virtual task run_phase(uvm_phase phase);
    phase.raise_objection(this, "Starting reset tests");
    rst_seq = virtual_reset_sequence::type_id::create("rst_seq", this);
    rst_seq.start(env.v_reset_sqr);
    clp.get_arg_value("+seq=", seq_name);
    $cast(selftest_seq, uvm_factory::get().create_object_by_name(seq_name));
    env.regmodel.default_map.set_auto_predict(1);

    selftest_seq.model = env.regmodel;
    selftest_seq.start(env.h_agt.sqr);
    phase.drop_objection(this, "Done with register tests");
  endtask
endclass
```

0 – Explicit prediction
1 – Auto prediction

Select sequence at run-time

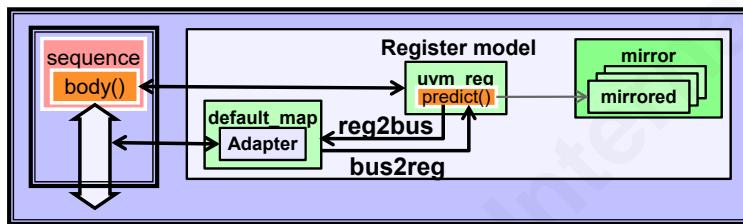
```
simv +UVM_TESTNAME=test_ral +seq=uvm_reg_hw_reset_seq
```

11-35

Enabling Auto Mirror Prediction

- To enable automatic mirror update when registers are written and read:
call `set_auto_predict(1)`

- Advantage:
 - Simple: Read, write methods automatically updates mirror. No user code required.
- Drawbacks:
 - Timing of FRONTDOOR update is not cycle accurate
 - Changes internal to DUT is not reflected in mirror
 - Not a good choice if mirror value is needed in user tests



11-36

Mirror prediction is set to explicit mirror prediction mode by default. To enable auto mirror prediction, one must call the `set_auto_predict()` method with the argument set to 1.

Explicit (Manual) Mirror Prediction

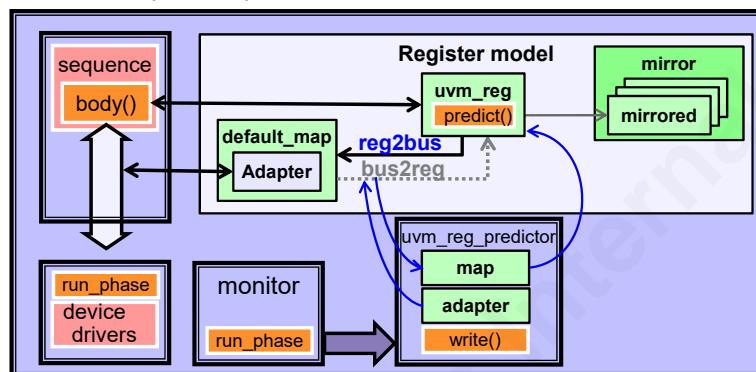
■ Mirror updates when monitor observe register changes

- Advantage:

- Mirror is updated based on observation of bus protocol
- Changes internal to DUT can be monitor and reflected in mirror
- Correct choice if mirror value is needed in user tests

- Drawbacks:

- More complex to set up. Requires a monitor.



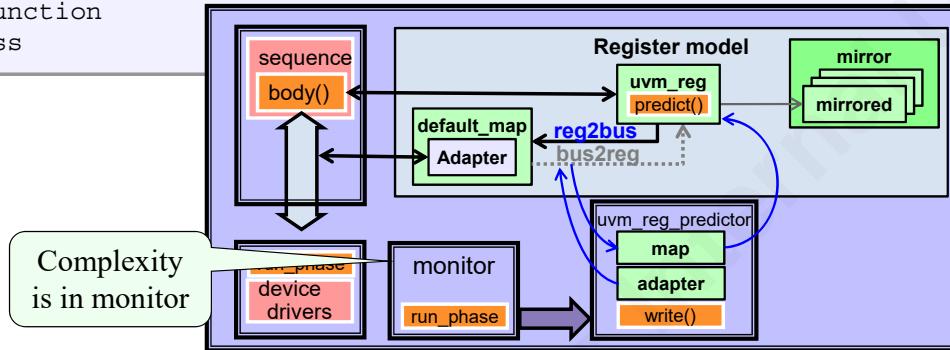
11-37

Mirror prediction is set to explicit mirror prediction mode by default. But nothing happens by default.

To get true explicit mirror prediction, one must implement a monitor to detect changes and connect the monitor to the uvm_reg_predictor. See next slide.

Connecting Explicit Mirror Predictor

```
class host_env extends uvm_env; // Other support code not shown
  typedef uvm_reg_predictor #(host_data) hreg_predictor;
  hreg_predictor hreg_predict;
  virtual function void build_phase(uvm_phase phase); // Other code
    hreg_predict = hreg_predictor::type_id::create("hreg_predict", this);
  endfunction
  virtual function void connect_phase(uvm_phase phase); // Other code
    hreg_predict.map = regmodel.get_default_map();
    hreg_predict.adapter = adapter;
    regmodel.default_map.set_auto_predict(0); // Disable auto predict
    h_agt.analysis_port.connect(hreg_predict.bus_in);
  endfunction
endclass
```



11-38

The `uvm_reg_predictor` class is a class supplied in the UVM source code as a base class for you to use. There is a analysis export (`bus_in`) built into the class. You need to connect the predictor's analysis export to the monitor that is observing the physical bus. When a register transaction is observed by the monitor on the physical bus, the monitor passes the transaction on to the predictor via the analysis port. The predictor then uses the address map that's built into the regmodel and performs a reverse lookup of what register corresponds to the observed address. Then, the predictor populates the mirror of the register with the observed data.

If you use ralgen to generate backdoor access code, ralgen can generate a mirror update mechanism for you. Please see slide 11-99.

Unit Objectives Review

You should now be able to:

- Create ralf file to represent DUT registers
- Use ralgen to create UVM register classes
- Use UVM register in sequences
- Implement adapter to pass UVM register content to drivers
- Run built-in UVM register tests



11-39

Appendix

UVM Register Modes

UVM Memory Modes

ralgen Options

UVM Backdoor

UVM Register Classes

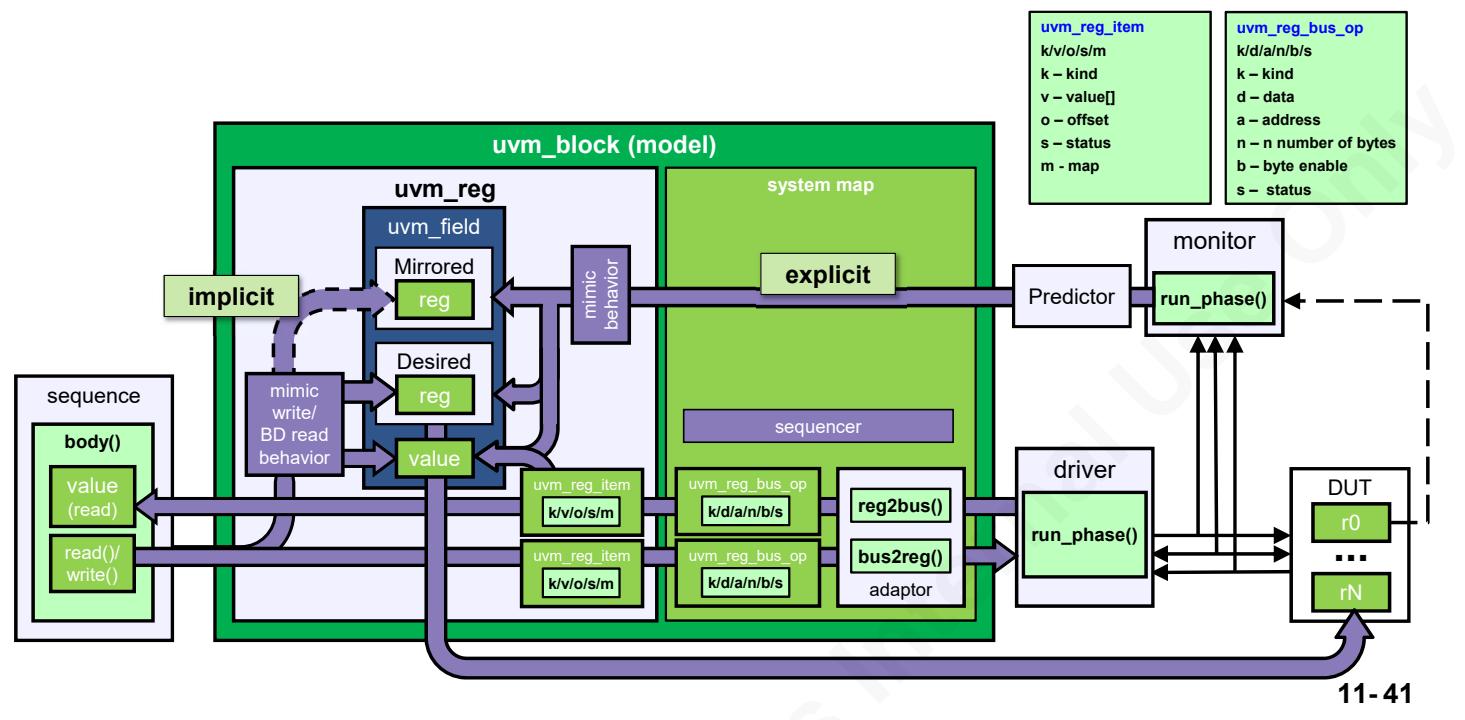
UVM Register Callbacks

Changing Address Offsets of a Domain

Accessing DUT signals via DPI

11-40

UVM Register Modes

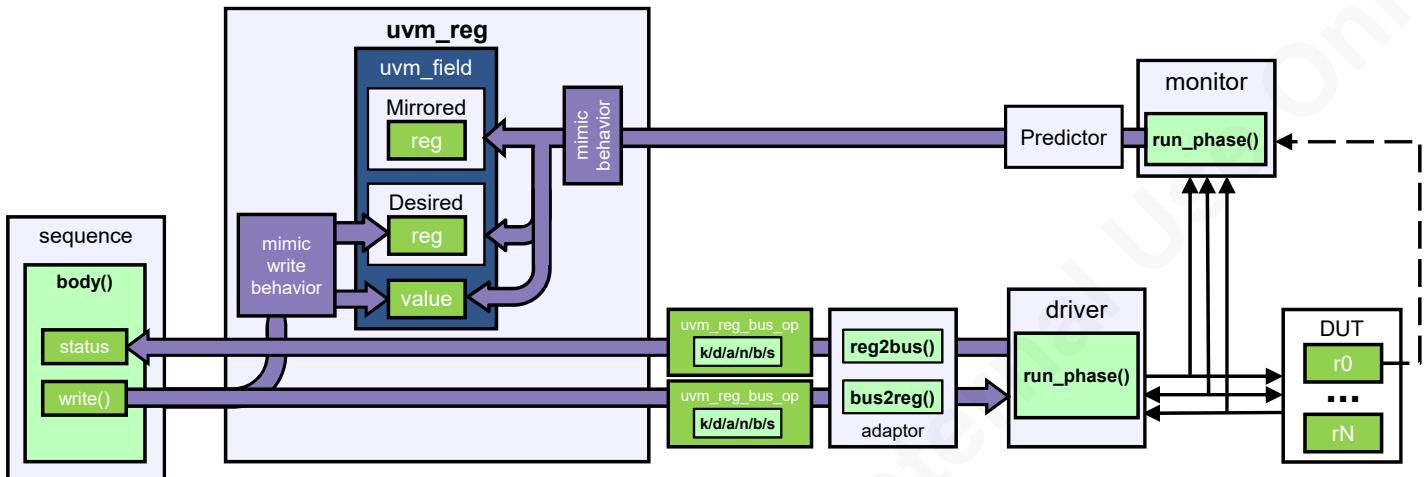


Within the `uvm_reg` class, the Mirrored content should always reflect the actual content of the DUT register. The Mirrored content can be updated implicitly or explicitly. In either case, it must reflect the actual value of the DUT register.

The Desired content is populated by the user. It reflects what the user desired value would be in DUT register. Think of it as a scratch pad for data manipulation before the content is used to populate the DUT register.

Register Frontdoor Write

- model.r0.write(status, value, [UVM_FRONTDOOR], .parent(this));



11-42

Sequence sets uvm_reg with value

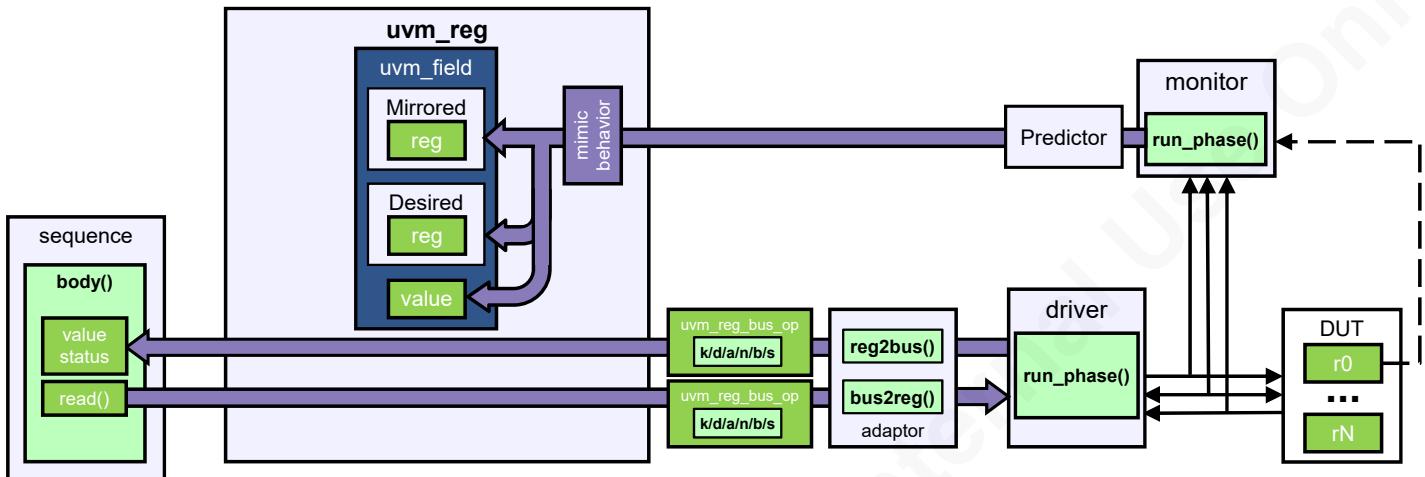
uvm_reg content is translated into bus transaction

Driver gets bus transaction and writes DUT register

Mirror can be updated either implicitly or explicitly

Register Frontdoor Read

- model.r0.read(status, value, [UVM_FRONTDOOR], .parent(this));



11-43

Sequence executes uvm_reg READ

uvm_reg READ is translated into bus transaction

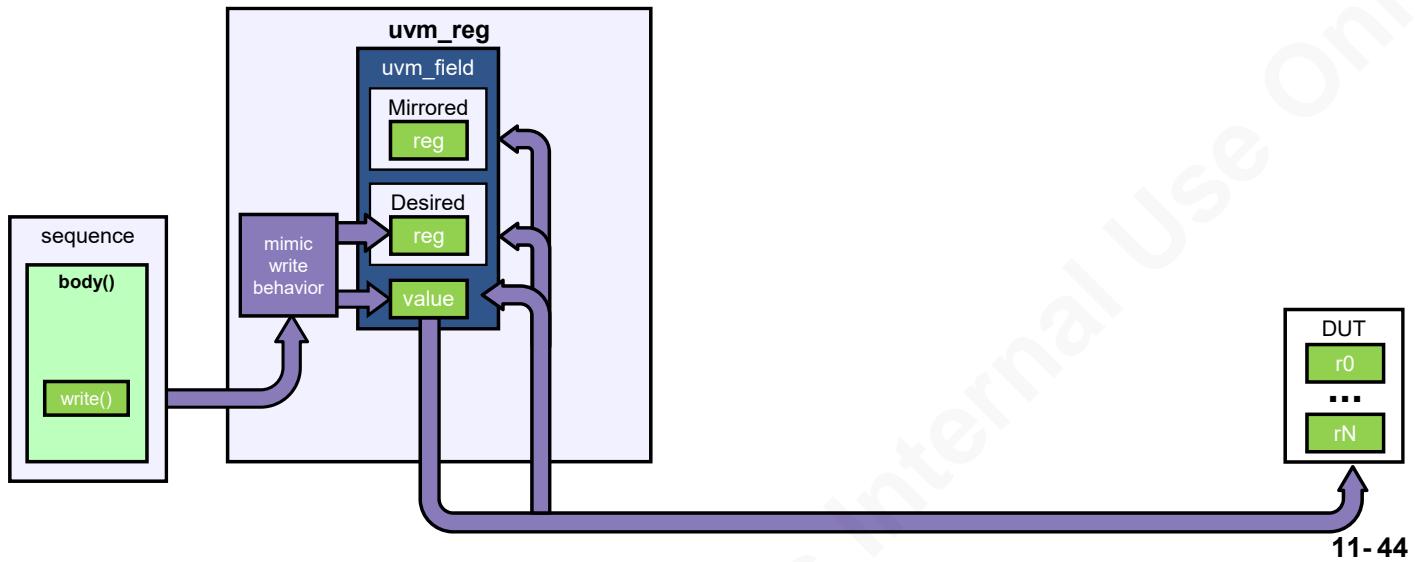
Driver gets bus transaction and read DUT register

Read value is translated into uvm_reg data and returned to sequence

Mirror updates

Register Backdoor Write

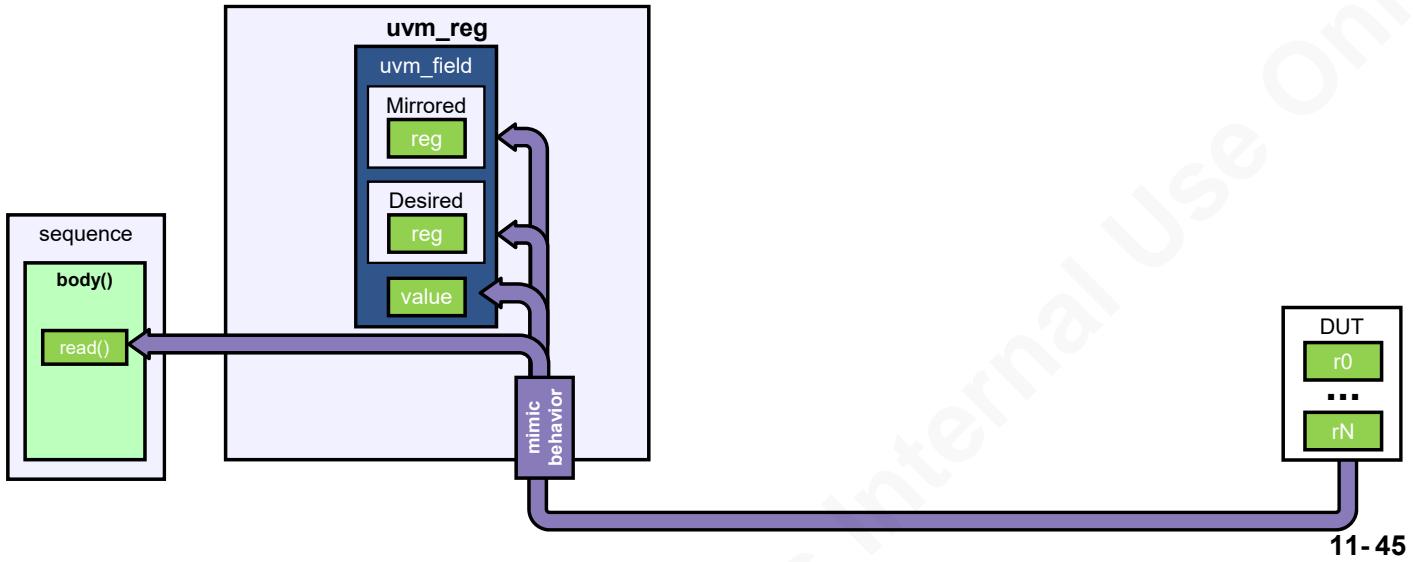
- `model.r0.write(status, value, UVM_BACKDOOR, .parent(this));`
 - uvm_reg uses DPI/XMR to set DUT register with value



Physical interface is bypassed.
Register behavior is mimicked.

Register Backdoor Read

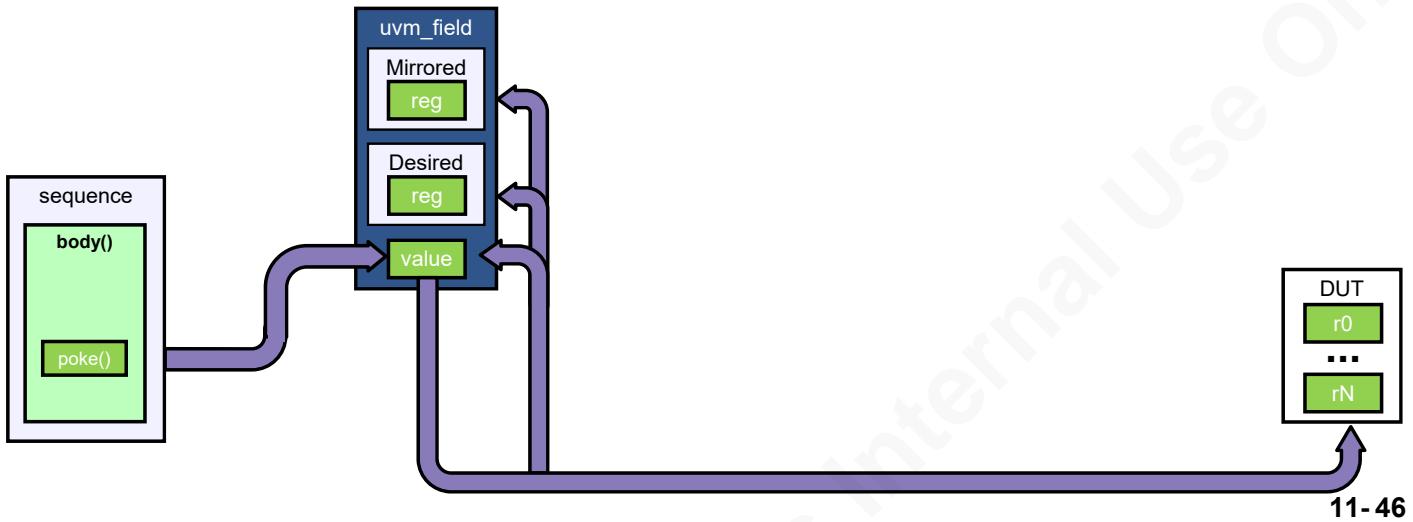
- `model.r0.read(status, value, UVM_BACKDOOR, .parent(this));`
 - uvm_reg uses DPI/XMR to get DUT register value



Physical interface is bypassed.
Register behavior is mimicked.

Register Backdoor Poke

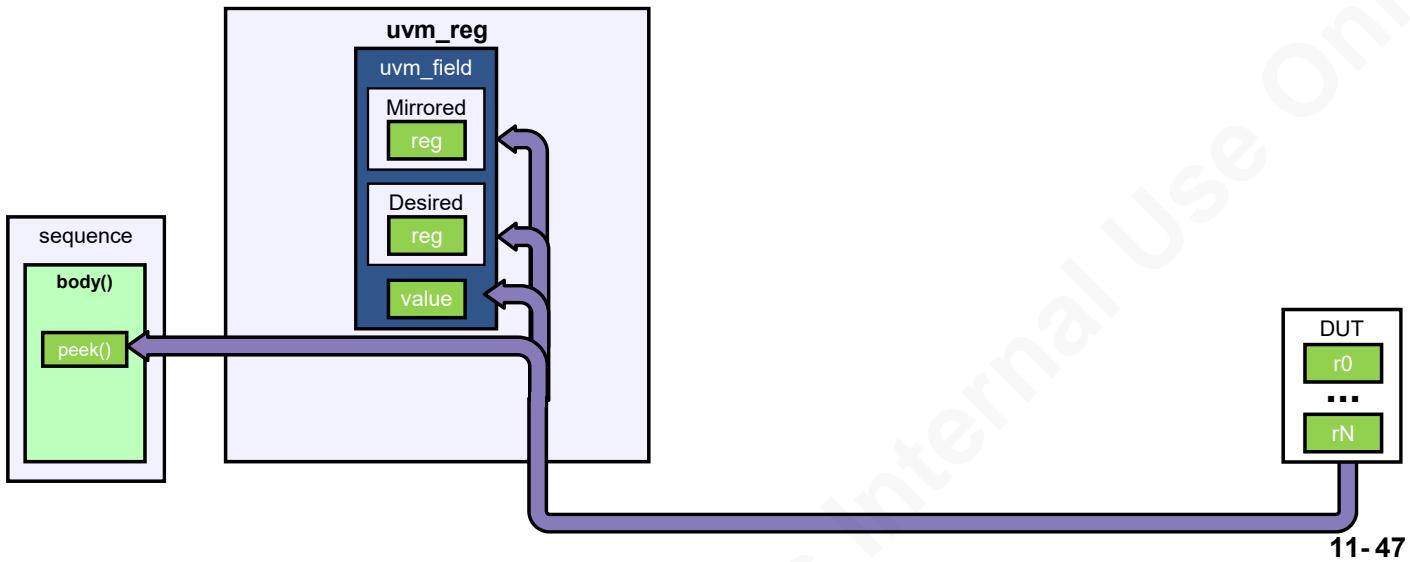
- `model.r0.poke(status, value, .parent(this));`
 - uvm_reg uses DPI/XMR to set DUT register with value as is



Physical interface is bypassed.
Register behavior is NOT mimicked.

Register Backdoor Peek

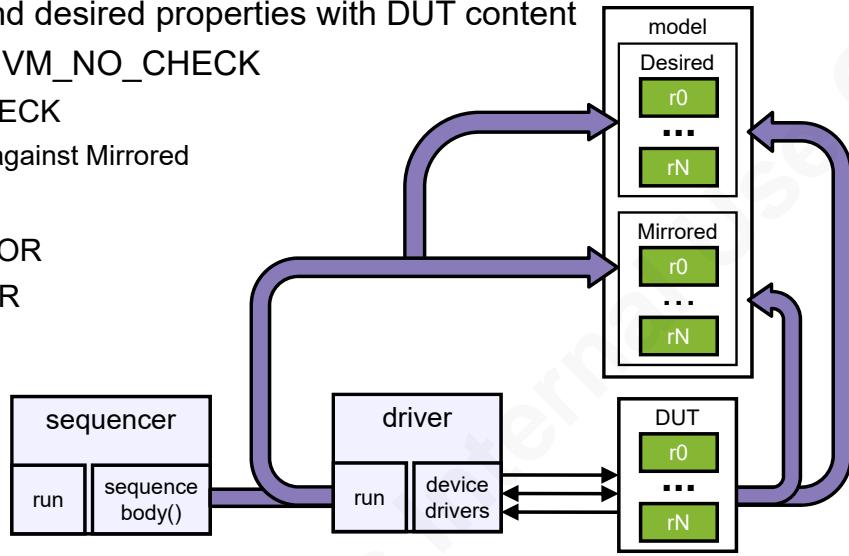
- **model.r0.peek(status, value, .parent(this));**
 - uvm_reg uses DPI/XMR to get DUT register value as is



Physical interface is bypassed.
Register behavior is NOT mimicked.

Mirrored & Desired Property Update

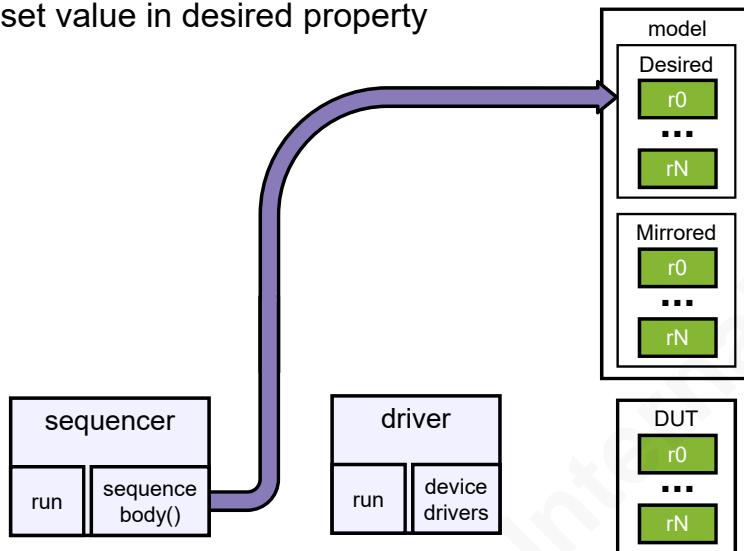
- `model.mirror(status, [check], [path], .parent(this));`
- `model.r0.mirror(status, [check], [path], .parent(this));`
 - Update mirrored and desired properties with DUT content
 - check defaults to UVM_NO_CHECK
 - ◆ If set to UVM_CHECK
 - Data compared against Mirrored
 - path can be
 - ◆ UVM_FRONTDOOR
 - ◆ UVM_BACKDOOR



11-48

UVM Register Desired Property Write

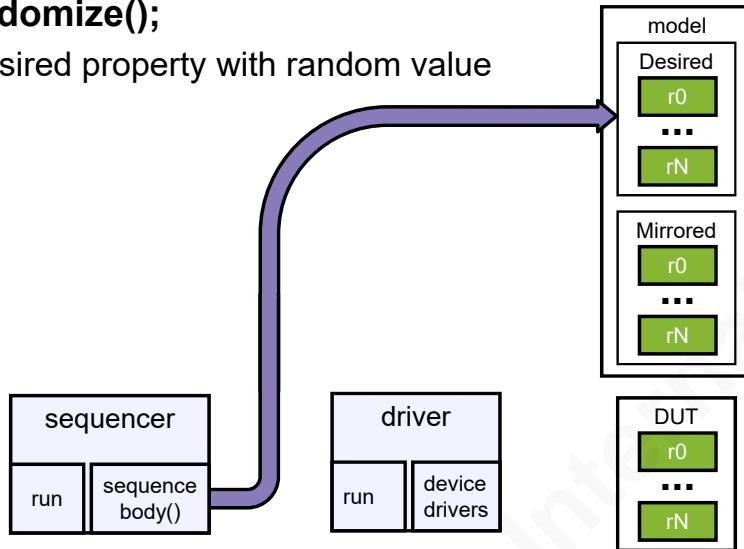
- ***model.r0.set(value);***
 - Set method set value in desired property



11-49

Randomize UVM Register Desired Property

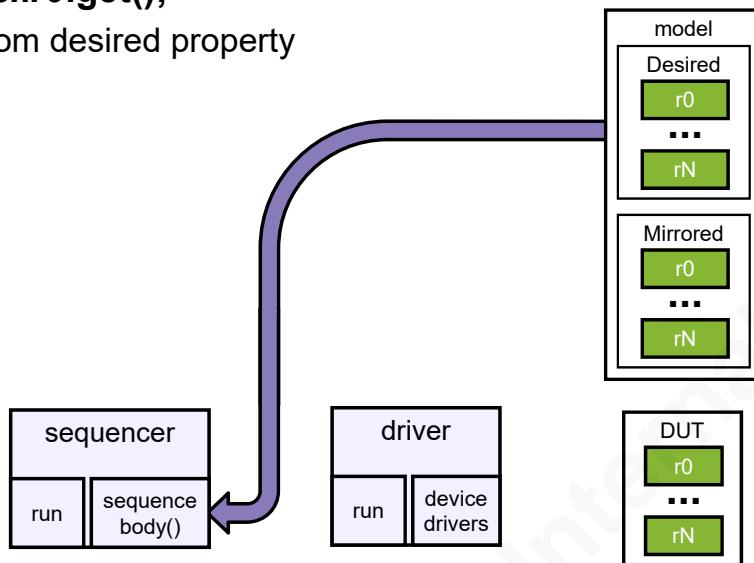
- `model.randomize();`
- `model.r0.randomize();`
 - Populate desired property with random value



11- 50

UVM Register Desired Property Read

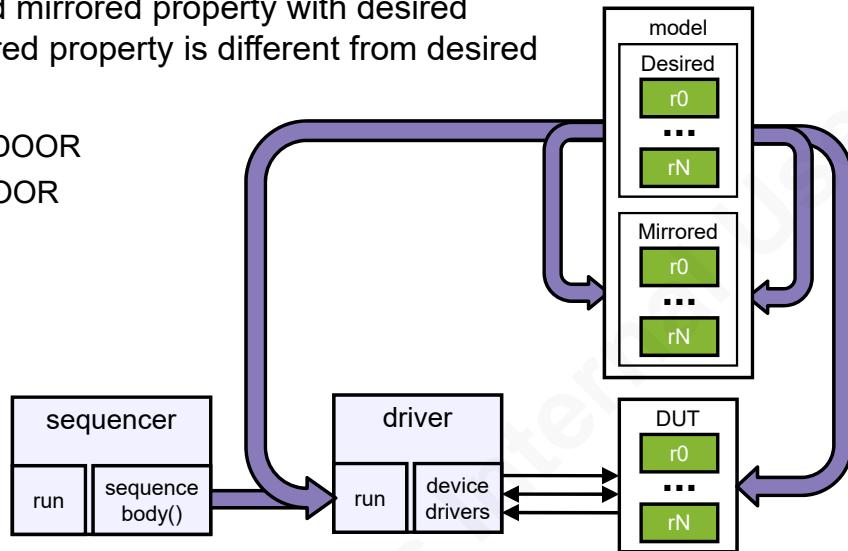
- **value = model.r0.get();**
 - Get value from desired property



11- 51

Mirrored & DUT Value Update

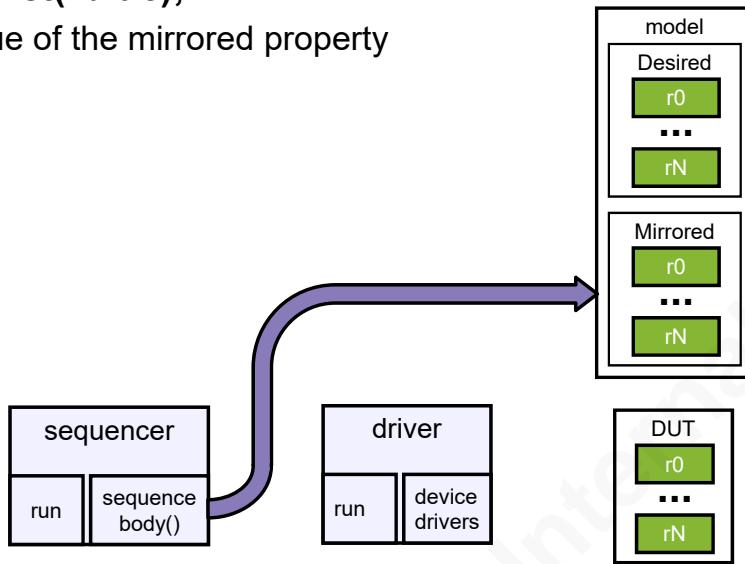
- `model.update(status, [path], .parent(this));`
- `model.r0.update(status, [path], .parent(this));`
 - Update DUT and mirrored property with desired property if mirrored property is different from desired
 - path can be
 - ◆ UVM_FRONTDOOR
 - ◆ UVM_BACKDOOR



11- 52

Writing to uvm_reg Mirrored Property

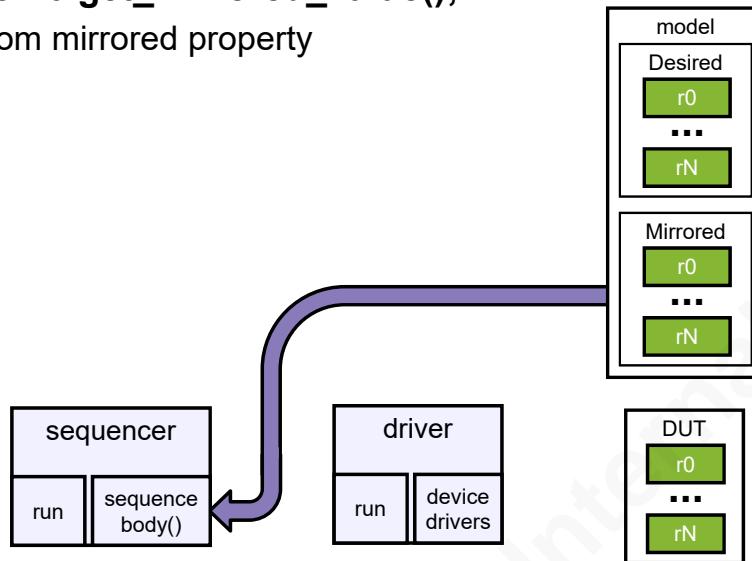
- **model.r0.predict(value);**
 - Sets the value of the mirrored property



11- 53

Reading uvm_reg Mirrored Property

- **value = model.r0.get_mirrored_value();**
 - Get value from mirrored property



11- 54

Appendix

UVM Register Modes

UVM Memory Modes

ralgen Options

UVM Backdoor

UVM Register Classes

UVM Register Callbacks

Changing Address Offsets of a Domain

Accessing DUT signals via DPI

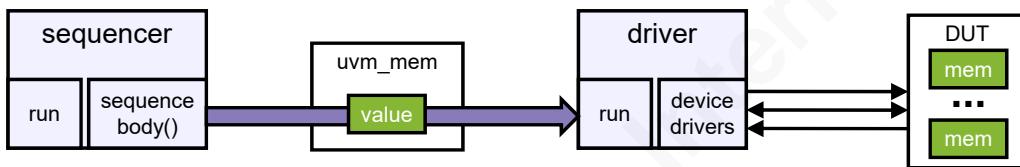
11- 55

Memory Frontdoor Write

- `model.mem.write(status, offset, value, [UVM_FRONTDOOR], .parent(this));`
- `model.mem.burst_write(status, offset, value[], [UVM_FRONTDOOR], .parent(this));`

Provide array populated with values to write to memory

- Sequence sets uvm_mem with value
- uvm_mem content is translated into bus transaction
- Driver gets bus transaction and writes DUT memory
- Memory is not mirrored



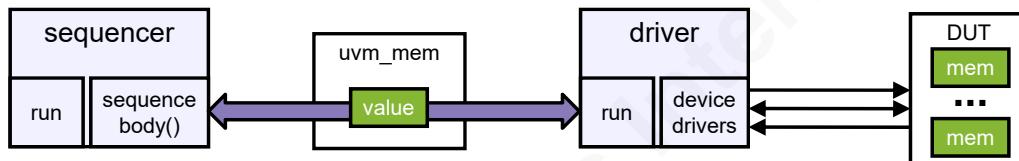
11- 56

Memory Frontdoor Read

- `model.mem.read(status, offset, value, [UVM_FRONTDOOR], .parent(this));`
- `model.mem.burst_read(status, offset, value[], [UVM_FRONTDOOR], .parent(this));`

Provide array sized to number of values to read from memory

- Sequence executes uvm_mem READ
- uvm_mem READ is translated into bus transaction
- Driver gets bus transaction and reads DUT memory
- Read value is translated to uvm_mem format and returned to sequence
- Memory is not mirrored



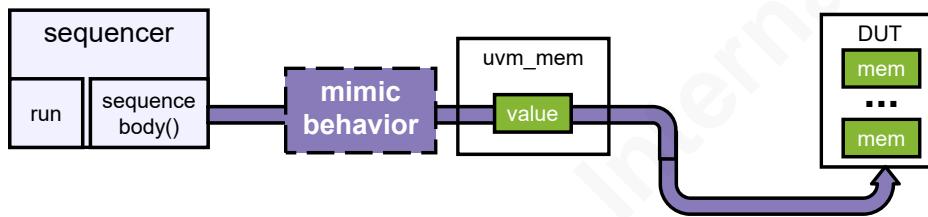
11-57

Memory Backdoor Write

- `model.mem.write(status, offset, value, UVM_BACKDOOR, .parent(this));`
- `model.mem.burst_write(status, offset, value[], UVM_BACKDOOR, .parent(this));`

Provide array populated with values to write to memory

- Sequence write to uvm_mem with value mimicking memory access policy (ro does not change memory value)
- uvm_mem uses DPI/XMR to set DUT memory with value
- Memory is not mirrored



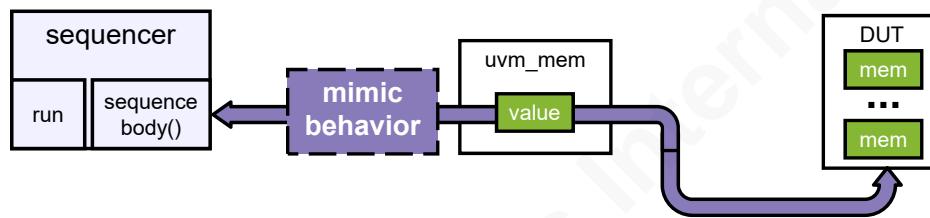
11- 58

Memory Backdoor Read

- `model.mem.read(status, offset, value, UVM_BACKDOOR, .parent(this));`
- `model.mem.burst_read(status, offset, value[], UVM_BACKDOOR, .parent(this));`

Provide array sized to number of values to read from memory

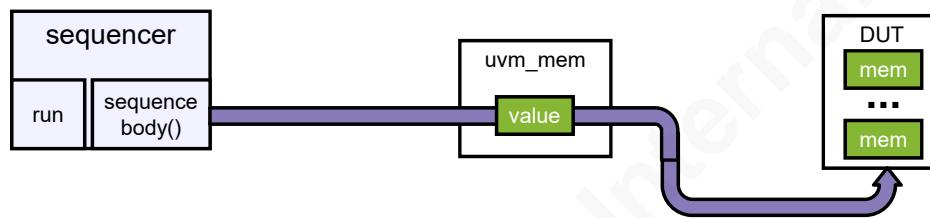
- Sequence executes uvm_mem READ
- uvm_mem uses DPI/XMR to get DUT memory value mimicking memory access policy (wo does not return memory value)
- Memory is not mirrored



11- 59

Memory Backdoor Poke

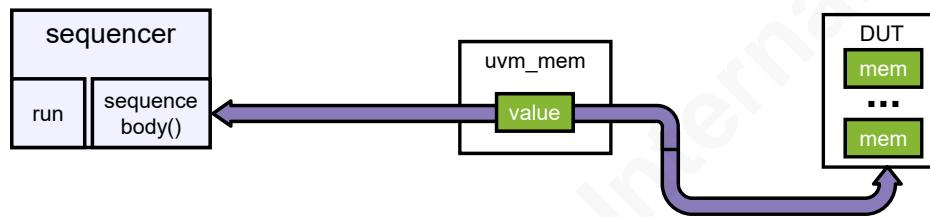
- **`model.mem.poke(status, offset, value, UVM_BACKDOOR, .parent(this));`**
 - Sequence writes to `uvm_mem` with value
 - ◆ Does not mimic memory access policy (ro memory WILL be modified)
 - `uvm_mem` uses DPI/XMR to set DUT memory with value
 - Memory is not mirrored



11-60

Memory Backdoor Peek

- **`model.mem.peek(status, offset, value, UVM_BACKDOOR, .parent(this));`**
 - Sequence executes uvm_mem PEEK
 - uvm_mem uses DPI/XMR to get DUT memory value
 - Memory is not mirrored



11-61

Appendix

UVM Register Modes

UVM Memory Modes

ralgen Options

UVM Backdoor

UVM Register Classes

UVM Register Callbacks

Changing Address Offsets of a Domain

Accessing DUT signals via DPI

11-62

ralgen Options: Common

```
ralgen [options] -uvm -t top_model file.ralf
```

■ Common options:

- -f file
 - ◆ Specify a file containing ralgen options
- -o fname
 - ◆ Specify output file name. Default is ral_<top_model>
- -c b | s | a | f | F
 - ◆ Generate functional coverage model
- -P
 - ◆ Generate model in separate packages

11- 63

For detailed ralgen usage, please consult UVM Register Abstraction Layer Generator User Guide in SolvNet.

ralgen Options: Advanced

- **Advanced options:**

- **-b**
 - ◆ Generate XMR-based (non-DPI) back-door access code
 - ◆ **-gen_vif_bkdr**
 - Generate back-door access with virtual interface
 - ◆ **-auto_mirror**
 - Generate code for automatic mirror update
- **-B**
 - ◆ Generate byte-base addressing

- **See uvm_ralgen_ug.pdf for other options**

- Or, execute: **ralgen -h**

11- 64

For detailed ralgen usage, please consult UVM Register Abstraction Layer Generator User Guide in SolvNet.

ralgen Address Granularity

```
register R {
    bytes 4;
    field C { bits 8; }
}
register S {
    bytes 4;
    field D { bits 8; }
}
register T {
    bytes 4;
    field A { bits 8; }
}
block BLK {
    bytes 4;
    register R;
    register S;
    register T;
}
```

```
regmodel = ral_block_BLK::type_id::create("regmodel", this);
regmodel.build();
regmodel.lock_model();
`uvm_info("ADDR_MAP", $sformatf("R addr = %0h", regmodel.R.get_address()), UVM_HIGH)
`uvm_info("ADDR_MAP", $sformatf("S addr = %0h", regmodel.S.get_address()), UVM_HIGH)
`uvm_info("ADDR_MAP", $sformatf("T addr = %0h", regmodel.T.get_address()), UVM_HIGH)
```

```
ralgen -uvm -t BLK file.ralf
```

```
UVM_INFO program.sv(11) @ 0: reporter [ADDR_MAP] R address = 0
UVM_INFO program.sv(12) @ 0: reporter [ADDR_MAP] S address = 1
UVM_INFO program.sv(13) @ 0: reporter [ADDR_MAP] T address = 2
```

```
ralgen -uvm -t BLK -B file.ralf
```

```
UVM_INFO program.sv(11) @ 0: reporter [ADDR_MAP] R address = 0
UVM_INFO program.sv(12) @ 0: reporter [ADDR_MAP] S address = 4
UVM_INFO program.sv(13) @ 0: reporter [ADDR_MAP] T address = 8
```

11-65

For detailed ralgen usage, please consult UVM Register Abstraction Layer Generator User Guide in SolvNet.

ralgen Functional Coverage

```
// host.ralf
register HOST_ID {
    field REV_ID {...}
    field CHIP_ID {...}
}

class ral_reg_HOST_ID extends uvm_reg; // other code not shown
uvm_reg_field REV_ID; uvm_reg_field CHIP_ID;
covergroup cg_bits ();
    option.per_instance = 1;
    REV_ID: coverpoint {m_data[7:0], m_is_read} iff(m_be) {
        wildcard bins bit_0_wr_as_0 = {9'b?????????00};
        wildcard bins bit_0_wr_as_1 = {9'b?????????10};
        ...
    }
endgroup
```

UVM
RAL
Classes

```
ralgen -c b -uvm -t dut_regmodel host.ralf
```

```
program automatic test; // other code not shown
initial begin
    uvm_reg::include_coverage("", UVM_CVR_ALL);
    run_test();
end
endprogram
class test_coverage extends
    virtual function void enelaboration_phase (uvm_phase phase);
        env.regmodel.set_coverage(UVM_CVR_ALL);
    endfunction
endclass
```

User must enable coverage with **include_coverage()** and **set_coverage()**

11-66

For detailed ralgen usage, please consult UVM Register Abstraction Layer Generator User Guide in SolvNet.

Appendix

UVM Register Modes

UVM Memory Modes

ralgen Options

UVM Backdoor

UVM Register Classes

UVM Register Callbacks

Changing Address Offsets of a Domain

Accessing DUT signals via DPI

11- 67

Optional: Backdoor Access

- **Two ways to generate the backdoor access:**
 - Via SystemVerilog Cross Module Reference (XMR)
 - Via SystemVerilog DPI call
- **Both allow register model to be part of SystemVerilog package**
- **XMR implementation is faster**
 - Requires user to compile one additional file and at compile-time provide top level path to DUT
 - VCS only
- **DPI implementation is slower**
 - No additional file is needed and top level path can be provided at run-time
 - Portable to other simulators

11- 68

Modified ralgen XMR Backdoor Access (1/3)

■ -b -gen_vif_bkdr option access register via Verilog XMR

```
// host.ralf
...
register HOST_ID (host_id) @'h0000;
    ralgen -b -gen_vif_bkdr -uvm -t host_regmodel host.ralf

// ral_host_regmodel_intf.sv
interface ral_host_regmodel_intf; // other code not shown
    initial
        uvm_resource_db#(virtual ral_host_regmodel_intf)::set("*",
            "uvm_reg_bkdr_if", interface::self());
    task ral_host_regmodel_HOST_ID_bkdr_read(uvm_reg_item rw);
        rw.value[0] = `HOST_REGMODEL_TOP_PATH.host_id;
    endtask
endinterface
```

Access method uses XMR

Self stored into database
(VCS only – see note)

vcs ral_host_regmodel_intf.sv +define+HOST_REGMODEL_TOP_PATH=router_test_top.dut ...

Interface must be compiled XMR path from harness to DUT must be specified

11-69

Caution:

`interface::self()` is NOT an IEEE defined mechanism. Other simulators may not support this call.

If this is not desired, there are three ways to make it fully IEEE compliant:

One – Comment out the line in question and manually instantiate and store the instance into the resource_db in the top module block.

Example:

```
module router_test_top;
    ral_host_regmodel_intf host_intf();
    initial begin
        uvm_resource_db#(virtual ral_host_regmodel_intf)::set(
            "*", "uvm_reg_bkdr_if", host_if);
    end
endmodule
```

Two – Use the DPI mechanism as shown in a couple more slide.

Three – Use the true XMR mechanism shown in appendix (11-96). The issue with this alternative is that the generated code cannot be embedded inside a package. (true XMR is not allowed in SystemVerilog packages)

ralgen XMR Backdoor Access (2/3)

- Generated backdoor class uses interface in to access registers

```
// ral_host_regmodel.sv
class ral_reg_host_regmodel_HOST_ID_bkdr extends uvm_reg_backdoor;
// other code not shown
virtual ral_host_regmodel_intf __reg_vif;
function new(string name);
super.new(name);
uvm_resource_db#(virtual ral_host_regmodel_intf)::read_by_name(get_full_name(),
"uvm_reg_bkdr_if", __reg_vif);
endfunction

virtual task read(uvm_reg_item rw);
do_pre_read(rw);
__reg_vif.ral_host_regmodel_HOST_ID_bkdr_read(rw);
rw.status = UVM_IS_OK;
do_post_read(rw);
endtask
endclass
```

Backdoor access class is generated by **ralgen**

XMR encapsulated in package-able interface

Interface retrieved via resource database (see note)

Register access method make use of interface access method

11-70

ralgen XMR Backdoor Access (3/3)

■ XMR access method flow:

```
regmodel.HOST_ID.read(status, data, UVM_BACKDOOR, .parent(this));  
task uvm_reg::read(...); ...  
  XreadX(status, value, path, map, parent, prior, extension, fname, lineno);  
endtask  
task uvm_reg::XreadX(...); ...  
  do_read(rw);  
endtask  
task uvm_reg::do_read(uvm_reg_item rw); ...  
  case (rw.path)  
    UVM_BACKDOOR: begin  
      uvm_reg_backdoor bkdr = get_backdoor();  
      if (bkdr != null) bkdr.read(rw);  
      else backdoor_read(rw);  
    ...  
    task ral_reg_host_regmodel_HOST_ID_bkdr::read(uvm_reg_item rw);  
      do_pre_read(rw);  
      __reg_vif.ral_host_regmodel_HOST_ID_bkdr_read(rw);  
      rw.status = UVM_IS_OK;  
      do_post_read(rw);  
    endtask
```

Uses
ralgen
generated
interface

Uses backdoor class
generated by **ralgen**

11-71

ralgen DPI Backdoor Access (1/2)

- No additional switch on the ralgen or vcs command

```
// host.ralf
...
register HOST_ID (host_id) @'h0000;
ralgen -uvm -t host_regmodel host.ralf
```

- No specialized backdoor interface generated
- Need to add top level path at run-time via database:

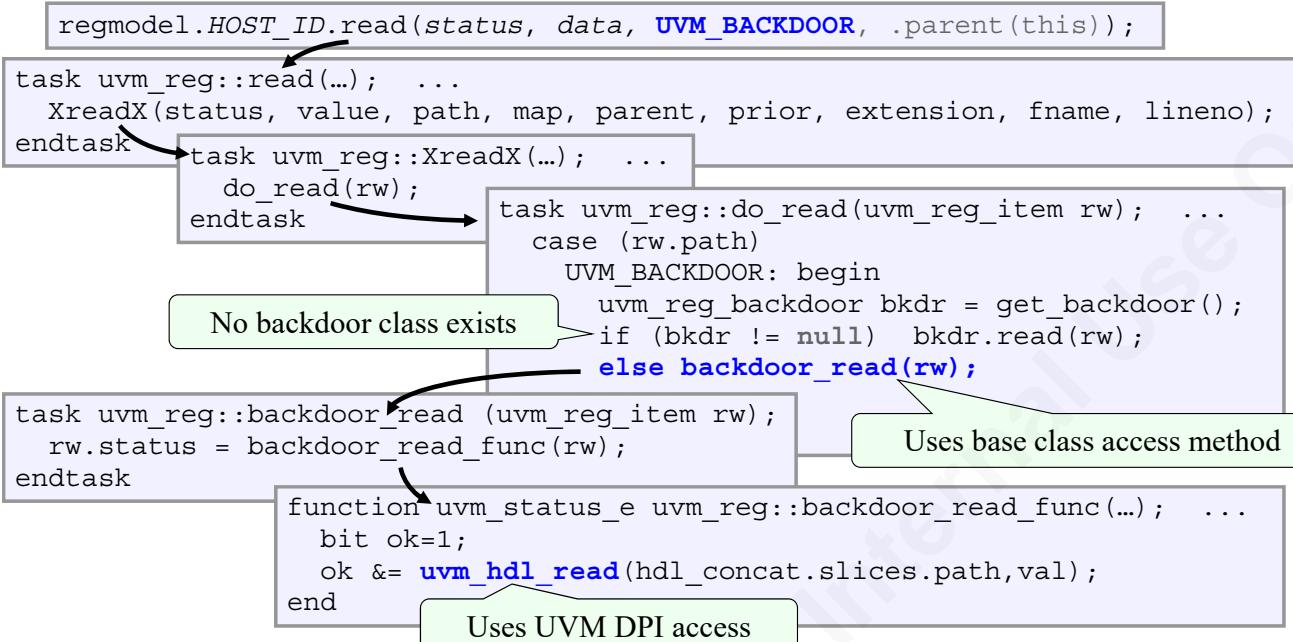
```
virtual function void test_ral_base::build_phase(uvm_phase phase); ...;
  uvm_resource_db #(string)::set("hdl_path","router_test_top.dut");
endfunction

function void host_env::build_phase(uvm_phase phase);
  if (!uvm_resource_db#(string)::read_by_type("hdl_path",hdl_path))
    regmodel.set_hdl_path_root(hdl_path);
  else `uvm_fatal("host_regmodel", "top path is not set!");
endfunction
```

11-72

ralgen DPI Backdoor Access (2/2)

DPI access flow



11-73

Appendix

UVM Register Modes

UVM Memory Modes

ralgen Options

UVM Backdoor

UVM Register Classes

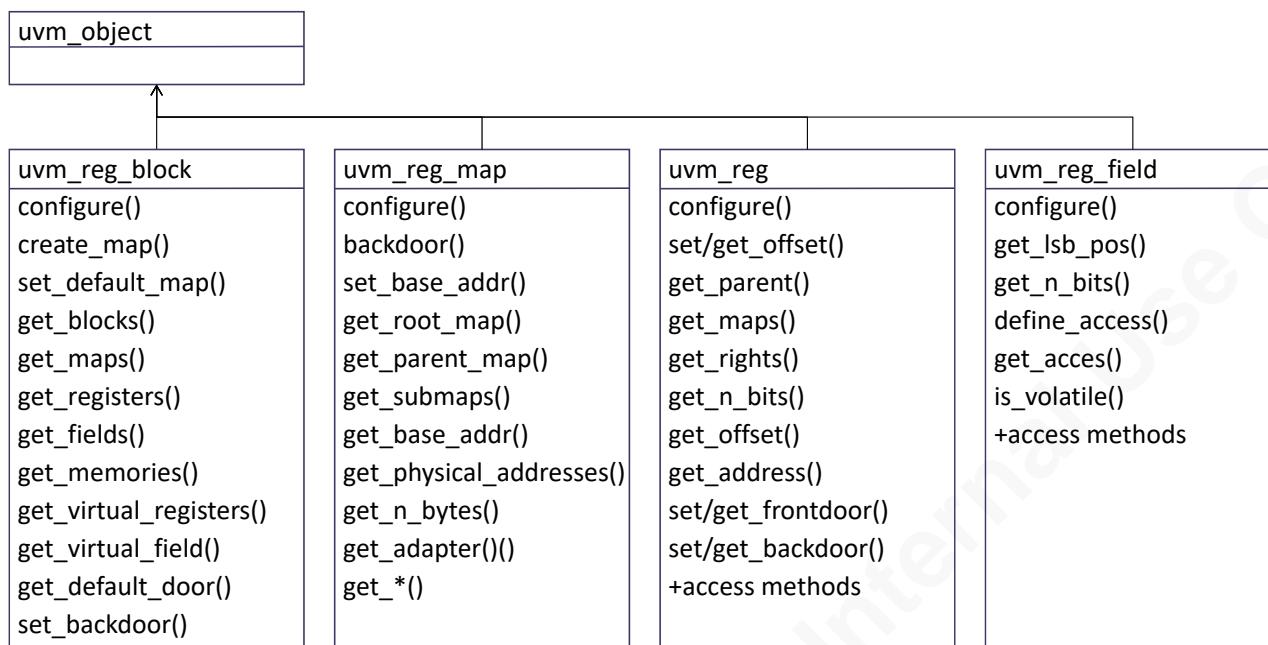
UVM Register Callbacks

Changing Address Offsets of a Domain

Accessing DUT signals via DPI

11-74

UVM Register Base Class Commonly Used Content



11-75

UVM Register Classes: uvm_reg_bus_op & uvm_reg_item

```
'define UVM_REG_ADDR_WIDTH 64
`define UVM_REG_DATA_WIDTH 64

typedef bit unsigned [`UVM_REG_DATA_WIDTH-1:0] uvm_reg_data_t ;
typedef bit unsigned [`UVM_REG_ADDR_WIDTH-1:0] uvm_reg_addr_t ;

typedef struct {
    uvm_reg_addr_t      addr;
    uvm_reg_data_t      data;
    uvm_access_e        kind;      // Can be: UVM_READ, UVM_WRITE, UVM_BURST_READ, UVM_BURST_WRITE
    int                 n_bits;   // Number of bits of <uvm_reg_item::value> being transferred by this transaction.
    uvm_reg_byte_en_t   byte_en;  // Enable byte lanes on the bus. Meaningful only when the bus supports byte enables
    uvm_status_e         status;   // Result can be: UVM_IS_OK, UVM_HAS_X, UVM_NOT_OK.
} uvm_reg_bus_op;
```

Caution: Not complete.
Only key members are shown.

```
class uvm_reg_item extends uvm_sequence_item;
    rand uvm_access_e      kind;
    rand uvm_reg_data_t    value[];
    rand uvm_reg_addr_t    offset;
    uvm_status_e           status;
    uvm_reg_map             map;
endclass
```

11-76

UVM Register Classes: uvm_reg_field

```
class uvm_reg_field extends uvm_object;
  rand uvm_reg_data_t value;
  local uvm_reg_data_t m_mirrored;
  local uvm_reg_data_t m_desired;
  function void configure(...);
  function void reset(...);
  task write(...);
  task read (...);
  task poke (...);
  task peek (...);
  function void           set      (...);
  function uvm_reg_data_t get      (...);
  function bit            predict   (...);
  task                  mirror   (...);
  function uvm_reg_data_t get_mirrored_value(...);
  function void uvm_reg_field::pre_randomize(); value = m_desired; endfunction
  function void uvm_reg_field::post_randomize(); m_desired = value; endfunction
endclass
```

11-77

UVM Register Classes: uvm_reg

```
class uvm_reg extends uvm_object;
protected bit          m_maps[uvm_reg_map];
protected uvm_reg_field m_fields[$];
local bit              m_read_in_progress;
local bit              m_write_in_progress;
protected bit          m_update_in_progress;

function void configure (...);
function void add_field (...);
function void add_map   (...);
function void add_hdl_path (...);
function void          set(...);
function uvm_reg_data_t get(...);
function bit            predict        (...);
function uvm_reg_data_t get_mirrored_value(...);
function void          reset(...);

task write   (...);
task read    (...);
task poke    (...);
task peek    (...);
task update  (...);
task mirror  (...);

endclass
```

11-78

UVM Register Classes: uvm_reg_map

```
class uvm_reg_map extends uvm_object;
  local uvm_reg_addr_t      m_base_addr;
  local uvm_reg_adapter     m_adapter;
  local uvm_reg_addr_t      m_submaps[uvm_reg_map];
  local uvm_reg_map_info    m_regs_info[uvm_reg];
  local uvm_reg_map_info    m_mems_info[uvm_mem];
  local uvm_reg              m_regs_by_offset[uvm_reg_addr_t];
  local uvm_mem              m_mems_by_offset[uvm_reg_map_addr_range];
  static local uvm_reg_map  m_backdoor;
  function void configure   (...);
  function void add_reg     (...);
  function void add_mem     (...);
  function void add_submap  (...);
  function void set_sequencer (...);
  function void set_submap_offset (...);
  function void set_base_addr (...);
  function void set_auto_predict (...);
  function void reset       (...);
  task          do_write    (...);
  task          do_read     (...);
  task          do_bus_write (...);
  task          do_bus_read  (...);
endclass
```

11-79

UVM Register Classes: uvm_reg_block

```
class uvm_reg_block extends uvm_object;
    uvm_reg_map      default_map;
local int unsigned   blks[uvm_reg_block];
local int unsigned   regs[uvm_reg];
local int unsigned   vregs[uvm_vreg];
local int unsigned   mems[uvm_mem];
local bit            maps[uvm_reg_map];
function uvm_reg_map create_map(...);
function void configure (...);
function void build   (...);
function void add_block (...);
function void add_map  (...);
function void add_reg   (...);
function void add_vreg  (...);
function void add_mem   (...);
function void lock_model();
function void reset     (...);
task update(...);
task mirror(...);
function void set_backdoor (...);
function void add_hdl_path (...);
function void set_hdl_path_root (...);
endclass
```

11-80

Appendix

UVM Register Modes

UVM Memory Modes

ralgen Options

UVM Backdoor

UVM Register Classes

UVM Register Callbacks

Changing Address Offsets of a Domain

Accessing DUT signals via DPI

11-81

RAL Class Key Callback Members

Caution: Simplified code for illustration. Most code left off.

```
// See class reference document and source code files
// for actual code
virtual class uvm_reg extends uvm_object;
    `uvm_register_cb(uvm_reg, uvm_reg_cbs)
    virtual task pre_write(uvm_reg_item rw); endtask
    virtual task post_write(uvm_reg_item rw); endtask
    virtual task pre_read(uvm_reg_item rw); endtask
    virtual task post_read(uvm_reg_item rw); endtask
endclass: uvm_reg
task uvm_reg::write(...);
    set(value);
    do_write(rw); // See next page
endtask
```

- **Two ways to implement callbacks**

- Simple callback – extend uvm_reg class
- UVM callback – extend uvm_reg_cbs class then register cb

11-82

RAL Class Key Callback Members

```
task uvm_reg::do_write(uvm_reg_item rw);
    uvm_reg_cb_iter cbs = new(this);
    foreach (m_fields[i]) begin
        uvm_reg_field_cb_iter cbs = new(m_fields[i]);
        uvm_reg_field f = m_fields[i];
        f.pre_write(rw);
        for (uvm_reg_cbs cb=cbs.first(); cb!=null; cb=cbs.next())
            cb.pre_write(rw);
    end
    pre_write(rw);
    for (uvm_reg_cbs cb=cbs.first(); cb!=null; cb=cbs.next())
        cb.pre_write(rw);
    // EXECUTE WRITE...
    for (uvm_reg_cbs cb=cbs.first(); cb!=null; cb=cbs.next())
        cb.post_write(rw);
    post_write(rw);
    foreach (m_fields[i]) begin
        uvm_reg_field_cb_iter cbs = new(m_fields[i]);
        uvm_reg_field f = m_fields[i];
        for (uvm_reg_cbs cb=cbs.first(); cb!=null; cb=cbs.next())
            cb.post_write(rw);
        f.post_write(rw);
    end
endtask: do_write
```

Caution: Simplified code for illustration only

11-83

Appendix

UVM Register Modes

UVM Memory Modes

ralgen Options

UVM Backdoor

UVM Register Classes

UVM Register Callbacks

Changing Address Offsets of a Domain

Accessing DUT signals via DPI

11- 84

Changing the Address offsets of a Domain

- **set_base_addr()** method allows user to modify the base address of a uvm register map
 - Can be called before or after **lock_model()**
 - If called after the model is locked, the address will be re-initialized

```
virtual function void build_phase(uvm_phase phase);
    ...; // other code left off
    model.build();
    model.APBD.set_base_addr('h2000_0000);
    model.WSHD.set_base_addr('h4002_0000);
    model.lock_model();
endfunction
```

11-85

It's possible that you don't want to specify base addresses in RALF as these might be subject to change (depending on chip mode, project, etc). Sometimes it is easier to specify only relative offsets in RALF, and then specify the base addresses for each domain in the testbench.

Appendix

UVM Register Modes

UVM Memory Modes

ralgen Options

UVM Backdoor

UVM Register Classes

UVM Register Callbacks

Changing Address Offsets of a Domain

Accessing DUT signals via DPI

11-86

Direct DUT Signal Access

■ From UVM `uvm_hdl.svh` file

```
// For all functions, returns 1 if the call succeeded, 0 otherwise.  
// uvm_hdl_deposit: sets the given HDL ~path~ to the specified ~value~.  
function int uvm_hdl_deposit(string path, uvm_hdl_data_t value);  
// uvm_hdl_read: gets the value at the given ~path~.  
function int uvm_hdl_read(string path, output uvm_hdl_data_t value);  
// uvm_hdl_force: forces the ~value~ on the given ~path~.  
function int uvm_hdl_force(string path, uvm_hdl_data_t value);  
// uvm_hdl_force_time: forces the ~value~ on the given ~path~ for the specified amount of  
// ~force_time~. If ~force_time~ is 0, <uvm_hdl_deposit> is called.  
task uvm_hdl_force_time(string path, uvm_hdl_data_t value, time force_time=0);  
// uvm_hdl_release_and_read: releases a value previously set with <uvm_hdl_force>.  
// ~value~ is set to the HDL value after the release.  
function int uvm_hdl_release_and_read(string path, inout uvm_hdl_data_t value);  
// uvm_hdl_release: releases a value previously set with <uvm_hdl_force>.  
function int uvm_hdl_release(string path);
```

11-87

This page was intentionally left blank

Agenda

DAY

3

9 UVM Advanced Sequence/Sequencer

10 UVM Phasing and Objections



11 UVM Register Abstraction Layer (RAL)

12 Summary



Key Elements of UVM

- Methodology
- Scalable architecture
- Standardized component communication
- Customizable component phase execution
- Flexible components configuration
- Flexible component search & replace
- Reusable register abstraction

12-2

UVM Methodology Guiding Principles

- **Top-down implementation methodology**

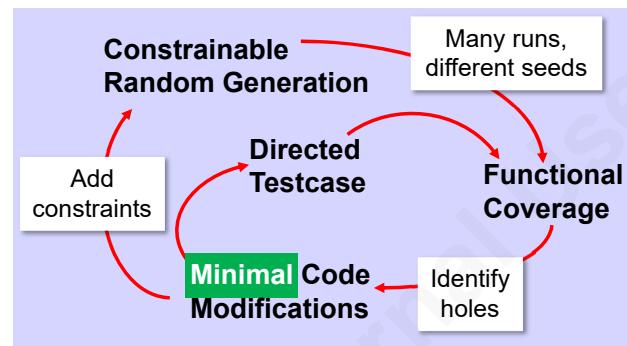
- Emphasizes “Coverage Driven Verification”

- **Maximize design quality**

- More testcases
- More checks
- Less code

- **Approaches**

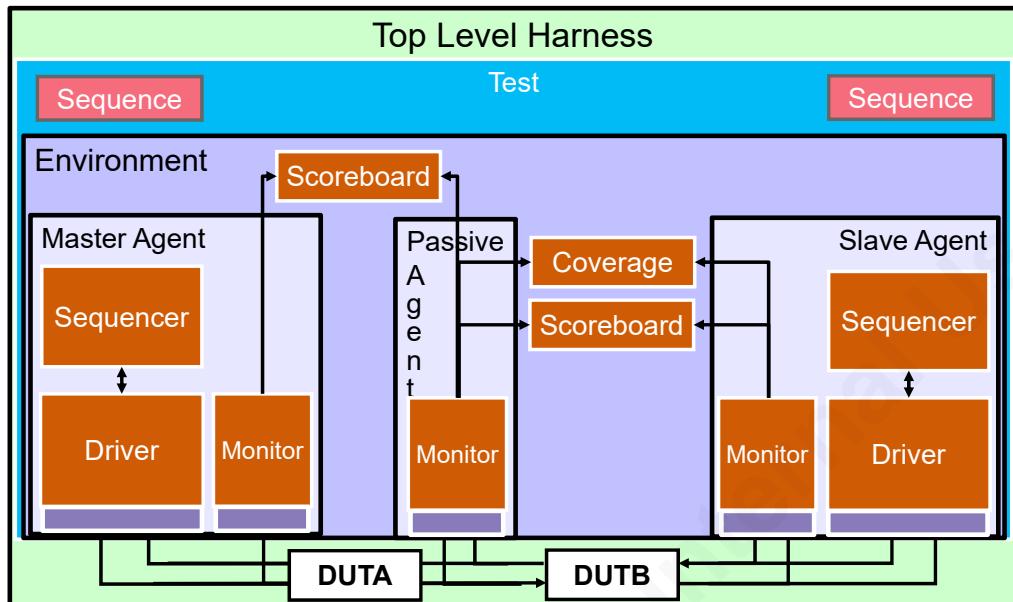
- Reuse
 - ◆ Across tests
 - ◆ Across blocks
 - ◆ Across systems
 - ◆ Across projects
- One verification environment, many tests
- Minimize test-specific code



12-3

Scalable Architecture

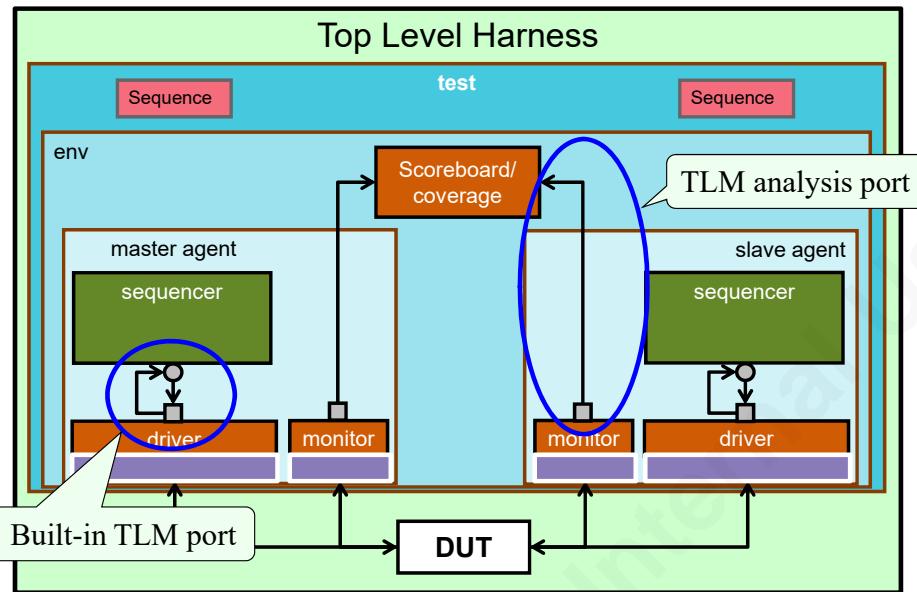
- Interface based agent enables block to system reuse



12-4

Standardized Component Communication

■ TLM, TLM 1.0, TLM 2.0

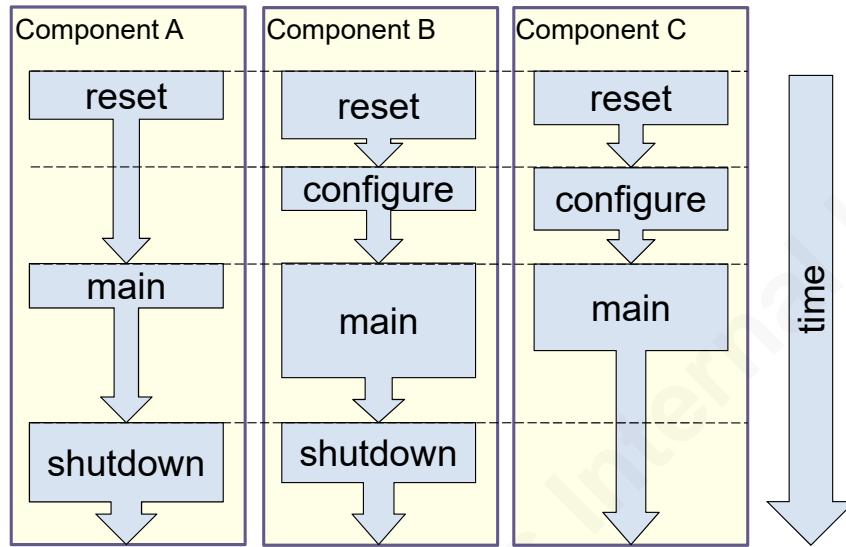


12-5

Customizable Component Phase Execution

■ Component phases are synchronized

- Ensures correctly organized configuration and execution
- Can be customized



12-6

Flexible Components Configuration (1/2)

- **uvm_config_db#(_type) ::get(...)**

```
class driver extends ...; // simplified code
    virtual router_io vif;
    function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        if (!uvm_config_db#(virtual router_io)::get(this, "", "vif", vif))
            `uvm_fatal("CFGERR", "Driver DUT interface not set");
    endfunction
endclass
```

- **uvm_config_db#(_type) ::set(...)**

```
class agent extends ...; // simplified code
    virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        uvm_config_db#(virtual router_io)::set(this, "*", "vif", vif);
    endfunction
endclass
```

12-7

Flexible Components Configuration (2/2)

- Sequence can be configured to be implicitly executed

- `uvm_config_db#(...)::set(...)` // objection must be embedded in sequence

```
class reset_sequence extends uvm_sequence#(reset_tr); // other code not shown
    function new(string name="reset_sequence");
        super.new(name);
        set_automatic_phase_objection(1); // UVM-1.2 & IEEE UVM Only
    endfunction
    virtual task body();
        `uvm_do(req);
    endtask
endclass
```

```
class router_env extends uvm_env; // simplified code
    virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        uvm_config_db#(uvm_object_wrapper)::set(this, "agt.sqr.reset_phase",
            "default_sequence", reset_sequence::get_type());
    endfunction
endclass
```

12-8

Flexible Component Search & Replace

```
class test_new extends test_base; ...
  `uvm_component_utils(test_new)
  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    set_inst_override_by_type("env.agt", agent::get_type(),
      new_agt::get_type());
  endfunction
endclass
```

Use component hierarchical path
to execute component overrides

```
class environment extends uvm_env;
  ...
  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    agt = agent::type_id::create("agt", this);
  endfunction
endclass
```

```
class new_agt extends agent;
  `uvm_component_utils(new_agt)
  function new(string name, uvm_component parent);
    virtual task class_task(...);
      // modified component functionality
    endtask
  endclass
```

Simulate with:
`simv +UVM_TESTNAME=test_new`

`create()` used to
construct component

Modify operation



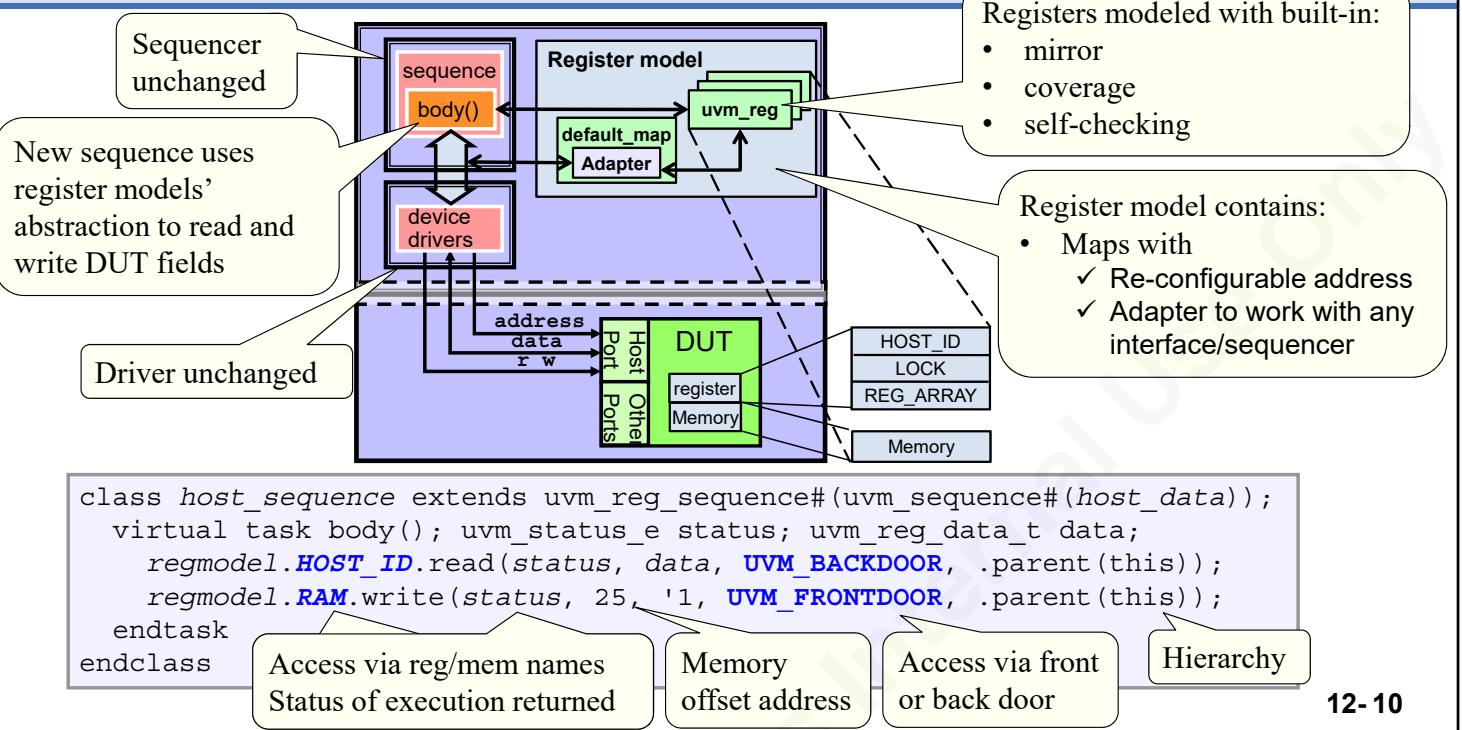
12-9

An important concept in UVM is that your testbench classes such as the environment, agent, drivers, etc, are written once at the start of the project, and modified very rarely if ever. This means that existing tests won't break when someone checks in a new version of a component. You can still change the behavior of components using "hooks" such as the UVM factory to change the behavior of a testbench, without editing existing code.

The UVM factory can build default components such as agents and drivers, but it can also allow you to replace an existing component with a derived one.

In this case the `agent` class has been replaced by `new_agt`. At the top level, the test tells the UVM factory to override the `agent` class with `new_agt`. When the environment constructs the `agt` object, it actually gets an object of `new_agt` class.

Standardized Register Abstraction



12-10

UVM Command Line Options

```
+uvm_set_verbosity=<comp>,<id>,<verbosity>,<phase>
+uvm_set_verbosity=<comp>,<id>,<verbosity>,time,<time>
+uvm_set_action=<comp>,<id>,<severity>,<action>
+uvm_set_severity=<comp>,<id>,<current severity>,<new severity>
+uvm_set_inst_override=<req_type>,<override_type>,<full_inst_path>
+uvm_set_type_override=<req_type>,<override_type>[,<replace>]
+uvm_set_config_int=<comp>,<field>,<value>
+uvm_set_config_string=<comp>,<field>,<value>
+uvm_set_default_sequence=<seqr>,<phase>,<type>
```

12-11

+uvm_set_default_sequence option is not available in UVM-1.1.

Getting Help

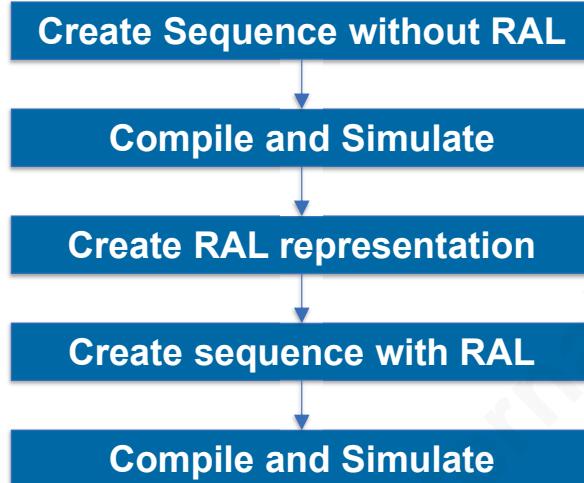
- **Code examples:**
 - \$VCS_HOME/doc/examples/uvm
- **Solvnet:**
 - <https://solvnet.synopsys.com/>
- **VCS support:**
 - vcs_support@synopsys.com
- **Synopsys verification video & SNUG:**
 - <https://www.synopsys.com/support/training/ces-training-videos-2016.html>
 - <https://www.youtube.com/user/synopsys>
 - www.snug-universal.org

12-12

Lab 6: Implement RAL

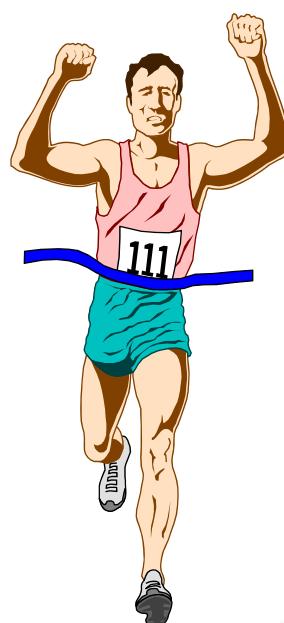


60 minutes



12-13

That's all Folks!



12-14

Customer Support



Synopsys Customer Education Services

© 2019 Synopsys, Inc. All Rights Reserved

20181001

Synopsys Support Resources

■ Build a solid foundation:

Hands-on training for Synopsys tools and methodologies

<https://synopsys.com/support/training.html>

- Workshop Schedule and Registration
- Download Labs (SolvNet ID required)

■ Drill down to areas of interest:

SolvNet online support

<https://solvnet.synopsys.com>

- Online technical information and access to support resources
- Documentation & Media

■ Ask an Expert:

Synopsys Support Center

<https://onlinecase.synopsys.com>

Training & Education

Hands-on training and education for Synopsys tools and methodologies



Learn from our experts who know Synopsys tools and industry best practices better than anyone else.

- Learn how to get the most out of your Synopsys tools
- Flexible options for learning online or in the classroom
- Tailor the curriculum to meet your requirements

Ready to get started?

Browse our training and education curriculum by product or service:

TRAINING COURSES

eLearning FreeView >	Physical Implementation >	RTL Synthesis >
Sign-Off >	Verification >	FPGA Design >
Software Security & Quality >	Optical Design >	DFM >

<https://training.synopsys.com>

CS - 2

SolvNet Online Support

- Immediate access to the latest technical information
- Product Update Training
- Methodology Training
- Thousands of expert-authored articles, Q&As, scripts and tool tips
- [Open a Support Center Case](#)
- Release information
- Online documentation
- License keys
- Electronic software downloads
- Synopsys announcements (latest tool, event and product information)

The screenshot shows the SolvNet® website interface. At the top, there's a navigation bar with links for Documentation, Support, Downloads, Training, Methodology, and My Profile. Below the navigation is a search bar with options for 'SEARCH', 'SELECT TYPE' (set to 'Articles and Documentation'), and 'Go' and 'Advanced' buttons. There are also links for 'Autocomplete ON', 'Search Help', and 'Search IP'. On the left, there are sections for 'New/Updated Articles' and 'Saved Articles'. The main content area displays a list of search results, each with a title and a date. Some titles include links like 'How to Find Scan Data Pins and Export to a Pin Path File (04-17-2017)', 'How to Check for Missing Generated Clock Definitions on Divider Circuits (04-17-2017)', and 'How to Select a Portion of a Pattern Set and Evaluate the New Test Coverage (04-12-2017)'. To the right of the search results, there are several sidebar boxes: 'SNUG Session Recordings' (listing sessions from Silicon Valley and Austin), 'Former Arteris Products' (using Synopsys Support Channels), 'Laker-Verdi Support Forum' (with a 'Sign Up Now' button), 'CODE V, LightTools, RSoft, and LucidShape Users' (customer support), and a footer with links for 'OPEN A SUPPORT ISSUE', 'GLOBAL SUPPORT CENTERS', 'SYNOPSYS TRAINING', 'SYNOPSYS USERS GROUP', and 'LAKER-VERDI FORUM'.

<https://solvnet.synopsys.com>

CS - 3

SolvNet Registration

- 1. Go to SolvNet page:**
 - <https://solvnet.synopsys.com/>
- 2. Click on:**
 - "Sign Up for an Account"
- 3. Pick a username and password.**
- 4. You will need your "Site ID"**
 - For Information on how to find your Site ID, select the "Synopsys Site ID" link
- 5. Authorization typically takes just a few minutes.**

The figure consists of three screenshots of the Synopsys New User Registration interface. The first screenshot shows the initial registration page with a 'SIGN UP FOR AN ACCOUNT' button highlighted. The second screenshot shows the user input fields for corporate email, username, password, and re-enter password. The third screenshot shows the final step where a 'Synopsys Site ID' is required before clicking 'Next'.

<https://solvnet.synopsys.com/ProcessRegistration> CS -4

Support Center

- **Industry seasoned Application Engineers:**
 - 50% of the support staff has >5 years applied experience
 - Many tool specialist AEs with >12 years industry experience
 - Engineers located worldwide
- **Great wealth of applied knowledge:**
 - Service >2000 issues per month
- **Remote access, and interactive debug, available via WebEx**

Contact us:
Open a support case

<https://www.synopsys.com/support/global-support-centers.html>

CS - 5

Other Technical Sources

- **Application Consultants (ACs):**
 - Tool and methodology pre-sales support
 - Contact your Sales Account Manager for more information
- **Synopsys Professional Services (SPS) Consultants:**
 - Available for in-depth, on-site, dedicated, custom consulting
 - Contact your Sales Account Manager for more details
- **SNUG (Synopsys Users Group):**
<https://www.synopsys.com/community/snug.html>

CS - 6

Summary: Getting Support

■ Customer Training

<https://www.synopsys.com/support/training.html>

- Register for a Class
- Download Labs

■ SolvNet

<https://solvnet.synopsys.com>

- Tool Documentation and Support Articles
- Product Update and Methodology Information / Training
- Open a Support Case (Support Center)

■ Other Technical Resources

- Synopsys Users Group (SNUG)
- Application Consultants
- Synopsys Professional Services

CS - 7

This page was intentionally left blank.