



SystemVerilog Verification UVM 1.1 Workshop

Student Guide

40-I-054-SSG-004

2012.09-SP1

Synopsys Customer Education Services
700 East Middlefield Road
Mountain View, California 94043

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

www.synopsys.com

Copyright Notice and Proprietary Information

Copyright © 2013 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.

Registered Trademarks (®)

Synopsys, AEON, AMPS, ARC, Astro, Behavior Extracting Synthesis Technology, Cadabra, CATS, Certify, CHIPit, CoMET, Confirma, CODE V, Design Compiler, DesignSphere, DesignWare, Eclypse, Formality, Galaxy Custom Designer, Global Synthesis, HAPS, HapsTrak, HDL Analyst, HSIM, HSPICE, Identify, Leda, LightTools, MAST, MaVeric, METeor, ModelTools, , NanoSim, NOVeA, OpenVera, ORA, PathMill, Physical Compiler, PrimeTime, SCOPE, SiVL, SNUG, SolvNet, Sonic Focus, STAR Memory System, SVP Café , Syndicated, Synplicity, logo Synplify, Synplify Pro, Synthesis ConstraintsOptimization Environment, TetraMAX, UMRBus, VCS, Vera, YieldExplorer .

Trademarks (™)

AFGen, Apollo, ASAP, Astro-Rail, Astro-Xtalk, Aurora, AvanWaves, Design Vision, DesignerHDL, DesignPower, DFTMAX, Direct Silicon Access, Discovery, Encore, EPIC, Galaxy,HANEX, HDL Compiler, Hercules, Hierarchical Optimization Technology, High-performance ASIC Prototyping System, HSIMplus, i-Virtual Stepper, IC Compiler, IICE, in-Sync, IN-Tandem, Intelli, Jupiter, BEST, Columbia, Columbia-CE, Cosmos, CosmosLE, CosmosScope, CRITIC, Jupiter-DP, JupiterXT, JupiterXT-ASIC, Liberty, Libra-Passport, Library Compiler, Macro-PLUS, Magellan, MAP-in SMMars, Mars-Rail, Mars-Xtalk, Milkyway, ModelSource, Module Compiler, MultiPoint, ORAengineering, Physical Analyst, Planet, Planet-PL, Platform Architect, Polaris, Power Compiler, Processor Designer, CustomExplorer, CustomSim, CustomWaveView, DC Expert, DC Professional, DC Ultra, Design Analyzer, Raphael, RippledMixer, Saturn, Scirocco, Scirocco-i, SiWare, SPW, Star-RCXT, Star-SimXT, StarRC, Symphony ModelSystem Compiler, System Designer, System Studio, TAP-in SMTaurus, TotalRecall, TSUPREM-4, VCSI, VHDL Compiler, Virtualizer, VMC, Worksheet Buffer.

Service Marks (SM)

MAP-in, SVP Café, and TAP-in are service marks of Synopsys, Inc.

SystemC is a trademark of the Open SystemC Initiative and is used under license. ARM and AMBA are registered trademarks of ARM Limited. Saber is a registered trademark of SabreMark Limited Partnership and is used under license. All other product or company names may be trademarks of their respective owners.

Document Order Number: 40-I-054-SSG-004
SystemVerilog Verification UVM 1.1 Student Guide

Table of Contents

Unit i:

Introductions	i-2
Facilities	i-3
Course Materials	i-4
Workshop Goal	i-5
Target Audience.....	i-6
Workshop Prerequisites	i-7
Day 1 Agenda	i-8
Day 2 Agenda	i-9
Day 3 Agenda	i-10
What Is the Device Under Test?	i-11
A Functional Perspective	i-12
Input Packet Structure.....	i-13
Output Packet Structure	i-14
Reset Signal	i-15
Icons Used in this Workshop	i-16

Unit 1: OOP Inheritance Review

Unit Objectives	1-2
Object Oriented Programming (OOP): Class	1-3
Object Oriented Programming: Inheritance.....	1-4
Object Oriented Programming: 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 Classes	1-17
Singleton Objects	1-18
Simple Singleton Proxy Object Example	1-19
Applying Proxy Class in Factory (1/5)	1-20
Applying Proxy Class in Factory (2/5)	1-21
Applying Proxy Class in Factory (3/5)	1-22
Applying Proxy Class in Factory (4/5)	1-23
Applying Proxy Class in Factory (5/5)	1-24

Table of Contents

Unit 2: UVM Overview

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
The Testbench Environment/Architecture.....	2-9
UVM Encourages Encapsulation for Reuse	2-10
UVM Structure is Scalable	2-11
Structural Support in UVM.....	2-12
Component Phasing	2-13
Hello World Example	2-14
Compile and Simulate.....	2-15
Inner Workings of UVM Simulation	2-16
User Control of Reporting and Filtering.....	2-17
Key Component Concepts: Parent-Child.....	2-18
Key Component Concepts: Logical Hierarchy	2-19
Key Component Concepts: Phase	2-20
Key Component Concepts: Search & Replace	2-21
Key UVM Elements (1/2).....	2-22
Key UVM Elements (2/2).....	2-23
Simple Example of Key UVM Elements.....	2-24
Key UVM Testbench Elements: TLM.....	2-25
Key UVM Element: Transaction Class.....	2-26
Key UVM Element: Sequence Class	2-27
Key UVM Element: Sequencer Class.....	2-28
Key UVM Element: Driver Class	2-29
Key UVM Element: Agent Class.....	2-30
Key UVM Element: Environment Class.....	2-31
Key UVM Element: Test Class.....	2-32
Test Program.....	2-33
Key UVM Testbench Elements: UVM Report.....	2-34
Embed Report Messages.....	2-35
Embed Report Messages.....	2-36
Controlling Report Message Verbosity	2-37
Default Simulation Handling	2-38
Modify Report Message Action in Test.....	2-39
Command Line Control of Report Messages.....	2-40
User Filter-able Code Block	2-41
Misc: Command Line Processor.....	2-42
Misc: Objection Run-Time Argument.....	2-43
Unit Objectives Review	2-44
Lab 1 Introduction.....	2-45

Table of Contents

Appendix	2-46
UVM Class Tree	2-47
UVM Class Tree	2-48
VCS Support for UVM	2-49
Synopsys UVM Support	2-50
Compiling UVM with VCS	2-51
DVE UVM Macro Debugging	2-52
DVE UVM Transaction Level Debugging	2-53
VCS enabled UVM STACKTRACE.....	2-54
VMM to UVM Mapping.....	2-55
Store UVM Report Messages in File.....	2-56
Store Report Message in File.....	2-57
Calling UVM Messaging From Modules	2-58
Calling UVM Messaging From Modules (1/2).....	2-59
Calling UVM Messaging From Modules (2/2).....	2-60
Demote/Promote UVM Messages	2-61
Demote/Promote UVM Report Messages	2-62
<uvmgen.....< u=""></uvmgen.....<>	2-63
Usage – Command line options	2-64
Environment Generation (Verbose Mode).....	2-65
Directory Structure	2-66
Individual Template Generation	2-67
uvmgen Quick Mode.....	2-68
Quick Environment Generation	2-69
Important uvmgen Options	2-70

Unit 3: Modeling Transactions

Unit Objectives	3-2
Transaction Flow	3-3
Modeling Transactions.....	3-4
Other Properties to be Considered (1/2)	3-5
Other Properties to be Considered (2/2)	3-6
Transactions: Must-Obey Constraints.....	3-7
Transactions: Should-Obey Constraints	3-8
Transactions: Constraint Considerations	3-9
Transaction Class Methods	3-10
`uvm_field_* Field Automation Macros	3-11
Methods Specified by FLAG	3-12
Print Radix Specified by FLAG.....	3-13
Transaction Methods.....	3-14
Method Descriptions	3-15
Customization of Methods	3-16
Using Transaction Methods	3-17
Modify Constraint in Transactions by Type	3-18

Table of Contents

Transaction Replacement Results	3-19
Modify Constraint in Transaction by Instance.....	3-20
Parameterized Transaction Class	3-21
Unit Objectives Review	3-22
Lab 2 Introduction.....	3-23
Appendix.....	3-24
uvm_sequence_item Class Tree.....	3-25
uvm_sequence_item Class Tree.....	3-26
uvm_object_utils Macro	3-27
uvm_object_utils Macro	3-28

Unit 4: Creating Stimulus Sequences

Unit Objectives	4-2
uvm_sequence Class Common Members	4-3
Implicit Sequence Execution Protocol.....	4-4
Mapping Protocol to Code (1/2)	4-5
Mapping Protocol to Code (2/2)	4-6
Sequence Class Requirements	4-7
Sequence Class Callbacks.....	4-8
User Sequence with Callback	4-9
User Can Manually Create and Send Item.....	4-10
`uvm_do Macro Interaction Detailed.....	4-11
Sequence with randc Transaction Property.....	4-12
User Can Implement Scenario Sequence.....	4-13
Nested Sequences	4-14
Executing User Sequences in Test	4-15
Implicit Sequence Execution at a Phase	4-16
Explicit Sequence Execution	4-17
Configuring Sequences (Instance-Based).....	4-18
Instance-Based Configuration Benefits	4-19
Configuring Sequences (Sequencer-Based).....	4-20
Sequencer-Based Configuration Benefits	4-21
Configuring Sequences (Agent-Based).....	4-22
Unit Objectives Review	4-23
Lab 3 Introduction.....	4-24
Appendix.....	4-25
Sequence Priority/Weight	4-26
Sequence Priority/Weight	4-27
Sequencer-Driver Response Port	4-28
Sequencer-Driver Response Port	4-29
Out-Of-Order Sequencer-Driver Port	4-30
Sequence	4-31
Out-Of-Order Driver.....	4-32

Table of Contents

Unit 5: Component Configuration & Factory

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
Other Examples: Physical Interface.....	5-9
Configuring Component's DUT Interface.....	5-10
Dynamic Control Variable Example.....	5-11
Global UVM Resource	5-12
Dynamic Control Variable Example (Global).....	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
Customize Transaction in Test.....	5-20
Component Factory.....	5-21
UVM Factory Component Creation.....	5-22
Customize Component in Test.....	5-23
Command-line Override	5-24
Visually Inspect Structural Correctness.....	5-25
Can Choose Print Format.....	5-26
Visually Inspect Factory Overrides	5-27
Parameterized Component Class	5-28
Unit Objectives Review	5-29
Appendix.....	5-30
uvm_component_utils Macro	5-31
uvm_component_utils Macros.....	5-32
Variable Length Configuration	5-33
Configuring Variable Length Members.....	5-34
Configuring Variable Length Members.....	5-35
UVM Configuration Data Base API	5-36
uvm_config_db::set() – (1/2)	5-37
uvm_config_db::set() – (2/2)	5-38
uvm_config_db::get() – (1/2).....	5-39
uvm_config_db::get() – (2/2).....	5-40
UVM Configuration Debugging.....	5-41
Debugging configuration issues (1/4).....	5-42
Debugging configuration issues (2/4).....	5-43
Debugging configuration issues (3/4).....	5-44
Debugging configuration issues (4/4).....	5-45

Table of Contents

UVM Resource and Config DB debug 5-46

Unit 6: Component Communication

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 4 Introduction.....	6-20
Appendix.....	6-21
TLM 2.0 Generic Payload.....	6-22
TLM 2.0 Generic Payload (1/4).....	6-23
TLM 2.0 Generic Payload (2/4).....	6-24
TLM 2.0 Generic Payload (3/4).....	6-25
TLM 2.0 Generic Payload (4/4).....	6-26
DVE Debugging Features	6-27
UVM Transaction Debug (VCS Installation)	6-28
DVE Methodology-Aware Panes	6-29
Managing Debugging Session with DVE	6-30
Local and Stack Pane Usage.....	6-31
Filtering Data	6-32
Watch Object in DVE	6-33
Class Browser in DVE	6-34
Filtering Classes.....	6-35
Hiding Methodology classes.....	6-36
Other Testbench Tricks.....	6-37
Thread Debug.....	6-38
DVE Instance Object IDs vs. UVM Instance ID	6-39
Class Instance Object IDs	6-40
Class Instance Object IDs	6-41
Breakpoints – Object IDs.....	6-42

Table of Contents

Breakpoints - Conditions	6-43
Per Instance Breakpoints –i.e m_name.....	6-44
Macro Breakpoints.....	6-45
Breakpoints - Methods.....	6-46
Step into a Randomize Call.....	6-47
Static View: Constraints in Class Browser	6-48
Dynamic View: Local Pane	6-49
Solver/Relation Inside Constraint Dialog	6-50
Interactive Constraint Debug	6-51
DVE UVM-Aware Debugging Features.....	6-52
UVM-Aware Features in DVE	6-53
Object Hierarchy Browser	6-54
Testbench Class Browser.....	6-55
Testbench Object Variables	6-56
UVM Factory Debug	6-57
UVM Resource Debug.....	6-58
UVM Phase/Objection Debug	6-59
UVM Sequence Debug	6-60
UVM Transaction Debug.....	6-61
Smart Log.....	6-62

Unit 7: Scoreboard & Coverage

Unit Objectives	7-2
Scoreboard - Introduction	7-3
Scoreboard – Data Streams.....	7-4
Scoreboard Implementation	7-5
Scoreboarding: Monitor	7-6
Embed Monitor in Agent	7-7
UVM Agent Example	7-8
Using UVM Agent in Environment.....	7-9
Parameterized Scoreboard	7-10
Scoreboard: Transformed Transaction.....	7-11
Scoreboard: Out-Of-Order	7-12
Scoreboard: Out-Of-Order	7-13
Functional Coverage	7-14
Connecting Coverage to Testbench	7-15
Configuration Coverage	7-16
Configuration Coverage	7-17
Stimulus Coverage	7-18
Correctness Coverage	7-19
Unit Objectives Review	7-20
Appendix.....	7-21
Multi-Stream Scoreboard.....	7-22
Scoreboard: Multi-Stream.....	7-23

Table of Contents

Scoreboard: Multi-Stream.....	7-24
-------------------------------	------

Unit 8: UVM Callback

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
User Callback Debug	8-13
Sequence Simple Callback Methods.....	8-14
Unit Objectives Review	8-15
Lab 5 Introduction.....	8-16

Unit 9: Virtual Sequence/Sequencer

Unit Objectives	9-2
Virtual Sequences	9-3
Virtual Sequence/Sequencer	9-4
Virtual Sequencer.....	9-5
Virtual Sequence.....	9-6
Connect Sequencer to Virtual Sequencer	9-7
Connect Sequencer to Virtual Sequencer	9-8
Sequence Execution Management	9-9
Synchronization Mechanism: uvm_event.....	9-10
Component Synchronization: uvm_barrier.....	9-11
Specialized Pools for Synchronization	9-12
uvm_event_pool Example (Trigger).....	9-13
uvm_event_pool Example (Wait for Trigger)	9-14
Protecting Stimulus (Grab/Ungrab)	9-15
Unit Objectives Review	9-16
Appendix.....	9-17
Resource Pool	9-18
Resource Pool: uvm_pool	9-19
Sequence Library	9-20
Sequence Library	9-21
Building a Sequence Library Package (1/2)	9-22
Building a Sequence Library Package (2/2)	9-23

Table of Contents

Reference Sequence Library in Environment.....	9-24
Register and Execute Sequences.....	9-25
Customize Sequence Library Object	9-26
Configuration Issues	9-27
Configure Sequence Library	9-28
Configure Sequence Library Example.....	9-29
User Define Sequence Execution.....	9-30

Unit 10: Component Phasing Revisited

Unit Objectives	10-2
Phasing in UVM 1.0+	10-3
Common Phases.....	10-4
Run-Time Task Phases	10-5
Task Phase Synchronization	10-6
Phase Objection (1/5).....	10-7
Phase Objection (2/5).....	10-8
Phase Objection (3/5).....	10-9
Phase Objection (4/5).....	10-10
Phase Objection (5/5).....	10-11
Phase Timeout.....	10-12
Advanced Features.....	10-13
Phase Domains (1/2).....	10-14
Phase Domains (2/2).....	10-15
User Defined Phase.....	10-16
Phase Jump: Backward	10-17
Phase Jump: Forward	10-18
Phase Callbacks (1/2).....	10-19
Phase Callbacks (2/2).....	10-20
Get Phase Execution Count	10-21
Unit Objectives Review	10-22
Lab 6 Introduction.....	10-23
Appendix.....	10-24
uvm_phase Class Key Methods	10-25
uvm_phase Class Key Methods	10-26
Component Phasing Guideline	10-27
Driver Guideline	10-28
Monitor Guideline.....	10-29
Agent Guideline	10-30
Scoreboard Guideline	10-31
Environment Phase Guideline.....	10-32
Test Phase Guideline.....	10-33
Jump Code Example	10-34
Driver Code for Jump	10-35
Driver Code for Jump	10-36

Table of Contents

Monitor Code for Jump.....	10-37
Scoreboard Jump Code	10-38

Unit 11: UVM Register Abstraction Layer (RAL)

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: Register File.....	11-20
UVM Register Abstraction File (.ralf) Syntax.....	11-21
UVM Register Abstraction: Memory	11-22
UVM Register Abstraction File (.ralf) Syntax.....	11-23
UVM Register Abstraction: Block.....	11-24
UVM Register Abstraction: Block.....	11-25
UVM Register Abstraction File (.ralf) Syntax.....	11-26
UVM Register Abstraction: System	11-27
Step 3: Create UVM Register Abstraction Model	11-28
Step 4: Create UVM Register Adapter	11-29
Sequencer Adapter Class (One per Interface).....	11-30
UVM Register Abstraction Sequence.....	11-31
Step 5: Instantiating UVM Register Model	11-32
Tie Sequencer Adapter to Register Map.....	11-33
Run RAL Sequence Implicitly or Explicitly.....	11-34
UVM Register Test Sequences	11-35
Implicit Mirror Predictor.....	11-36
Run RAL Test Sequences with Auto Predict.....	11-37
Explicit Mirror Predictor.....	11-38
Implementing Explicit Mirror Predictor	11-39
RAL Test Sequence with Explicit Predict	11-40
Unit Objectives Review	11-41

Table of Contents

Appendix.....	11-42
UVM Register Modes	11-43
UVM Register Modes	11-44
Register Frontdoor Write	11-45
Register Frontdoor Read	11-46
Register Backdoor Write.....	11-47
Register Backdoor Read	11-48
Register Backdoor Poke.....	11-49
Register Backdoor Peek	11-50
Mirrored & Desired Property Update	11-51
UVM Register Desired Property Write.....	11-52
Randomize UVM Register Desired Property	11-53
UVM Register Desired Property Read	11-54
Mirrored & DUT Value Update.....	11-55
Writing to uvm_reg Mirrored Property.....	11-56
Reading uvm_reg Mirrored Property.....	11-57
Typical use of uvm_reg predict() method (1/2).....	11-58
Typical use of uvm_reg predict() method (2/2).....	11-59
UVM Memory Modes.....	11-60
Memory Frontdoor Write.....	11-61
Memory Frontdoor Read.....	11-62
Memory Backdoor Write	11-63
Memory Backdoor Read	11-64
Memory Backdoor Poke	11-65
Memory Backdoor Peek	11-66
ralgen Options	11-67
ralgen Options	11-68
ralgen Address Granularity	11-69
ralgen XMR	11-70
ralgen Functional Coverage	11-71
UVM Register Class Tree	11-72
UVM Register Base/Library Class Hierarchy	11-73
UVM Register/Memory Class Members	11-74
UVM Register Class Key Properties	11-75
UVM Memory Class Key Properties	11-76
uvm_reg_bus_op Definition	11-77
UVM Register Callbacks	11-78
RAL Class Key Callback Members	11-79
RAL Class Key Callback Members	11-80
Changing Address Offsets of a Domain	11-81
Changing the Address offsets of a Domain	11-82

Table of Contents

Unit 12: Summary

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 7 Introduction.....	12-13
That's all Folks!	12-14

Unit CS: Customer Support

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

SYNOPSYS®

UVM-1.1

SystemVerilog Testbench

VCS 2012.09-SP1

Synopsys Customer Education Services
© 2013 Synopsys, Inc. All Rights Reserved

Synopsys 40-I-054-SSG-004

Introductions

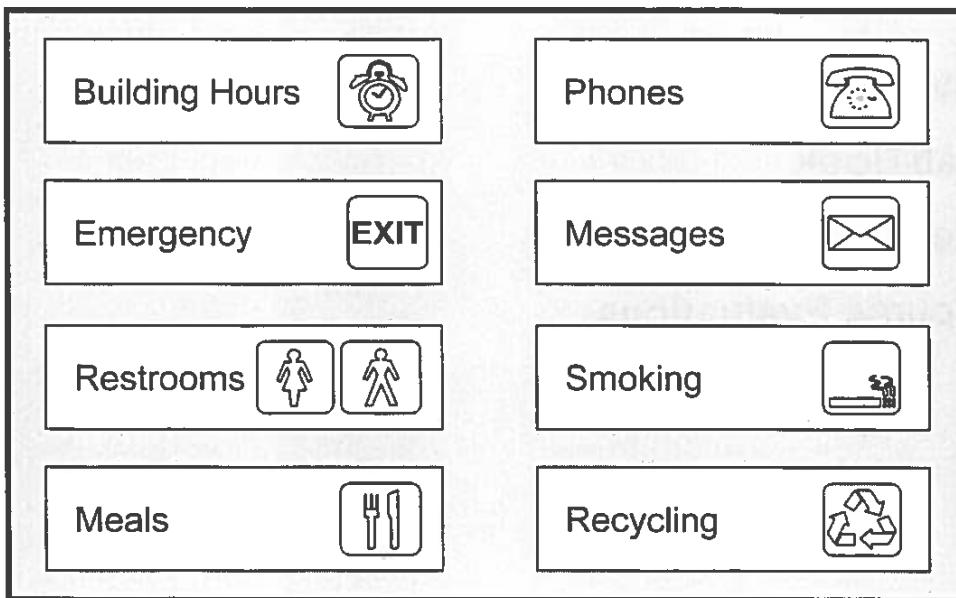
- Name
- Company
- Job Responsibilities
- EDA Experience
- Main Goal(s) and Expectations for this Course

i- 2



EDA = Electronic Design Automation

Facilities



Please turn off cell phones and pagers

i- 3

Course Materials

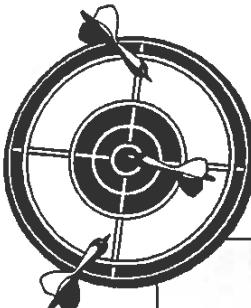
- Student Workbook
- Lab Book
- Reference Materials
- Course Evaluations

i- 4



EDA = Electronic Design Automation

Workshop Goal



Acquire the skills to implement a UVM
testbench to verify Verilog/VHDL designs with
coverage-driven random stimulus

i- 5

Target Audience

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



i- 6

Workshop Prerequisites

- You should have experience in the following areas:
 - Familiarity with a UNIX text editor
 - Programming skills in SystemVerilog
 - Debugging experience with SystemVerilog

i- 7

Day 1 Agenda

**DAY
1**

1 OOP Inheritance Review

2 UVM Overview



3 Modeling Transactions



4 Creating Stimulus Sequences



i- 8

Day 2 Agenda

**DAY
2**

5 Component Configuration & Factory



6 Component Communication



7 Scoreboard & Coverage



8 UVM Callback



i- 9

Day 3 Agenda

**DAY
3**

9 Virtual Sequence/Sequencer



10 Component Phasing Revisited

11 UVM Register Abstraction Layer (RAL)



12 Summary

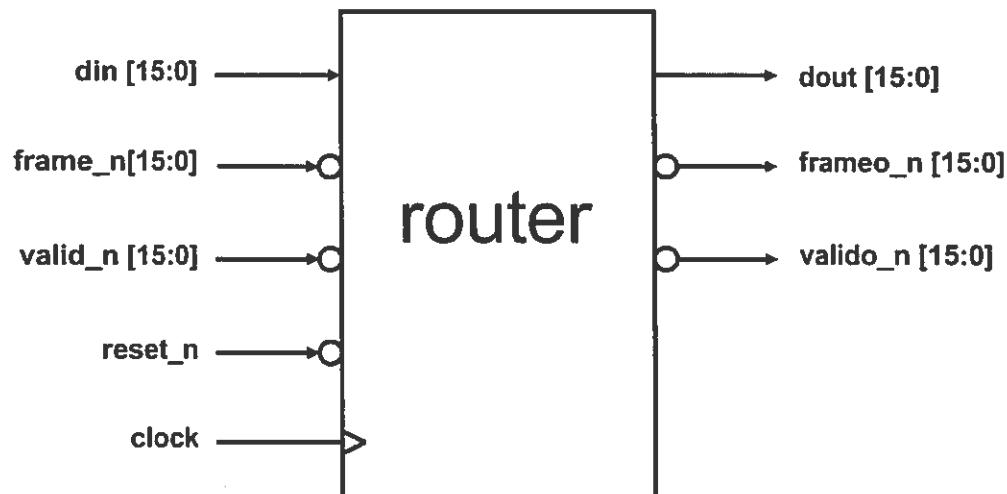
CS Customer Support

i-10

What Is the Device Under Test?

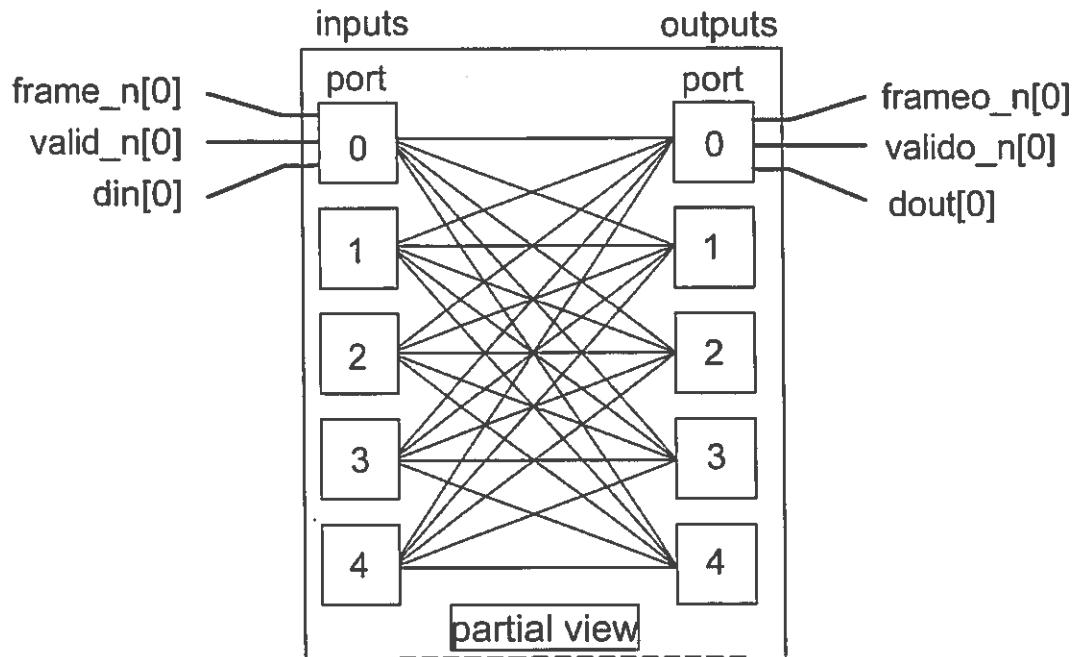
A router:

16 x 16 crosspoint switch



i-11

A Functional Perspective



i-12

Input Packet Structure

- **frame_n:**

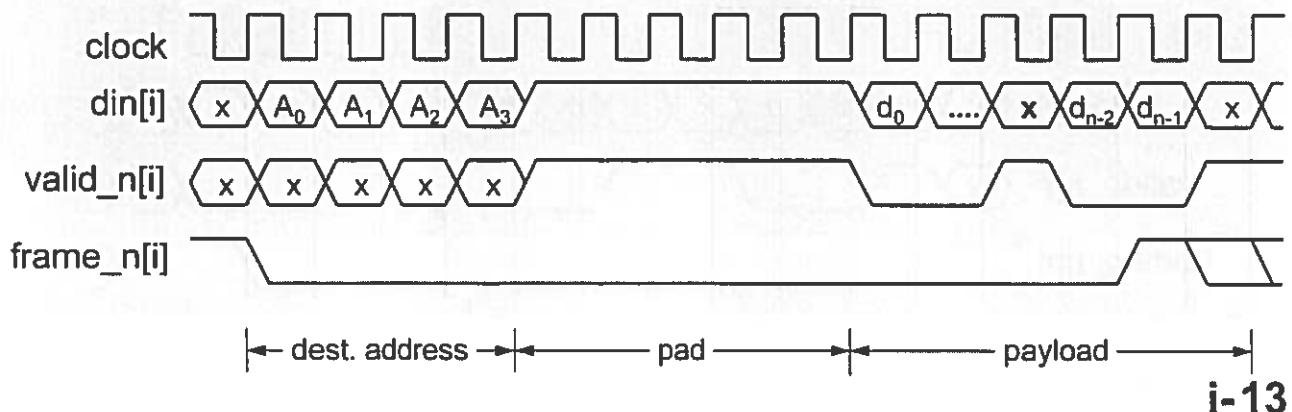
- Falling edge indicates first bit of packet
- Rising edge indicates last bit of packet

- **din:**

- Header (destination address & padding bits) and payload

- **valid_n:**

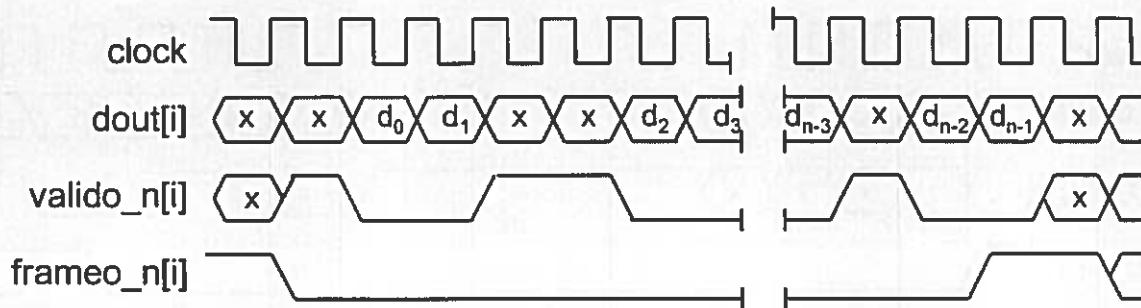
- valid_n is low if payload bit is valid, high otherwise



i-13

Output Packet Structure

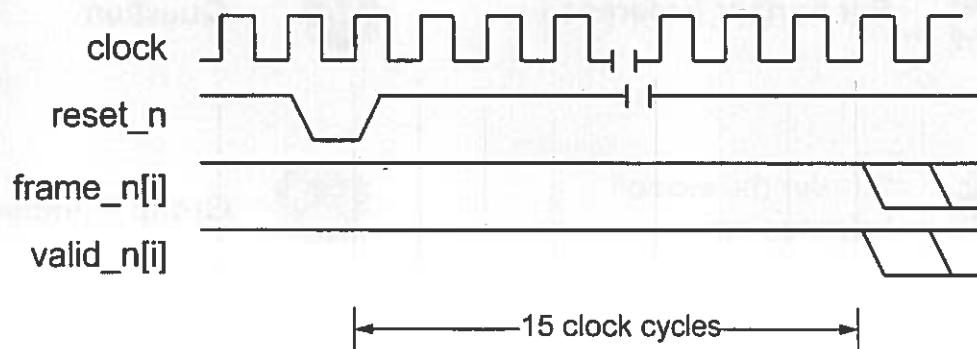
- Output activity is indicated by:
`frameo_n`, `valido_n`, and `dout`
- Data is valid only when:
 - `frameo_n` output is low (except for last bit)
 - `valido_n` output is low
- Header field is stripped



i-14

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-15

Icons Used in this Workshop



Lab Exercise



Caution



Recommendation



Definition of Acronyms



For Further Reference



Question



“Under the Hood” Information



Group Exercise

i-16

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.

Under the Hood Information: Information about the internal behavior of the tool.

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

Definition of Acronyms: Defines the acronym used in the slides.

Question: Marks questions asked on the slide.

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

Agenda: Day 1

**DAY
1**

1 OOP Inheritance Review

2 UVM Overview



3 Modeling Transactions



4 Creating Stimulus Sequences



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

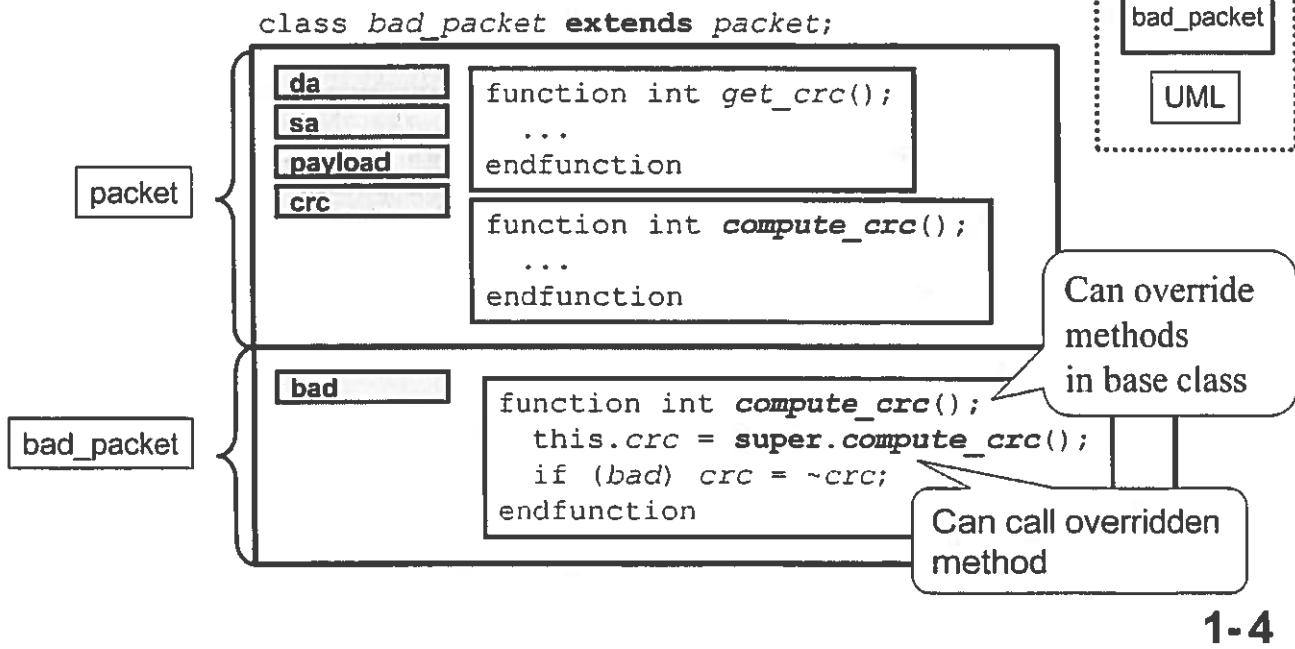
    function int compute_crc(); // packet class method
        ...
    endfunction
endclass: packet
```

1-3

Object Oriented Programming: Inheritance

■ OOP inheritance

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



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()`.

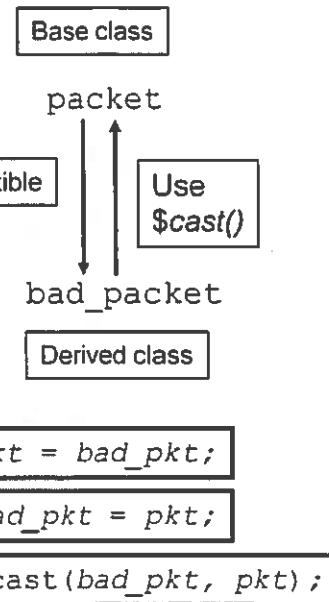
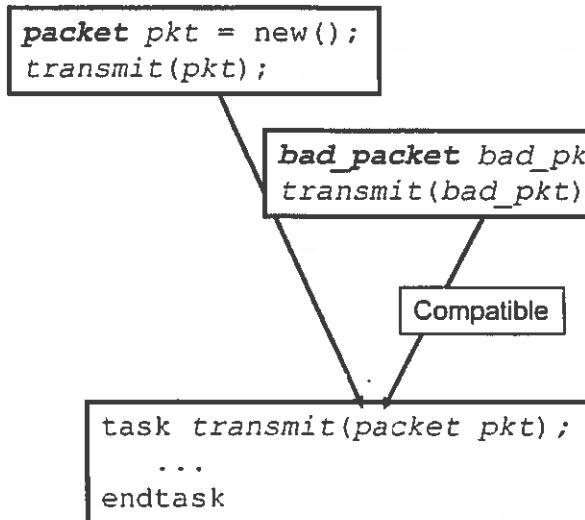
A class can't go up more than one level – `super.super.name` is not legal.

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.

Object Oriented Programming: Inheritance

■ Derived classes compatible with base class

- Can reuse code



1-5

Now look at how you would use these objects.

On the left is a diagram showing how a task can pass a `packet` object to the `transmit` method.

This method will read and write the class members such as `sa`, and `compute_crc()` with `pkt.sa`, and `pkt.compute_crc()`.

Or, a task could create a `bad_packet`. It can pass this to `transmit()` as every `bad_packet` is also a `packet`, with `sa`, and `compute_crc()`.

On the lower right, this idea is shown by the assignment `pkt = bad_pkt`; After this assignment, the handle `pkt` can still refer 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` pointed to a `bad_packet` object, then it is 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)` first 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.

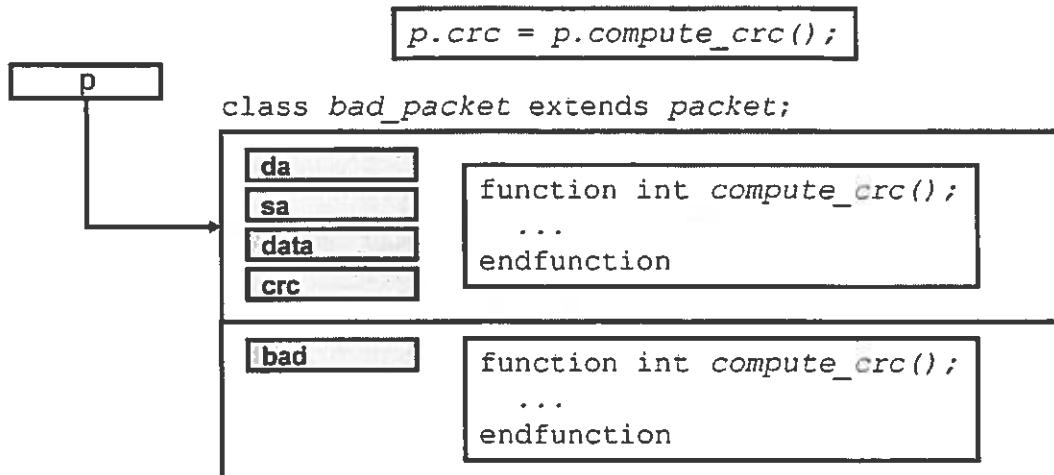
When you call `$cast` as a task, it will cause termination of simulation if the source object is not type compatible with the destination handle. However, if you call `$cast` as a function, it silently returns a 1 for success, and 0 for failure.

`if (!$cast(bad_pkt, pkt))`

`$display("pkt type is not compatible with bad_pkt handle");`

OOP: Polymorphism

■ Which method gets called?



■ Depends on

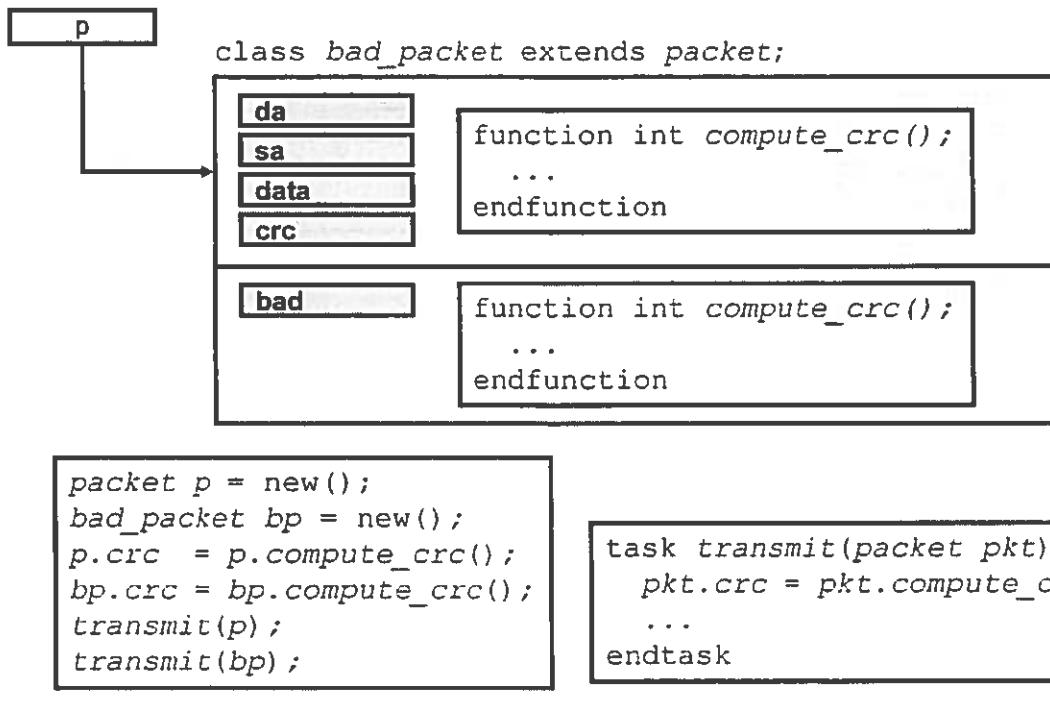
- Type of handle *p* (e.g. “*packet*” or “*bad_packet*”?)
- Whether *compute_crc()* is *virtual* or not

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,

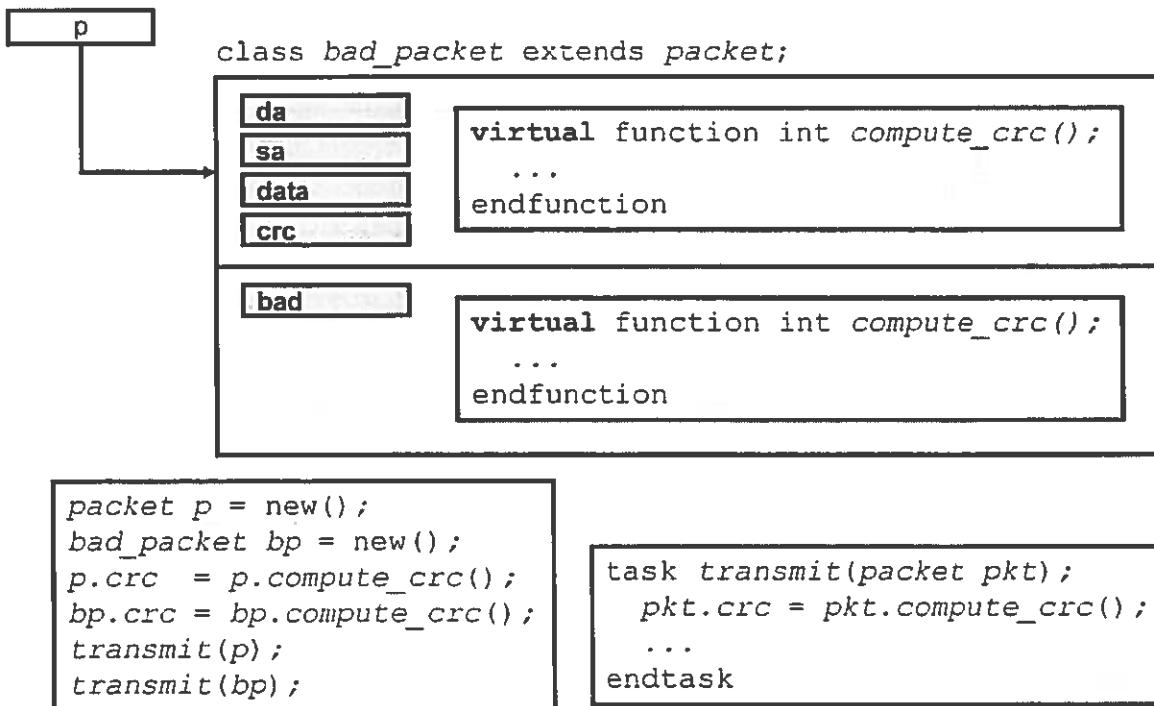
`transmit(bp)` also executes the `compute_crc()` method in `packet` method despite the fact that `bp` was the object passed in.

The reason is that the object executing `compute_crc()`, `pkt`, is a `packet` object. 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 polymorphism is disabled in methods.

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.

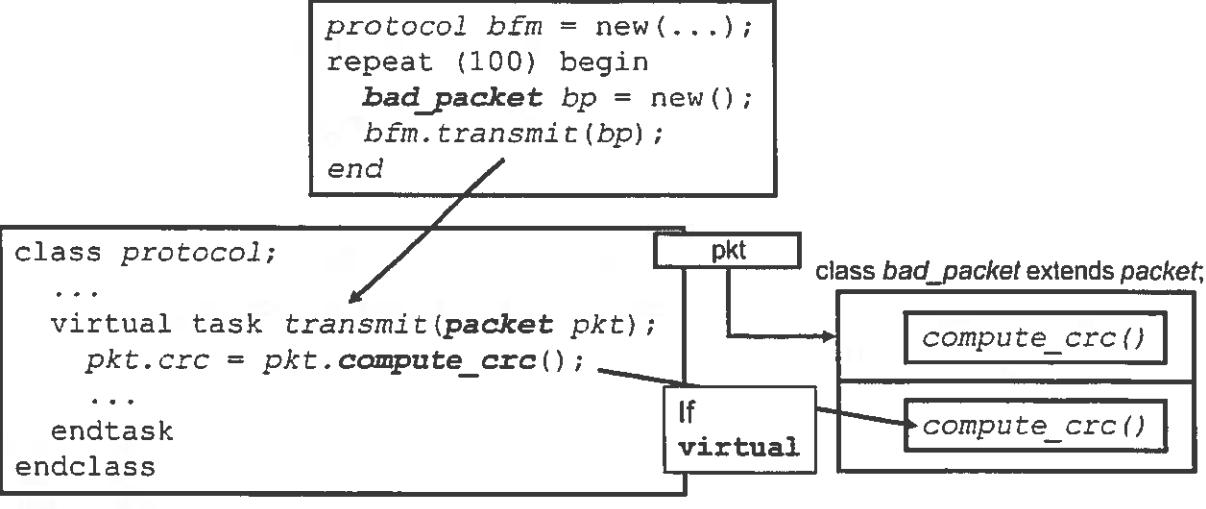
Example, the following will not compile:

```
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



Guideline: methods should be virtual

Can inject CRC errors
without modifying original code

1-9

User class should implement 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, the general guide is to make public methods non-virtual. This is to ensure that the user who extends from the base class does not corrupt the intent of the public method.

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**

Parameterized Classes

■ Written for a generic type

- Type parameter passed at instantiation, just like parameterized modules
- Allows reuse of common code

```
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
```

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

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: Generic_Generator
```

One benefit of parameterized classes is that checking is done at compile time for type safety.

Parameterized classes can be extended just like other classes.

typedef

- Often need to use a class before declaration

- e.g. two classes need handle to each other

```
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

- Or, to 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 define a handle of type `S2`. If you reverse the class definitions, then class `S2` can't declare a handle of `S1`. Use `typedef` to tell the compiler that a class will be declared later on.

Another use for `typedef` is creating new types. Rather than typing the fully parameterized class name, you can use `typedef` to create shortcuts.

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 property is not required.

If you give any default values, they must be specified on both the extern declaration and the body below.

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 definition, not the object.
 - Can store meta-data, such as number of instances constructed
 - Shared by all objects of that class

```
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 must be re-allocated 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

- Call a method without a handle
- Can only access static properties

```
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

At the end of the test, you may want to see how many packets were created. Normally this would require constructing a packet object, and then accessing the count property.

But a static method can be called with the class name::method(), no handle needed.

Singleton Classes

- Used to define a “global” activity such as printing or factory registry
- No instance of the singleton class exists/allowed
- Contains only static members

```
class print;
    static int err_count = 0, max_errors = 10;
    static function void error (string msg);
        $display("@%t: ERROR %s", $realtime, msg);
        if (err_count++ > max_errors)
            $finish;
    endfunction
    local function new(); → No object allowed to exist
    endfunction
endclass

if (expect != actual)
    print::error("Actual did not match expected");
```

1-17

Some classes should not require objects to be constructed. For example, printing is a global activity. It would be a large burden if you had to construct this object, and then pass it into every class that ever needed to print a message.

Instead, treat the print class as a singleton class. To call its method, use the class name and the scope resolution operator, as in:

```
print::error("Message");
```

Singleton Objects

- A singleton object is a globally accessible static object which provides customized 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

```
class simple_class;  
    static simple_class me = get();  
    static function simple_class get();  
        if (me == null) me = new(); return me;  
    endfunction  
    local function new();  
    endfunction  
    extern function void print();  
endclass
```

```
simple_factory test_factory = simple_factory::get();  
test_factory.print();
```

Object created at compile-time

Globally accessible at run-time

No object is allowed to be constructed at run-time

1-18

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 `get()` method.

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

    static function this_type get();
        if (me == null) me = new(); return me;
    endfunction

    static function T create();
        create = new();
    endfunction
endclass
```

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

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

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

1-19

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 creates an object. It works similar to calling `new()`, but does some special actions.

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 interface methods for proxy services
 - ◆ Construct objects that the proxy represent
 - ◆ Get type name

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

1-20

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
```

1-21

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]; Original proxy type  
  
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
```

To be replaced by overridden proxy

1-22

Applying Proxy Class in Factory (4/5)

■ Proxy class

- Implements the abstracted interface methods
- Registers itself in factory
- Create object using factory

```
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();
        factory f = factory::get(); if (me == null) begin me = new();
        f.register(me); end return me;
    endfunction
    static function T create(string name);
        factory f = factory::get();
        $cast(create, f.create_object_by_type(me, name));
    endfunction
    virtual function base create_object(string name);
        T object_represented = new(name);
        return object_represented;
    endfunction
endclass
```

Use factory to create object

1-23

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();
    // other code not shown
end
```

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

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

1-24

Agenda: Day 1

DAY

1

1 OOP Inheritance Review

2 UVM Overview



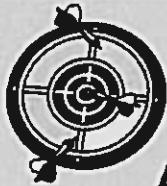
3 Modeling Transactions



4 Creating Stimulus Sequences



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**

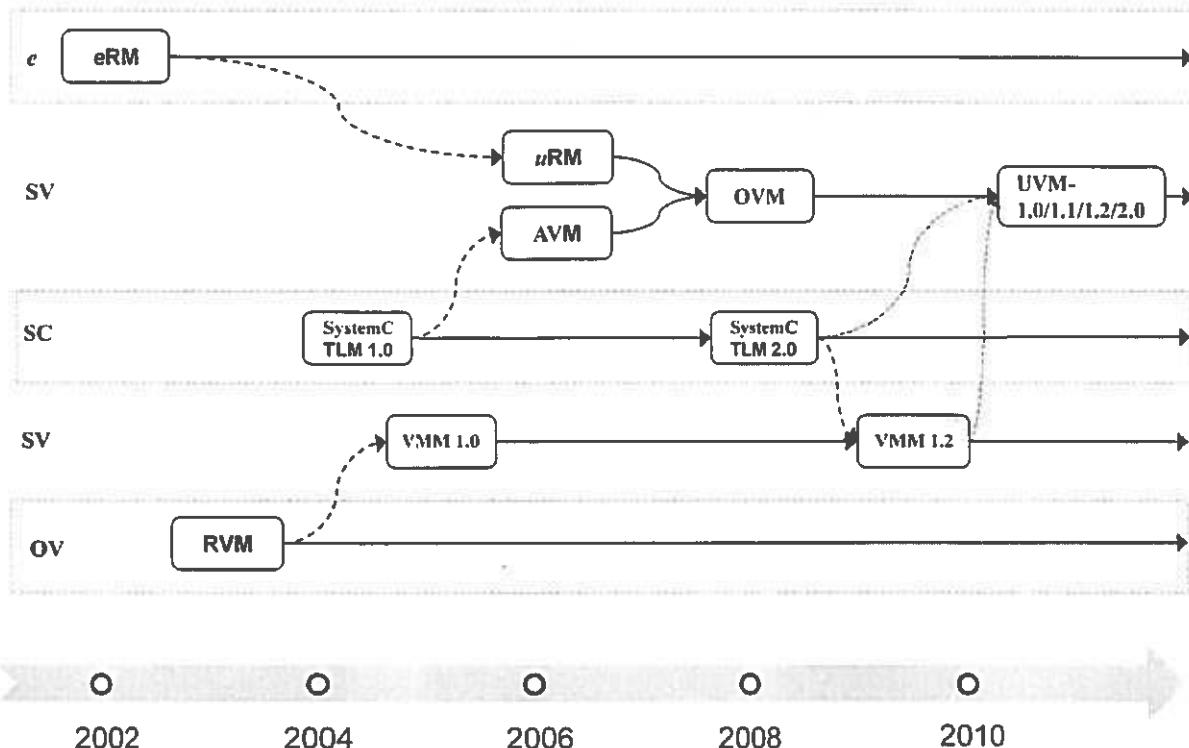
2-2

UVM - Universal Verification Methodology

- An effort (by an Accellera committee) to define a standard verification methodology & base class library
 - Uses classes and concepts from VMM, OVM
 - Still work in progress
 - Published after all vendors' approval
- Related Websites:
 - UVM Public Website - <http://www.accellera.org/community/uvm>
 - Member Website - <http://www.accellera.org/apps/org/workgroup/vip>
 - Mantis (Bug Tracking) - http://eda.org/svdb/view_all_bug_page.php
 - Sourceforge - <http://uvm.git.sourceforge.net>

2-3

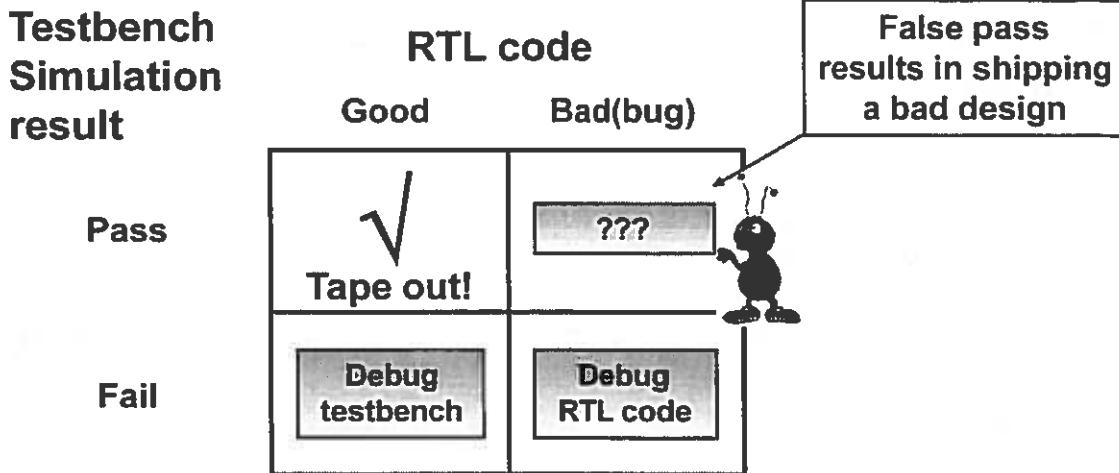
Origin of UVM



2-4

Verification Goal

- Ensure full conformance with specification:
 - Must avoid false passes



How do we achieve this goal?

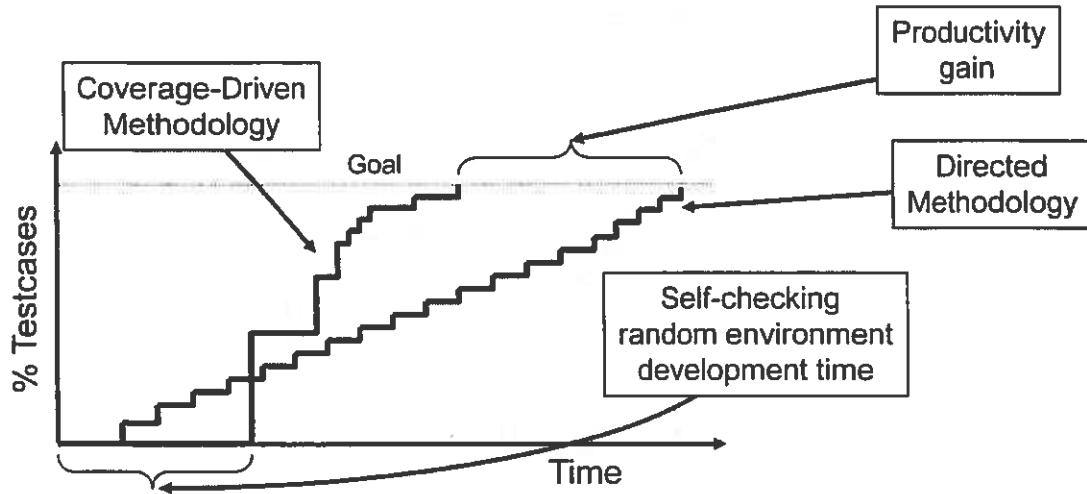
2-5

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

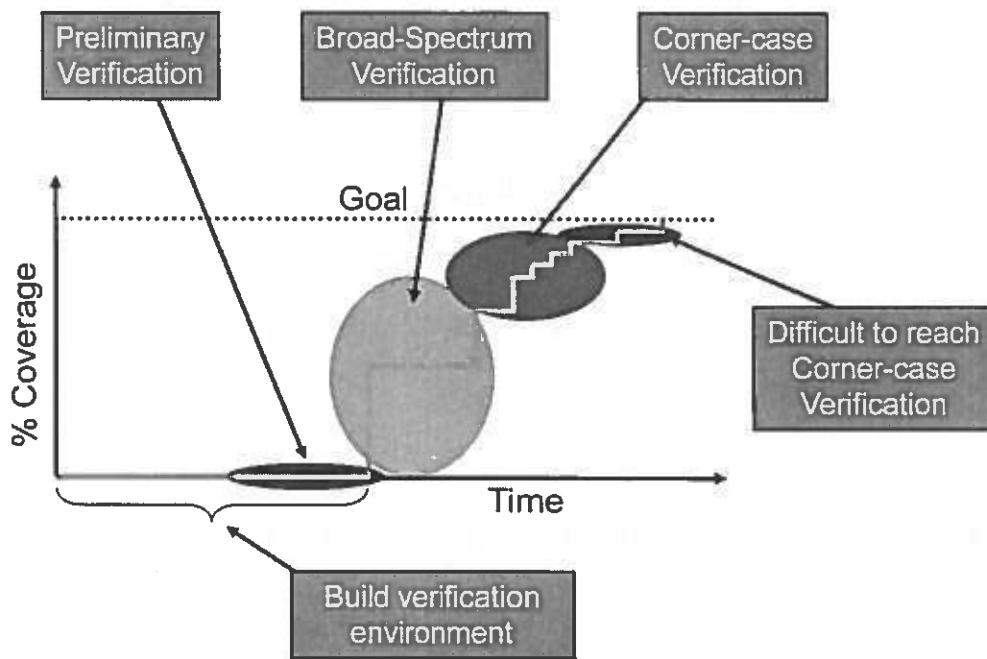
- Focus on uncovered areas
- Trade-off authoring time for run-time
- Progress measured using functional coverage metrics



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.

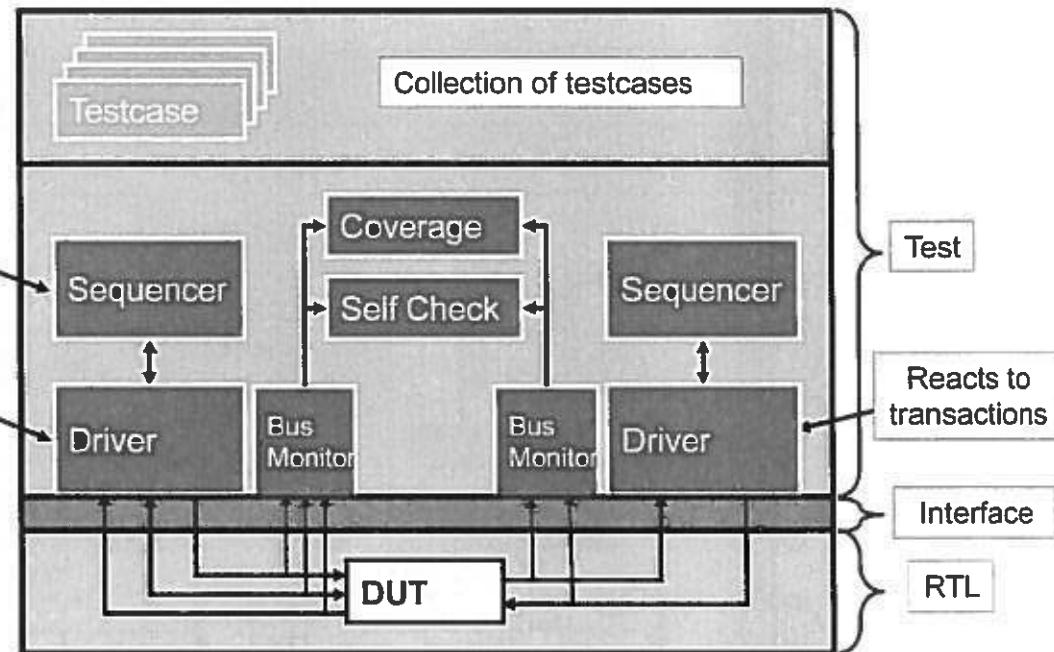
Run More Tests, Write Less Code

- **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, 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

The Testbench Environment/Architecture

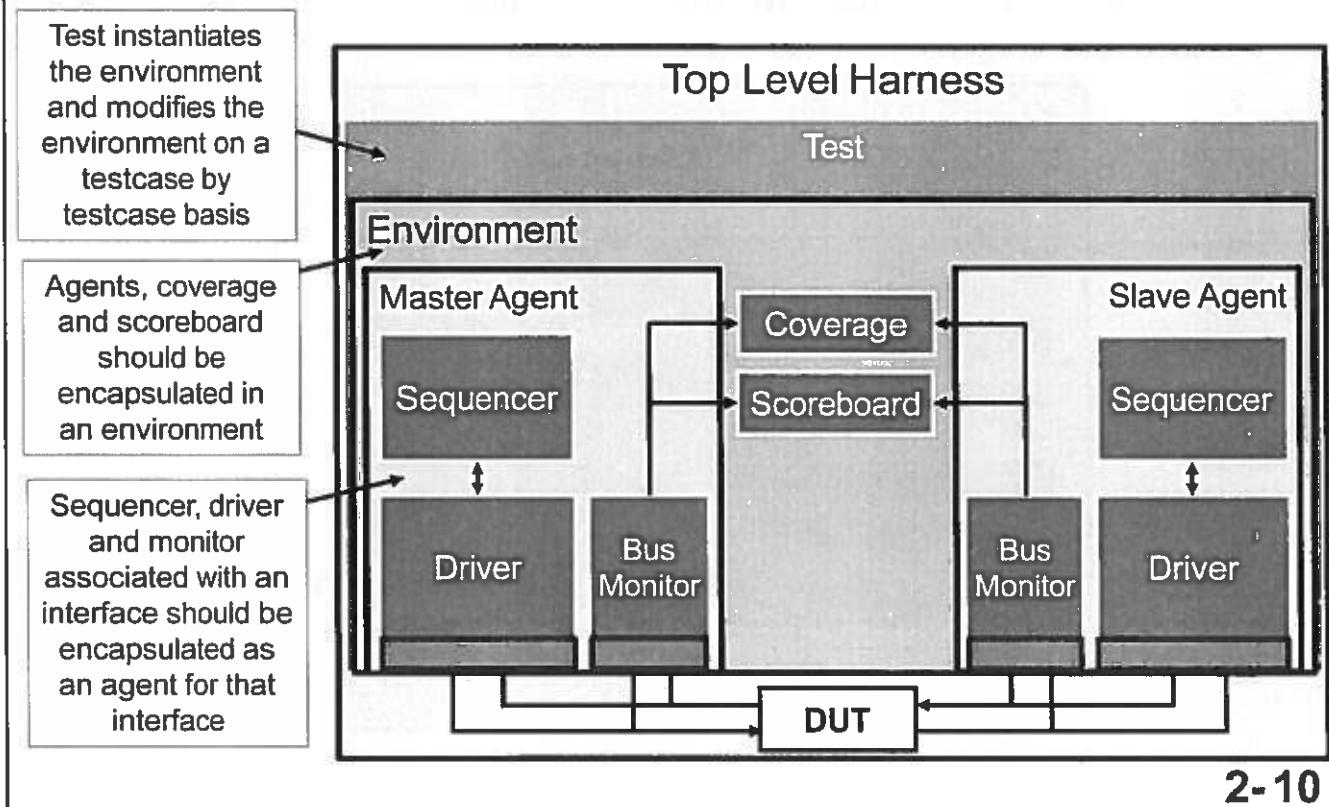
■ SystemVerilog testbench structure



2-9

UVM Encourages Encapsulation for Reuse

■ Structure should be architected for reuse



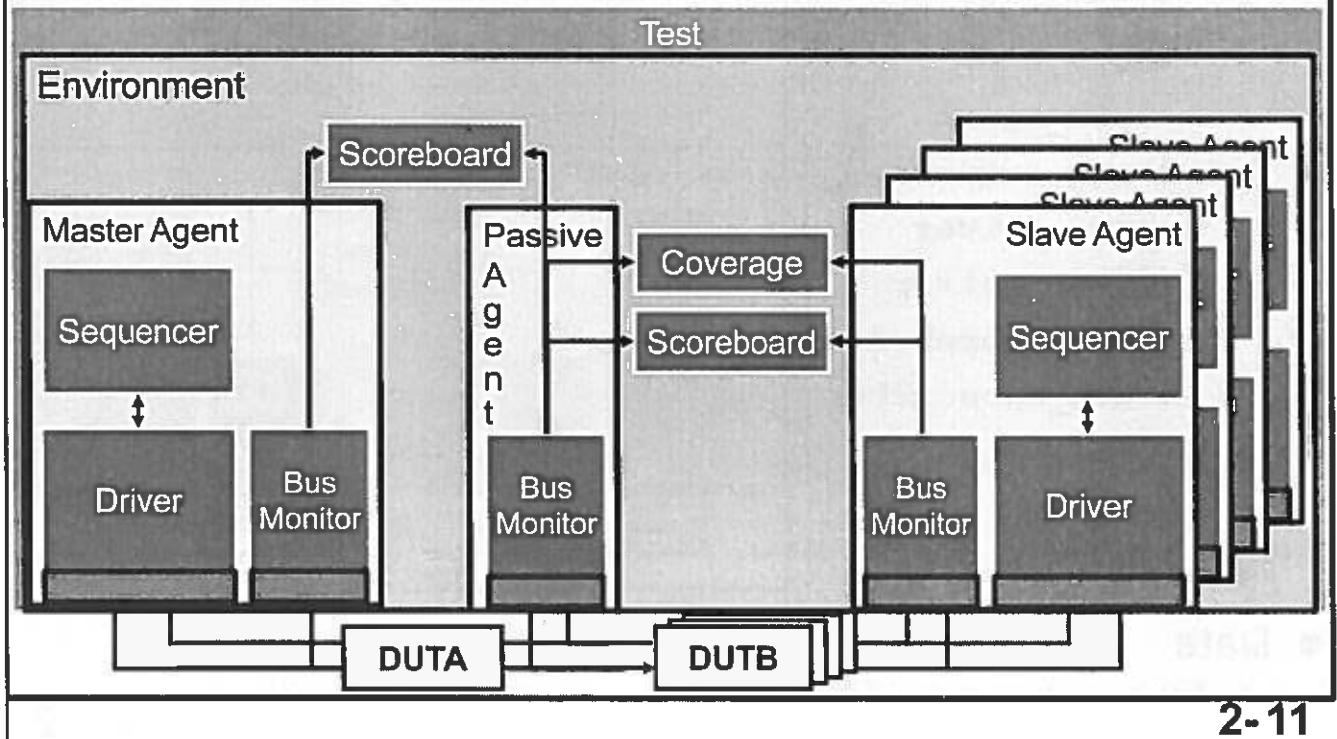
The protocol-specific blocks (sequencer, driver, monitor) are grouped together to create an agent that can be instantiated as a unit. This reduces the work to connect to a protocol.

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 the building blocks across test/projects

Top Level Harness

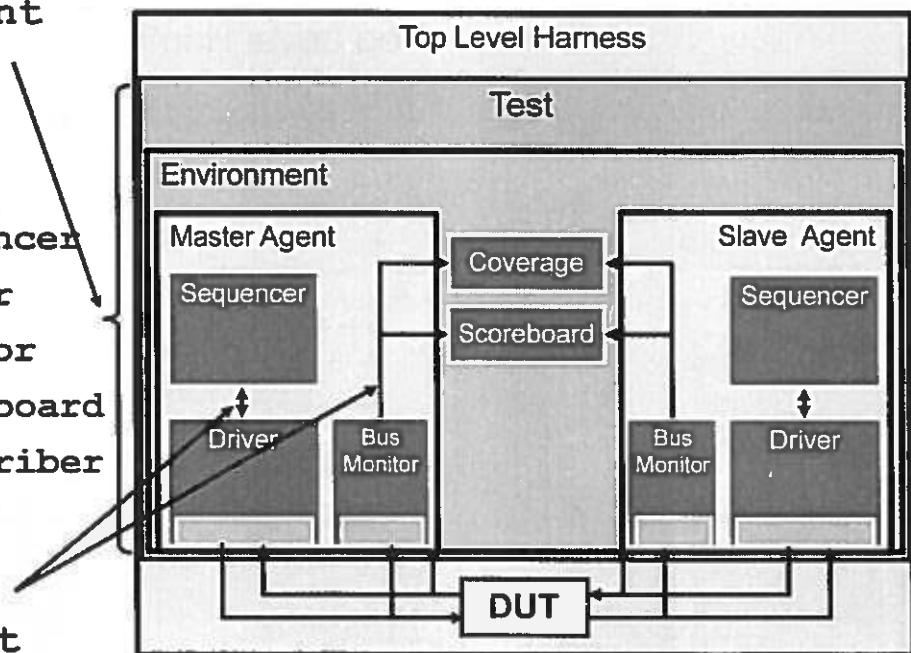


2-11

Structural Support in UVM

■ Structural & Behavioral

- `uvm_component`
 - `uvm_test`
 - `uvm_env`
 - `uvm_agent`
 - `uvm_sequencer`
 - `uvm_driver`
 - `uvm_monitor`
 - `uvm_scoreboard`
 - `uvm_subscriber`



■ Communication

- `uvm_*_port`
- `uvm_*_socket`

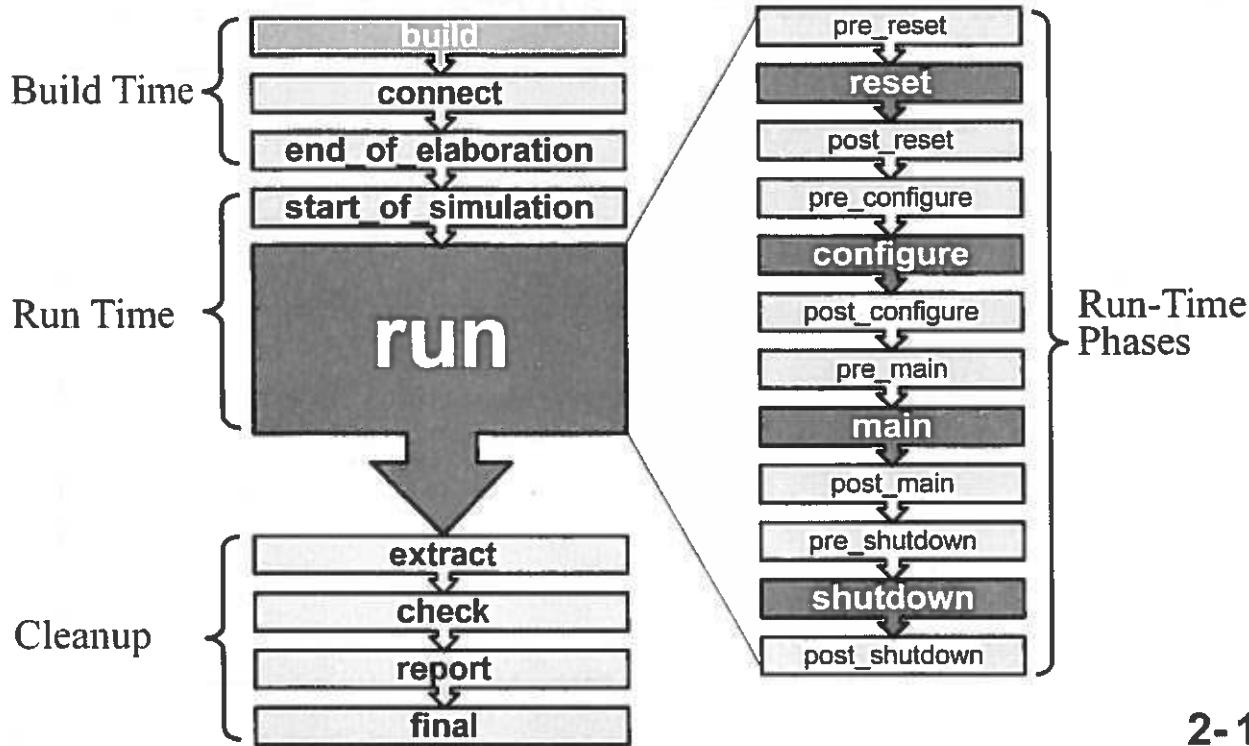
■ Data

- `uvm_sequence_item`

2-12

Component Phasing

- The run phase task executes concurrently with the scheduled run-time phase tasks



2-13

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 random configuration.

In the connect phase, the testbench connects TLM ports

In the end_of_elaboration phase, check the connections

In the start_of_simulation phase, you could print the testbench configuration

In the run phase task, the test is actually run (the scheduled run-time task phases execute in order)

In the extract phase, data is extracted from the testbench

In the check phase, any final checking is performed

In the report phase, the scoreboard and other checkers report the simulation results

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

The run phase and the pre_reset through post_shutdown phases are concurrently executed. Run phase starts at the same time as pre_reset phase. Run phase terminates at the same time that post_shutdown terminates.

Hello World Example

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

DUT functional verification code resides in one of the task phases

Execute test

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

Create and register test name

Message

2-14

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

The macro ``uvm_component_utils()` puts this class in a registry so that it can be run from the command line.

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

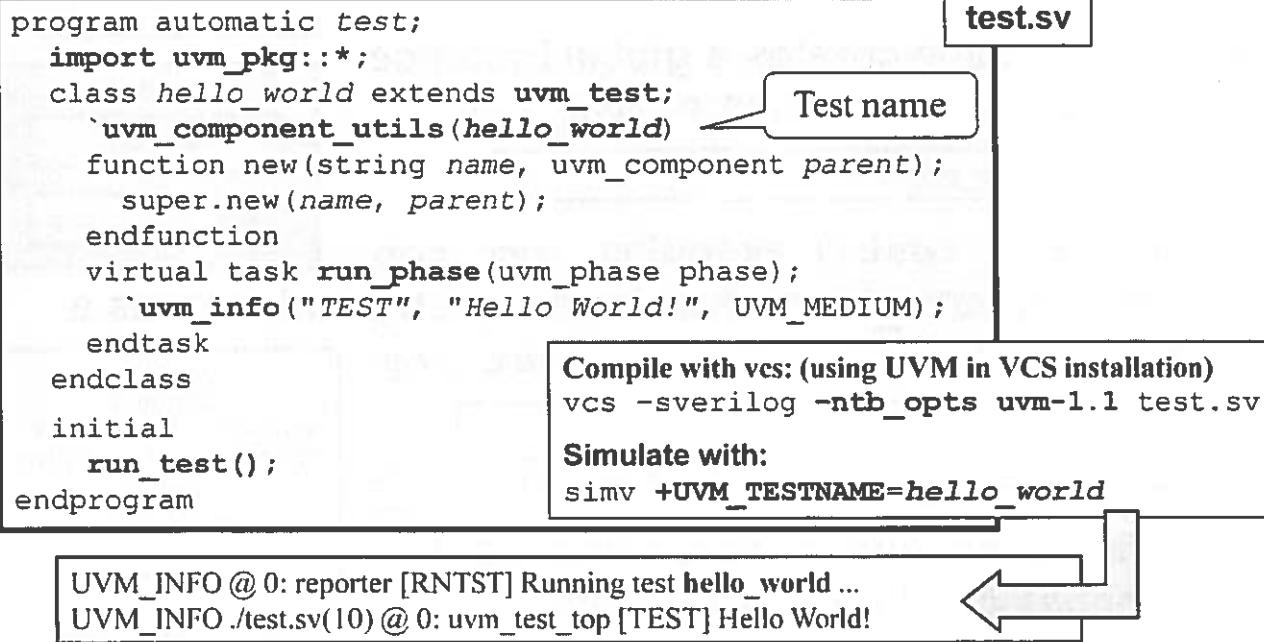
The actual message is printed with the ``uvm_info()` macro that has an ID, the message, and a verbosity level.

The `run_test()` global method starts the test execution.

Class `hello_world` does not need to be instantiated!

Compile and Simulate

- Compile with `-ntb_opts uvm-1.1` switch
- Specify test to run with `+UVM_TESTNAME` switch



2-15

Compile with VCS using custom UVM version

```
% 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-1.1/vcs ${VCS_HOME}/etc/uvm-1.1/vcs/uvm_custom_install_vcs_recorder.sv
```

Note: There are multiple switches possible for `-ntb_opt`: `uvm`, `uvm-ea`, `uvm-1.0`, `uvm-1.1` (and in the future `uvm-1.2`, `uvm-2.0` etc).

The reason for the existence of multiple switches is that UVM is still undergoing changes. UVM-1.0 EA was released in April 2010. UVM-1.0 was released in February 2011 and UVM-1.1 in June 2011. UVM-1.2 and UVM-2.0 will be released in the future.

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 2012.09's default UVM version is UVM-1.1. But, future VCS versions may default to future UVM version.

Inner Workings of UVM Simulation

- Macro registers the class in factory registry table

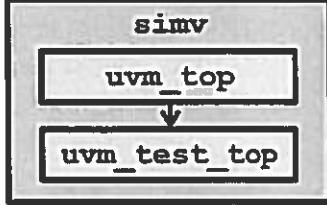
```
class hello_world extends uvm_test;  
  `uvm_component_utils(hello_world)
```

Registry table

"hello_world"

- UVM package creates a global instance of **uvm_root** class named **uvm_top**

```
import uvm_pkg::*;


```

- When **run_test()** executes, **uvm_top** retrieves **+UVM_TESTNAME** from registry and creates a child component called **uvm_test_top**

```
initial  
  run_test(); simv +UVM_TESTNAME=hello_world
```

build
connect
end_of_elaboration
start_of_simulation
run*
extract
check
report
final

- During execution, parent component manages children components' phase execution

2-16

The **uvm_pkg** package contains a global **uvm_root** object called **uvm_top**. It controls the phasing of all components including the test. It also provides mechanism for searching for components, printing component topology, and other global component structure-related functions.

When you call **run_test()**, it executes a global routine that calls **uvm_top.run_test()**. The **uvm_top.run_test()** method gets the test name from the command line (**+UVM_TESTNAME**) and constructs a component of that type. This component instance is called "**uvm_test_top**". **uvm_top** manages all component phase executing after **uvm_test_top** is constructed.

User Control of Reporting and Filtering

■ Print messages with UVM macros

```
'uvm_fatal("CFGERR", "Fatal message");
'uvm_error("RNDERR", "Error message");
'uvm_warning("WARN", "Warning message");
'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 always displayed

More verbose

Info messages need to specify verbosity

■ Verbosity filter defaults to UVM_MEDIUM

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

```
simv +UVM_VERBOSITY=UVM_DEBUG +UVM_TESTNAME=hello_world
```

2-17

Key Component Concepts: Parent-Child

■ Parent-child relationships

- Set up at component creation
- Establishes component phasing execution order
- Establishes component search path for component configuration and factory replacement
- Logical hierarchy – Not OOP or composition hierarchy

■ Phase execution order

- Each component follows the same sequence of phase execution

■ Search path allow tests to:

- Search and configure components
- Search and replace components in environment



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

2-18

UVM puts components in a logical hierarchical tree, based on component name, and the parent that is passed to the constructor.

This is **not** the OOP inheritance hierarchy, where **uvm_object** is the parent to **uvm_component**.

In this structural hierarchy, the test ("uvm_test_top") is at the top, the environment below, and components such as agents, drivers and monitors below that.

This structural hierarchy is important as it allows you to configure components, or replace them with derived objects, using 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

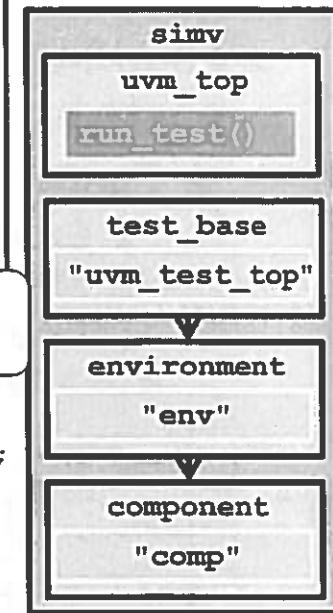
  class environment extends uvm_env;
    component comp;
    `uvm_component_utils(environment)
    function new(string name, uvm_component parent);
      virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        comp = component::type_id::create("comp", this);
      endfunction

  class component extends uvm_component;
    `uvm_component_utils(component)
    function new(string name, uvm_component parent);
      super.new(name, parent);
    endfunction
    virtual task component_task(...);
  endclass
```

Establish parent-child relationship at creation

Set parent

Parent handle



2-19

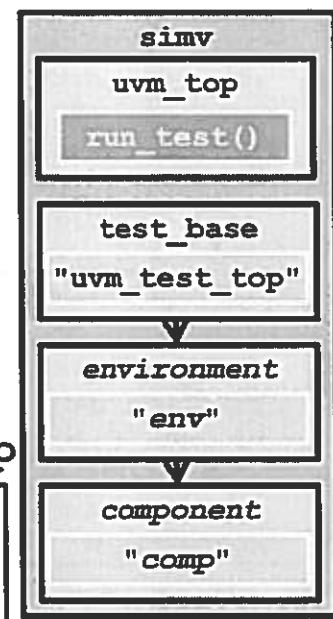
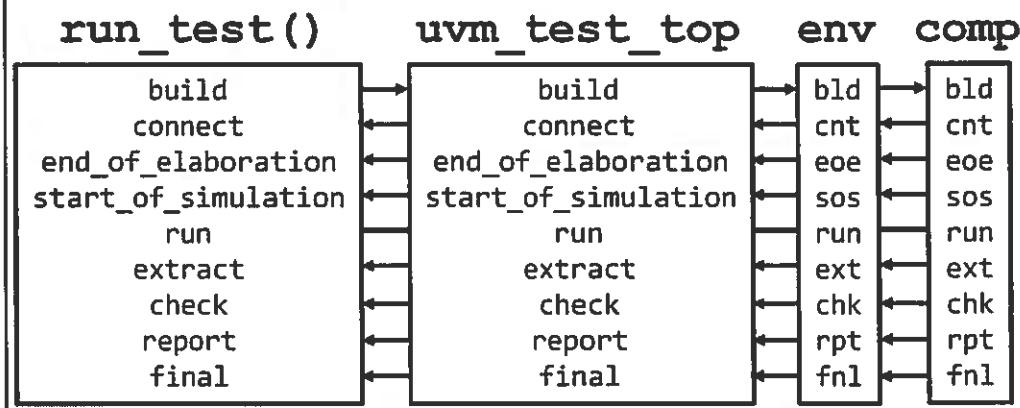
This slide shows a simple hierarchy. At the bottom is the component class. Its constructor has two arguments, the logical name, and its parent.

The next item is the environment which contains an instance of the component. In the call `component::type_id::create()`, the UVM factory builds an instance of the component class, under the environment.

At the top is base test class, which builds an instance of the environment.

Key Component Concepts: Phase

- Parent-child relationship dictates phase execution order
 - Functions are executed bottom-up
 - ◆ Except for build phase which is executed top-down
 - Tasks are forked into concurrent executing threads



2-20

The **run_test()** method runs all the phases of the test.

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

All the function phases are executed bottom up except build.

The build phase has to execute top down so that configuration settings set at the top (test) can propagate down to the components.

In some cases, components might not even get built if the configuration says they are not used in this simulation.

Key Component Concepts: Search & Replace

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

Simulate with:
simv +UVM_TESTNAME=test_new

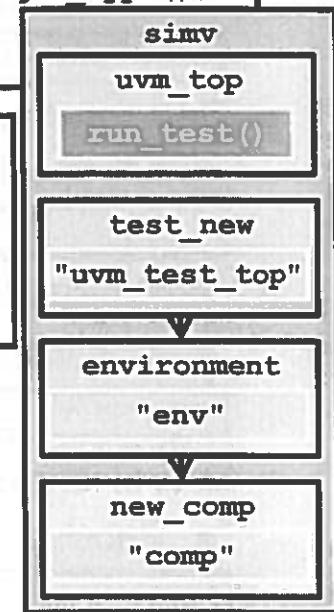
Use parent-child relationship to do search and replace for components

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

create() used to build component

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

Modify operation



2-21

An important concept in UVM is that your testbench classes such as the environment, drivers, etc, can be 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 drivers, but it can also allow you to replace an existing component with an extended one.

In this case the *component* class has been replaced by *new_comp*. At the top level, the test tells the UVM factory to override the *component* class with *new_comp*.

Now when the environment creates a *component*, it actually gets *new_comp*.

Key UVM Elements (1/2)

■ TLM library classes

- Classes for communication between components

■ Transaction class

- Encapsulates stimulus data structure

```
class packet extends uvm_sequence_item;
```

■ Sequence class

- Encapsulates transactions to be processed by driver

```
class packet_sequence extends uvm_sequence #(packet);
```

■ Sequencer class

- Executes sequence and passes transactions to driver

```
typedef uvm_sequencer #(packet) packet_sequencer;
```

■ Driver class

- Receives transaction from sequencer and drives DUT

```
class driver extends uvm_driver #(packet);
```

2-22

Key UVM Elements (2/2)

■ Agent class

- Encapsulates sequencer, driver and monitor on a per interface basis

```
class router_agent extends uvm_agent;
```

■ Environment class

- Encapsulates all components of testbench
- Connect components via TLM ports

```
class router_env extends uvm_env;
```

■ Test class

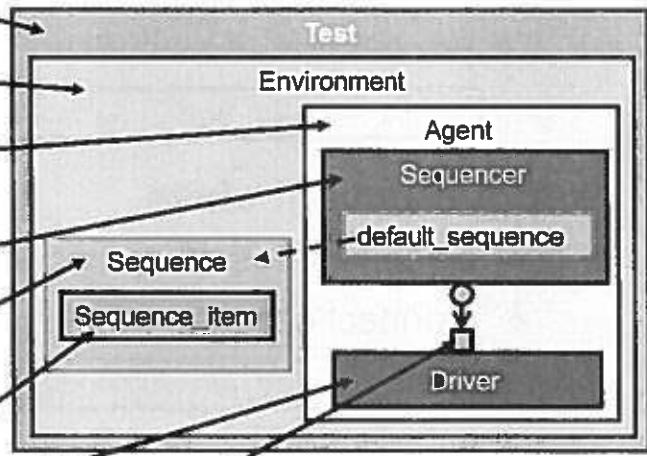
- Encapsulates environment objects
- Sets test-level configuration/modification

```
class test_base extends uvm_test;
```

2-23

Simple Example of Key UVM Elements

- **Test**
 - `uvm_test`
- **Environment**
 - `uvm_env`
- **Agent**
 - `uvm_agent`
- **Sequencer**
 - `uvm_sequencer`
- **Data class**
 - `uvm_sequence`
 - `uvm_sequence_item`
- **Driver (BFM)**
 - `uvm_driver`
- **Communication (TLM)**



For simplicity, monitor and other components are left off

2-24

Key UVM Testbench Elements: TLM

■ Communication between testbench components

- Unidirectional or bidirectional communication
- Blocking and non-blocking interfaces
- `put()` and `get()` functionality

Details in TLM Unit

■ Analysis ports for passive communication

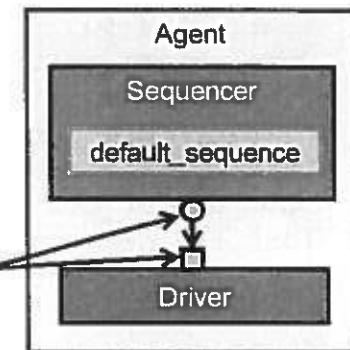
- Monitors, Scoreboards, Functional Coverage
- Always non-blocking, Broadcasting
- `write()` functionality

■ TLM Base Classes

- `uvm_*_port/_imp/export`
- `uvm_*_fifo_*`
- `uvm_*_socket`



Many UVM components
have built-in TLM ports

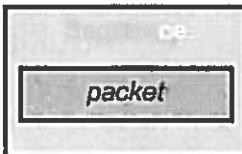


2-25

Key UVM Element: Transaction Class

■ Declare stimulus properties

- Enable all properties for randomization with `rand`
- Does not have phase methods
- Does not have `uvm_component` parent



■ Use macros to create support infrastructure

```
class packet extends uvm_sequence_item;
    rand bit[3:0] sa, da;
    rand bit[7:0] payload[$];
    constraint valid {payload.size() inside {[2:100]};}
    `uvm_object_utils_begin(packet)
        `uvm_field_int(sa, UVM_ALL_ON)
        `uvm_field_int(da, UVM_ALL_ON)
        `uvm_field_queue_int(payload, UVM_ALL_ON)
    `uvm_object_utils_end
    function new(string name = "packet");
        super.new(name);
    endfunction
endclass
```

Macro creates support methods:
`copy()`, `print()`, etc.

Default required

2-26

Transactions are created in UVM sequences and passed on to driver by the UVM sequencer, shown on an upcoming slide. Transaction classes must extend the `uvm_sequence_item` class.

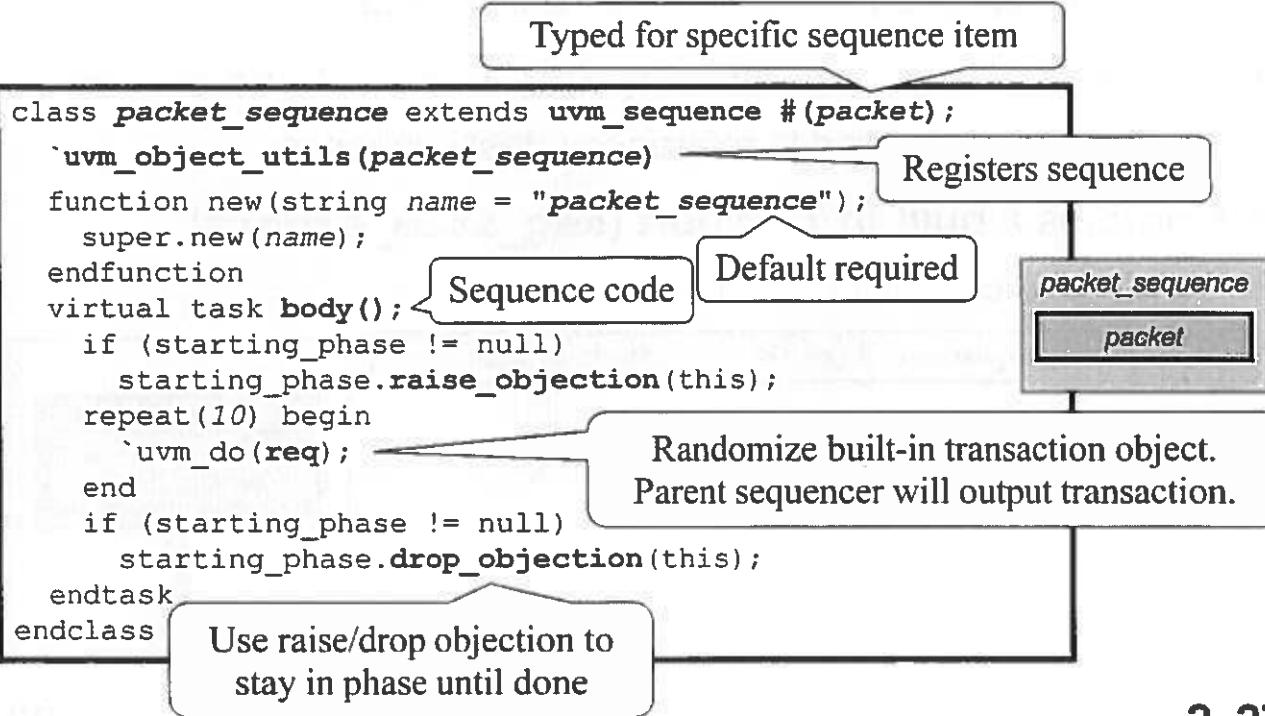
This transaction has a 4-bit source and destination addresses, and a queue of bytes to hold the payload. The constraint assures that the packets have a valid size.

The macros starting with ``uvm_field` process the listed field property in UVM methods such as `copy`, `clone`, `compare`, `print`, etc. These methods will be used in your tests.

The full set of macros and their usage are described in the Modeling Transactions section.

Key UVM Element: Sequence Class

- Stimulus transactions are created inside a UVM sequence extended from `uvm_sequence` class



2-27

A sequence consists of one or more sequence items. Each sequence item for the sequence is created in the `body()` task, often with pre-defined UVM macros such as `'uvm_do()`. This macro randomizes the sequence item and makes it available for driver to access.

The `uvm_sequence` class has a handle `req` that is typed to the sequence item parameter. You should use this built-in handle as the sequence item reference.

The `raise_objection()` / `drop_objection()` calls are to prevent the Run Time phase that the sequence is executed in from terminating before the sequence completes. Without these calls, if a Run Time phase consumes time and objection was not raised, then that phase will execute but terminates in 0 time. When the phase terminates, all child threads of that phase will be killed. One must check to make sure `starting_phase` is not null before raising and dropping objection. The `starting_phase` is only available to the parent sequence. If a sequence is a sub-sequence of another sequence, the `starting_phase` will be null.

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 answered (objections dropped).

The `raise/drop_objection` has an optional second string argument that's useful for debugging.
`starting_phase.raise_objection(this, $sformatf("Starting %s", this.get_name()));`

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

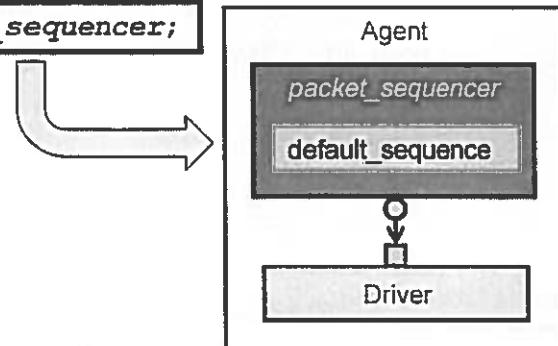
`UVM_INFO @ 0.0s: main [OBJTN_TRC] Object uvm_test_top.env.i_agent.seqr.packet_sequence raised 1 objection(s) (Starting packet_sequence): count=1 total=1`

Key UVM Element: Sequencer Class

- Use `uvm_sequencer` class
 - Parameterized to a chosen transaction class
- Contains a sequence descriptor ("`default_sequence`")
 - Must be configured to reference a test sequence
- Contains a built-in TLM port (`seq_item_export`)
 - Must be connected to a driver

```
typedef uvm_sequencer #(packet) packet_sequencer;
```

Must be typed for a sequence item class



2-28

`uvm_sequencer` is a parameterized class and rarely needs to be extended.

The code above creates a new type, a specialization of `uvm_sequencer` that works on *packet* objects.

A common typo is mixing up `sequence` and `sequencer`. You will get odd compile errors because of the missing 'r'.

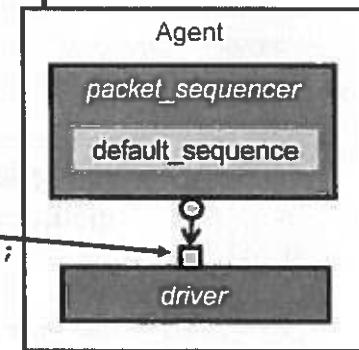
Key UVM Element: Driver Class

■ Driver should extend from `uvm_driver` class

- `uvm_driver` class has built-in sequence item handle (`req`) and TLM port to communicate with the sequencer

```
class driver extends uvm_driver #(packet);
  `uvm_component_utils(driver)
  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfunction
  virtual task run_phase(uvm_phase phase);
    forever begin
      seq_item_port.get_next_item(req);
      `uvm_info("DRV_RUN", req.sprint(), UVM_MEDIUM);
      seq_item_port.item_done();
    end
  endtask
endclass
```

Must be typed for a sequence item class



Driver should implement `run phase` to handle sequence item received from any Run Time task phases

2-29

A UVM driver typically receives sequence items from the sequencer and processes them. This one just prints them out.

A driver is a testbench component, and needs to know where it resides in the testbench hierarchy (test>env>agent>driver).

This why it has an parent argument to the constructor.

During the `run_phase`, the driver does not raise or drop objection. The objection mechanism will be controlled within the sequences.

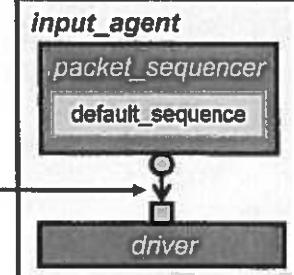
The `sprint()` function returns a formatted string with the contents of the object. The function was populated for you by ``uvm_field`` macro in the sequence item class (`packet`).

Key UVM Element: Agent Class

■ 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;
    packet_sequencer seqr; driver drv;
    // utils macro and constructor not shown
    virtual function void build_phase(uvm_phase phase); //super not shown
        seqr = packet_sequencer::type_id::create("seqr", this);
        drv = driver::type_id::create("drv", this);
    endfunction
    In build phase, use factory create()
method to construct components
    virtual function void connect_phase(uvm_phase phase);
        drv.seq_item_port.connect(seqr.seq_item_export);
    endfunction
endclass
In connect phase, connect built-in TLM ports
```



2-30

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. All **component** classes must call **super.build_phase()** in the **build_phase()** method. The base class **build_phase()** method will update all configuration field for the component. For all other methods, such as **connect_phase**, the call to the super method is optional.

Within the **build_phase** method, whenever a child component is to be constructed, call the child component'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.

Key UVM Element: Environment Class

■ Encapsulate agent in environment class

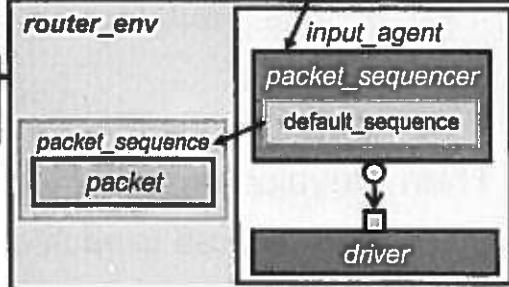
- Only agent for now. (Scoreboard and others in later units)
- Configure the agent sequencer's `default_sequence`

```
class router_env extends uvm_env; // Extend from uvm_env
  input_agent i_agent;
  // utils macro and constructor not shown
  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);

    i_agent = input_agent::type_id::create("i_agent", this);
    uvm_config_db #(uvm_object_wrapper)::set(this, ".seqr.main_phase",
      "default_sequence", packet_sequence::get_type());
  endfunction
endclass
```

Set agent's sequencer to execute a **default sequence**

In build phase,
create agent object



2-31

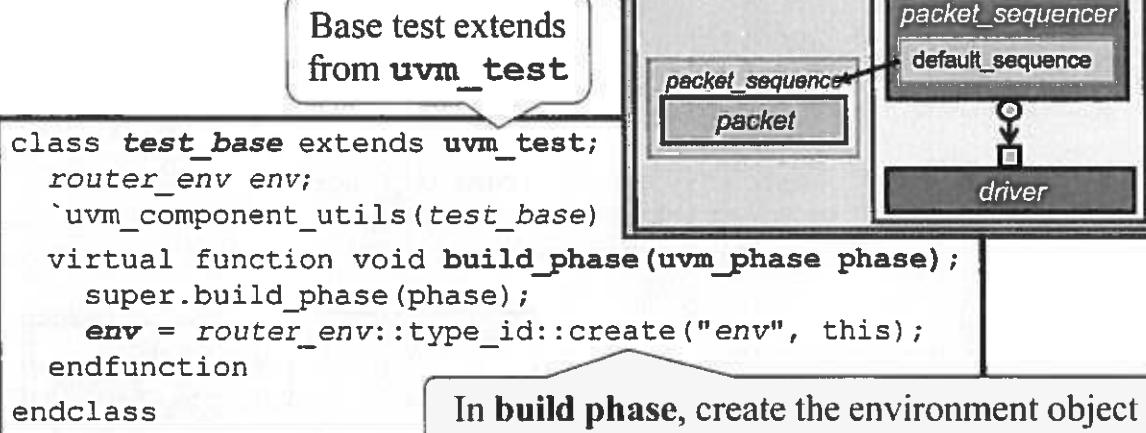
UVM maintains a database of settings. The call to `uvm_config_db::set()` tells the sequencer to execute the `packet_sequence` in the `main phase`.

Key UVM Element: Test Class

■ Develop Tests In Two Steps

- Create default test (unconstrained)

- ◆ `test_base` extending from the `uvm_test` base class



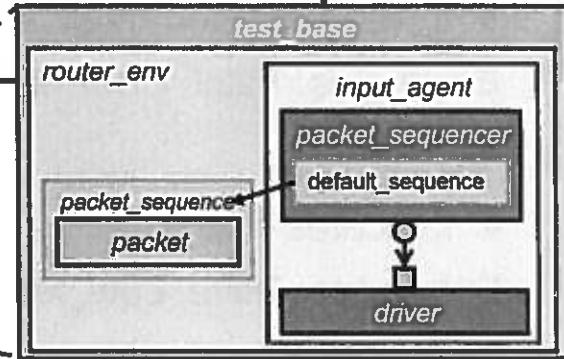
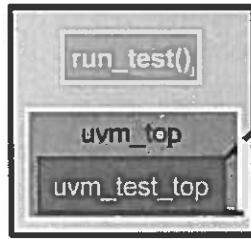
- Then, develop targeted tests extending from `test_base`
- ◆ Will be covered in each of the subsequent units

2-32

Test Program

- Execute test in program's initial block

```
program automatic test;  
...  
initial begin  
    run_test();  
end  
endprogram
```



Simulate with:

```
simv +UVM_TESTNAME=test_base
```

2-33

Synopsys recommends using a program block to reduce the race conditions between the testbench and the design under test (RTL) in modules. UVM works with the testbench in a program or module.

Key UVM Testbench Elements: UVM Report

- Every UVM report has a severity, verbosity and simulation handling specification
 - Each independently specified and controlled
- Severity
 - Indicates importance**Examples:** Fatal, Error, Warning, Info
- Verbosity
 - Indicates filter level**Examples:** None, Low, Medium, High, Full, Debug
- Action
 - Controls simulator behavior**Examples:** Exit, Count, Display, Log, Call Hook, No Action

2-34

```
'define uvm_file `__FILE__
`define uvm_line `__LINE__

`define uvm_info(ID, MSG, VERBOSITY) begin \
    if (uvm_report_enabled(VERBOSITY, UVM_INFO, ID)) \
        uvm_report_info(ID, MSG, VERBOSITY, `uvm_file, `uvm_line); \
    end
`define uvm_warning(ID,MSG) begin \
    if (uvm_report_enabled(UVM_NONE,UVM_WARNING, ID)) \
        uvm_report_warning (ID, MSG, UVM_NONE, `uvm_file, `uvm_line); \
    end
`define uvm_error(ID,MSG) begin \
    if (uvm_report_enabled(UVM_NONE,UVM_ERROR, ID)) \
        uvm_report_error (ID, MSG, UVM_NONE, `uvm_file, `uvm_line); \
    end
`define uvm_fatal(ID,MSG) begin \
    if (uvm_report_enabled(UVM_NONE,UVM_FATAL, ID)) \
        uvm_report_fatal (ID, MSG, UVM_NONE, `uvm_file, `uvm_line); \
    end
```

Embed 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 in Configuration randomization");
    end
endfunction
```



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

Severity

Time

ID

MSG

File line no.

Object name

2-35

Alternatives to the macro calls are the following global methods:

```
uvm_report_fatal(string ID, string MSG, int verbosity = UVM_NONE, string filename = "", int line = 0);
uvm_report_error(string ID, string MSG, int verbosity = UVM_LOW, string filename = "", int line = 0);
uvm_report_warning(string ID, string MSG, int verbosity = UVM_MEDIUM, string filename = "", int line = 0);
uvm_report_info(string ID, string MSG, int verbosity = UVM_MEDIUM, string filename = "", int line = 0);
```

The issue with these methods is that because they are methods, if one were to format a string (\$sformatf() or obj.sprint()) within the argument of the report method, the formatting would execute whether or not the message get filtered out. The result is a slow down of simulation.

Example:

```
uvm_report_info("DEBUG", obj.sprint());
```

In the example above, the sprint() will always execute.

The macros will do the filtering first before executing the above equivalent methods.

If using the report methods directly, user must set the filename argument to `__FILE__ and the line argument to `__LINE__, otherwise, the file & line no. will not be displayed. Alternatively, user can set the UVM_USE_FILE_LINE compile option to set these arguments to `__FILE__ and `__LINE__ automatically, as has been done for the `uvm_info/warning/error/fatal macro.

Embed Report Messages

■ Message ID (free text)

- Allows easy search (“grep”) in the simulation log
- Total count is reported at the end of simulation:

```
** Report counts by id
[Comparator Match]    154
[Comparator Mismatch]   6
[DRV_RUN]            160
[Got Input Packet]    160
[Got Output Packet]   160
[MISCMP]             12
[RNTST]              1
[Scoreboard Report]    1
```

2-36

Controlling Report Message Verbosity

- Default Verbosity is UVM_MEDIUM

- Set with run-time switch

```
simv +UVM_VERBOSITY=UVM_HIGH
```

Can be:

LOW, MEDIUM, HIGH, FULL, DEBUG,
UVM_LOW, UVM_MEDIUM, UVM_HIGH, UVM_FULL, UVM_DEBUG

```
typedef enum {
    UVM_NONE      = 0,
    UVM_LOW       = 100,
    UVM_MEDIUM    = 200,
    UVM_HIGH      = 300,
    UVM_FULL      = 400,
    UVM_DEBUG     = 500
} uvm_verbosity;
```

- Set verbosity for a component or hierarchy:

```
drv.set_report_verbosity_level(verbosity);
env.set_report_verbosity_level_hier(verbosity);
```

- Exception:

`uvm_fatal, `uvm_error, and `uvm_warning

can not be filtered out via set_report_verbosity

2-37

The verbosity level describes how many messages a testbench writes to the log. When set at the default of UVM_MEDIUM, any message that uses UVM_HIGH or above is filtered out. A message with the verbosity level UVM_NONE can't be disabled by setting the level.

You can set the level with the runtime switch `+verbosity=level` or `+UVM_VERBOSITY=level` where level can be `LOW, MEDIUM, HIGH, FULL, DEBUG, UVM_LOW, UVM_MEDIUM, UVM_HIGH, UVM_FULL, UVM_DEBUG`.

You can set a uvm_component's verbosity with `set_report_verbosity_level(level)`, or hierarchically set the level with `set_report_verbosity_level_hier(level)`

```
class packet_sequence extends uvm_sequence...
task body();
    `uvm_info("TEST", "Macro", UVM_HIGH);           // Won't print
    m_sequencer.uvm_report_info("TEST", "Macro", UVM_HIGH); // Prints!
endtask
endclass

// Set in test
sequencer.set_report_verbosity_level(UVM_HIGH);
```

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, call <code>set_report_max_quit_count()</code>
UVM_DISPLAY	Display message on console
UVM_LOG	Captures message in a named file
UVM_CALL_HOOK	Calls callback method
UVM_NO_ACTION	Do nothing

2-38

```
typedef int uvm_action;
```

```
typedef enum
{
    UVM_NO_ACTION      = 'b000000,
    UVM_DISPLAY        = 'b000001,
    UVM_LOG            = 'b000010,
    UVM_COUNT          = 'b000100,
    UVM_EXIT           = 'b001000,
    UVM_CALL_HOOK      = 'b010000,
    UVM_STOP           = 'b100000
} uvm_action_type;
```

- | | |
|---------------|---|
| UVM_NO_ACTION | - No action is taken |
| UVM_DISPLAY | - Sends the report to the standard output |
| UVM_LOG | - Sends the report to the file(s) for this (severity, id) pair |
| UVM_COUNT | - Counts the number of reports with the COUNT attribute.
When this value reaches max_quit_count, the simulation terminates |
| UVM_EXIT | - Terminates the simulation immediately. |
| UVM_CALL_HOOK | - Callback the report hook methods. |

Modify Report Message Action in Test

■ Actions can be modified by user

- In `start_of_simulation` phase

```
set_report_severity_action(Severity, Action);
set_report_id_action(ID, Action);
set_report_severity_id_action(Severity, ID, Action);
```

■ Priority:

```
set_report_severity_action()      (lowest)
set_report_id_action()
set_report_severity_id_action()  (highest)
```

Example:

```
drv.set_report_severity_action(UVM_FATAL, UVM_LOG | UVM_DISPLAY);
drv.set_report_id_action("CFG_ERROR", UVM_NO_ACTION);
drv.set_report_severity_id_action(UVM_ERROR, "CFG_ERROR", UVM_EXIT);
```

2-39

<code>UVM_NO_ACTION</code>	No action is taken
<code>UVM_DISPLAY</code>	Send to standard output and captured in log file when you run <code>simv -l log</code>
<code>UVM_LOG</code>	Sends the report to the file(s) for this (severity, id) pair
<code>UVM_EXIT</code>	Terminates the simulation immediately.
<code>UVM_CALL_HOOK</code>	Callback the report hook methods
<code>UVM_COUNT</code>	Counts the number of reports with the COUNT attribute. When this value reaches <code>max_quit_count</code> , the simulation terminates

The message macros are defined as:

```
'define uvm_info(ID, MSG, VERTOSITY) \
begin \
  if (uvm_report_enabled(VERTOSITY, UVM_INFO, ID)) \
    uvm_report_info(ID, MSG, VERTOSITY, `uvm_file, `uvm_line); \
end
```

If you use the macro in an if-else statement, you must enclose it in a begin-end block.

```
if(cond) begin
  `uvm_info(...);
end
else ...
```

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.agent1.*,_ALL_,UVM_FULL,build`
`+uvm_set_verbosity=uvm_test_top.env.agent1.*,_ALL_,UVM_FULL,time,800`

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

- Can be `UVM_NO_ACTION` or a | separated list of the other actions

`+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_ERROR,UVM_WARNING`

2-40

User Filterable Code Block

- Control filtering of block of code

- Based on the uvm_report mechanism

Verbosity

Severity

ID

```
if (uvm_report_enabled(UVM_HIGH, UVM_INFO, "CODE_BLOCK")) begin  
    $display("Code Block to be filtered");  
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
```

2-41

Misc: 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();
  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase); // simplified code
    if (clp.get_arg_value("+packet_item_count=", value))
      uvm_config_db#(int)::set(this, "*.*", "item_count",
                                value.atoi());
  endfunction
  virtual function void start_of_simulation_phase(uvm_phase phase);
    string tool, ver, args[$], m;
    tool = clp.get_tool_name();
    ver = clp.get_tool_version();
    `uvm_info("TOOL_INFO",
              $sformatf("Tool: %s, Version : %s", tool, ver), UVM_LOW);
    clp.get_args(args);
    foreach (args[i])
      $sformat(m, {m, args[i], " "});
    `uvm_info("ARGS", $sformatf("Sim cmd is:\n%s", m), UVM_LOW);
  endfunction
endclass
```

2-42

simv +UVM_TESTNAME=test_base +packet_item_count=20

Misc: Objection Run-Time Argument

- User can debug raising/dropping of objections with
+UVM_OBJECTION_TRACE

Simulate with:

```
simv +UVM_TESTNAME=test_base +UVM_OBJECTION_TRACE
```



```
UVM_INFO @ 0.0ns: main [OBJTN_TRC] Object uvm_test_top.env.agent.seqr.packet_sequence raised 1 objection(s): count=1 total=1
UVM_INFO @ 0.0ns: main [OBJTN_TRC] Object uvm_test_top.env.agent.seqr added 1 objection(s) to its total (raised from source object ): count=0 total=1
UVM_INFO @ 0.0ns: main [OBJTN_TRC] Object uvm_test_top.env.agent added 1 objection(s) to its total (raised from source object ): count=0 total=1
UVM_INFO @ 0.0ns: main [OBJTN_TRC] Object uvm_test_top.env added 1 objection(s) to its total (raised from source object ): count=0 total=1
UVM_INFO @ 0.0ns: main [OBJTN_TRC] Object uvm_test_top added 1 objection(s) to its total (raised from source object
uvm_test_top.env.agent.seqr.packet_sequence): count=0 total=1
UVM_INFO @ 0.0ns: main [OBJTN_TRC] Object uvm_top added 1 objection(s) to its total (raised from source object
uvm_test_top.env.agent.seqr.packet_sequence): count=0 total=1
...
UVM_INFO @ 0.0ns: main [OBJTN_TRC] Object uvm_test_top.env.agent.seqr.packet_sequence dropped 1 objection(s): count=0 total=0
UVM_INFO @ 0.0ns: main [OBJTN_TRC] Object uvm_test_top.env.agent.seqr.packet_sequence all_dropped 1 objection(s): count=0 total=0
UVM_INFO @ 0.0ns: main [OBJTN_TRC] Object uvm_test_top.env.agent.seqr subtracted 1 objection(s) from its total (dropped from source
object ): count=0 total=0
...
UVM_INFO @ 0.0ns: main [OBJTN_TRC] Object uvm_top subtracted 1 objection(s) from its total (dropped from source object
uvm_test_top.env.agent.seqr.packet_sequence): count=0 total=0
UVM_INFO @ 0.0ns: main [OBJTN_TRC] Object uvm_top subtracted 1 objection(s) from its total (all_dropped from source object
uvm_test_top.env.agent.seqr.packet_sequence): count=0 total=0
UVM_INFO @ 0.0ns: reporter [UVMTOP] UVM testbench topology:
```

2-43

Unit Objectives Review

Having completed 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

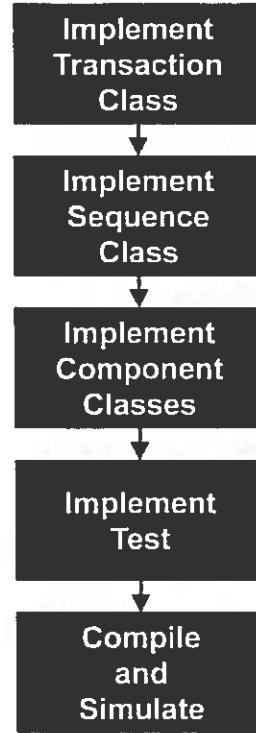
2-44

Lab 1 Introduction

Implement transaction, components, and test



45 min



2-45

Appendix

UVM Class Tree

VCS Support for UVM

Store UVM Report Messages in File

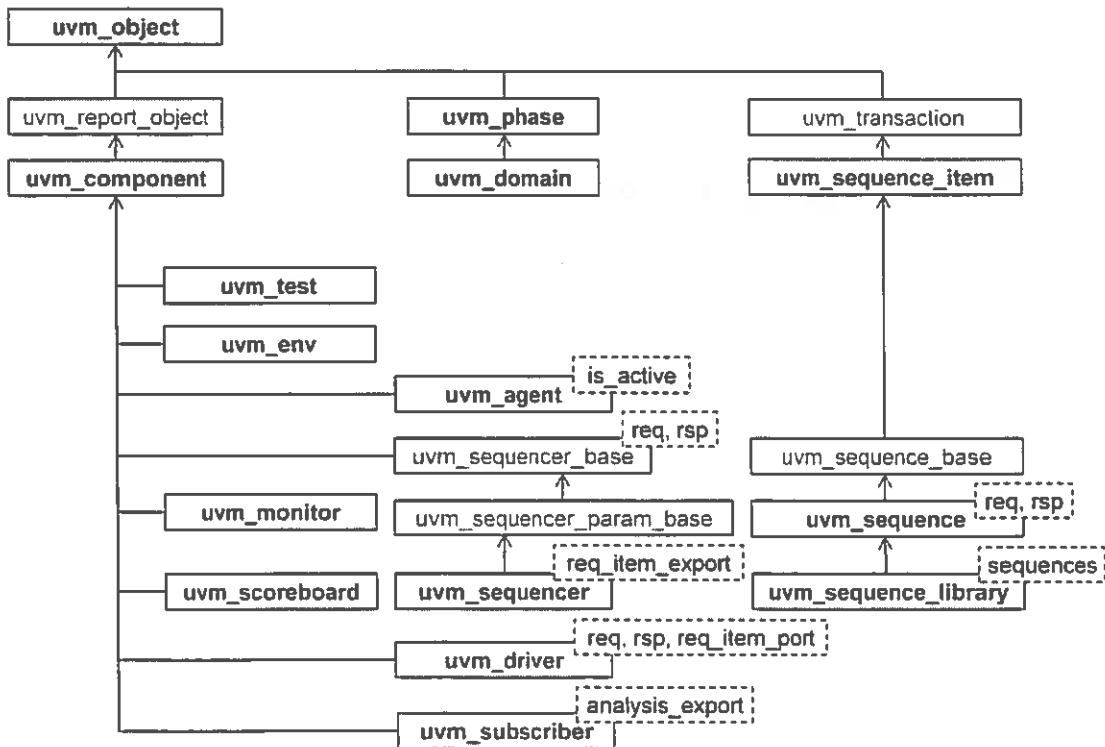
Calling UVM Messaging From Modules

Demote/Promote UVM Messages

uvmgen

UVM Class Tree

UVM Class Tree



2-48

VCS Support for UVM

Synopsys UVM Support

- **UVM 1.1 supported in VCS 2011.03-SP1 onwards**
 - `$VCS_HOME/etc/uvm-1.1/` (UVM source code)
 - Interoperability package:
`$VCS_HOME/doc/examples/testbench/sv/uvm_vmm_interop_kit`
- **VCS/VCS-MX Documentation**
(`$VCS_HOME/doc/UserGuide/pdf/uvm_*`)
 - UVM 1.1 Users Guide
 - UVM 1.1 Class Reference
 - UVM RAL (Register Abstraction Layer) User Guide
- **SolvNet (requires login)**
 - <https://solvnet.synopsys.com/> > Search for ‘UVM’ in Articles

2-50

Compiling UVM with VCS

■ Single compile flow

```
* vcs -sverilog file.sv ... -ntb_opts uvm-1.1 ...
```

Compile UVM library
first with no source files

■ UUM compile flow

```
* vlogan -sverilog -work work1 -ntb_opts uvm-1.1
* vlogan -sverilog -work work1 -ntb_opts uvm-1.1 file1.v
* vlogan -sverilog -work work2 -ntb_opts uvm-1.1 file2.v
* vhdlan -work work3 file3.vhd
* vcs top ... -ntb_opts uvm-1.1
```

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

2-51

DVE UVM Macro Debugging

```
26 `define APB_RW_SV
27
28 class eph_rw extends uvm_sequence_item;
29
30   typedef enum {READ, WRITE} kind_e;
31   rand bit [31:0] addr;
32   rand logic [31:0] data;
33   rand kind_e kind;
34
35   `uvm_object_utils_begin(epb_rw)
36     `uvm_field_int(addr, UVM_ALL_ON | UVM_NOPACK);
37     `uvm_field_int(data, UVM_ALL_ON | UVM_NOPACK);
38     `uvm_field_enum(kind_e,kind, UVM_ALL_ON | UVM_NOPACK);
39   begin
40     case (what__)
41       UVM_CHECK_FIELDS:
42         _n_uvm_status_container.do_field_check("kind", this);
43       UVM_COPY:
44         begin
45           if(local_data_ == null) return;
46           if(!((UVM_ALL_ON | UVM_NOPACK) & UVM_NOCOPY)) kind = local_data_.kind;
47         end
48       UVM_COMPARE:
49         begin
50           if(local_data_ == null) return;
51           if(!((UVM_ALL_ON | UVM_NOPACK) & UVM_NOCOMPARE)) begin
52             if(kind != local_data_.kind) begin
53               _n_uvm_status_container.do_field_check("kind", this);
54             end
55           end
56         end
57     endcase
58   end
59
60   `uvm_object_utils_end(epb_rw)
```

UVM Object Utils Macros

2-52

DVE UVM Transaction Level Debugging

In uvm-1.1 transaction recording is enabled by default

■ Simple compile flow

```
% vcs -sverilog -ntb_opts uvm-1.1 -debug ...
```

```
% simv +UVM_TR_RECORD +UVM_LOG_RECORD ...
```

Dumps UVM sequence items to VPD

Dumps UVM log messages to VPD

■ UUM compile flow

```
% vlogan ... -ntb_opts uvm-1.1 -debug -lca ...
% vlogan ... -ntb_opts uvm-1.1 -debug -lca file.sv
% vcs <top_module> ... $VCS_HOME/etc/uvm-1.1/dpi/uvm_dpi.cc
```

```
% simv +UVM_TR_RECORD +UVM_LOG_RECORD ...
```

■ Recording can be disabled with **-tld_logoff**

```
% vcs -sverilog -ntb_opts uvm-1.1 -debug -tld_logoff ...
```

2-53

VCS enabled UVM STACKTRACE

```
./simv  
+UVM_STACKTRACE[=default|all|info|warn|error|fatal]
```

```
#0 in \uvm_report_server::process_report at /disk/vcs/etc/uvm-1.1/base/uvm_report_server.svh:370  
#1 in \uvm_report_server::report at /disk/vcs/etc/uvm-1.1/base/uvm_report_server.svh:294  
#2 in \uvm_report_handler::report at /disk/vcs/etc/uvm-1.1/base/uvm_report_handler.svh:345  
#3 in \uvm_report_object::uvm_report_error at /disk/vcs/etc/uvm-1.1/base/uvm_report_object.svh:129  
#4 in \uvm_component::set_name at /disk/vcs/etc/uvm-1.1/base/uvm_component.svh:1908  
  
#5 in \test::run_phase at test.sv:15  
#6 in \uvm_run_phase::exec_task at /disk/vcs/etc/uvm-1.1/base/uvm_common_phases.svh:258  
#7 in \uvm_task_phase::execute at /disk/vcs/etc/uvm-1.1/base/uvm_task_phase.svh:145  
#8 in \uvm_phase::execute_phase at /disk/vcs/etc/uvm-1.1/base/uvm_phase.svh:1203  
  
UVM ERROR /disk/vcs/etc/uvm-1.1/base/uvm_component.svh(1908) @ 1: uvm test top  
[INVSTNM] It is illegal to change the name of a component. The component  
name will not be changed to "foo"
```

UVM within VCS gives users full STACK for user to see which file/line is not library code for them to investigate.

FYI, User is calling `set_name()` function at test.sv line 15

2-54

VMM to UVM Mapping

- `uvm_info/warning/error
- uvm_sequence_item
- uvm_component
- uvm_agent
- uvm_env
- uvm_callback
- uvm_tlm_fifo
- uvm_analysis_port
- uvm_event
- uvm_sequencer
- “Virtual” uvm_sequencer
- `vmm_note/warning/error
- vmm_data
- vmm_xactor
- vmm_subenv, vmm_group
- vmm_env, vmm_group
- vmm_callback
- vmm_channel
- vmm_tlm_analysis_port
- vmm_notify
- vmm_scenario_gen
- vmm_ms_scenario_gen

2-55

Store UVM Report Messages in File

Store Report Message in File

- Message Action set to UVM_LOG causes the report message to be stored in file

```
set_report_default_file(UVM_FILE);
set_report_severity_file(Severity, UVM_FILE)
set_report_id_file(ID, UVM_FILE);
set_report_severity_id_file(Severity, ID, UVM_FILE);
```

Example:

```
UVM_FILE log, err_log;
log      = $fopen("log", "w");
err_log = $fopen("err.log", "w");
set_report_default_file(log);
set_report_severity_file(UVM_FATAL, err_log);
set_report_severity_action(UVM_FATAL, UVM_EXIT | UVM_LOG | UVM_DISPLAY);
set_report_id_file("Trace", UVM_LOG | UVM_DISPLAY)
```

Combine multiple actions with bit-wise or

2-57

The example above shows how messages can be written to two separate files, *log* and *err.log*.

<code>set_report_default_file</code>	By default, all messages are written to <i>log</i> .
<code>set_report_severity_file</code>	Fatal messages are written to <i>err.log</i> and not <i>log</i>
<code>set_report_severity_action</code>	Fatal messages are written to the <i>log</i> , the <i>display</i> (<i>simv.log</i>), and causes simulation to exit.
<code>set_report_id_file</code>	Any message with ID="Trace" will be written to the <i>log</i> and the <i>display</i> .

Calling UVM Messaging From Modules

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 reg[100*8:1] ID,
                          input reg[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)
```

2-59

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(input reg reset_1, ...);
    parameter DISPLAY_WIDTH = 1000;
    reg [DISPLAY_WIDTH*8:1] message;

    always @(negedge reset_1) begin
        message = "reset asserted";
        `G_UVM_INFO("RESET", message, `G_UVM_HIGH);
    end
endmodule
```

2-60

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 below.

Demote/Promote UVM Messages

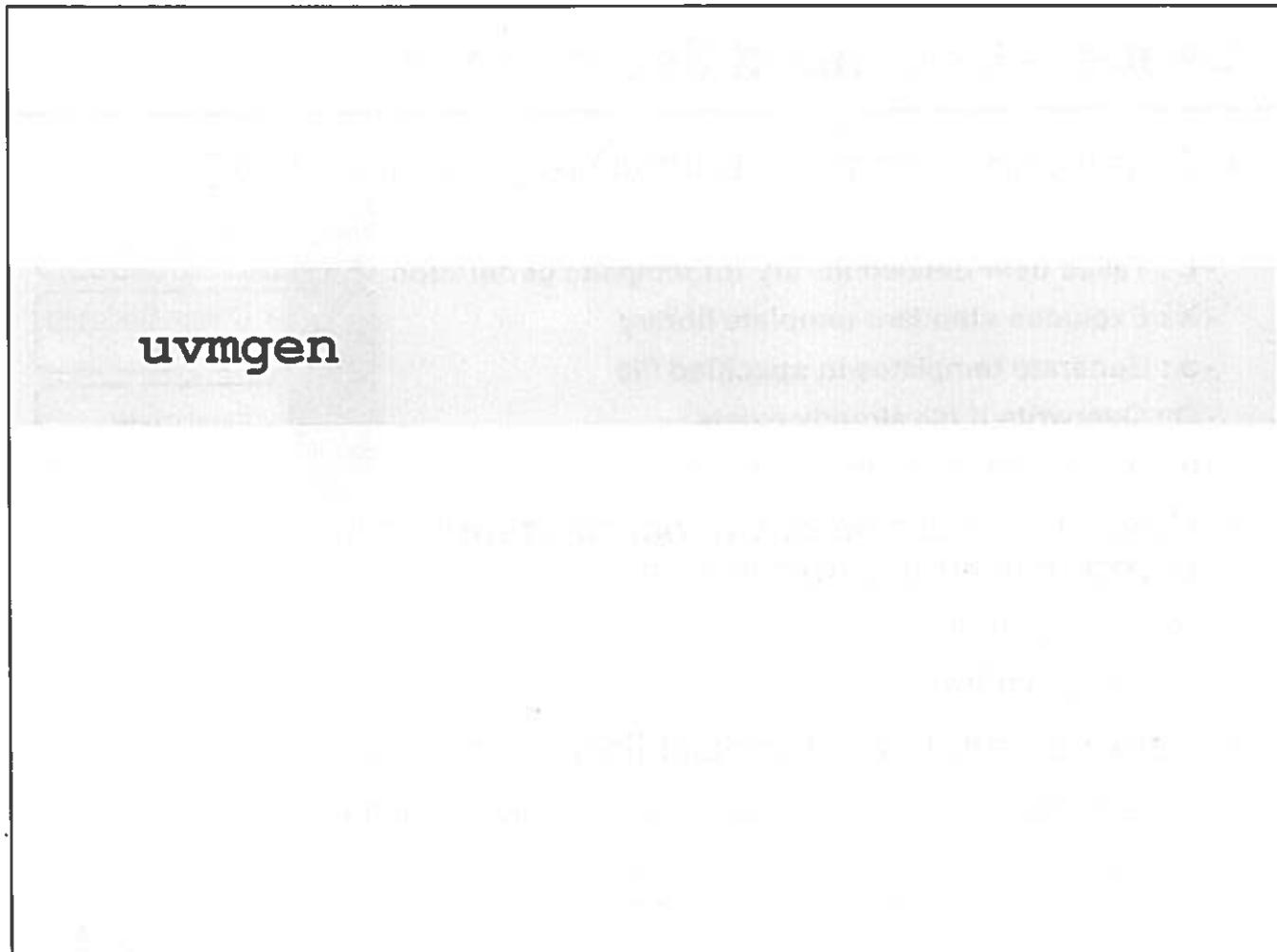
Demote/Promote UVM Report Messages

```
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);
  endfunction
endclass
```

2-62

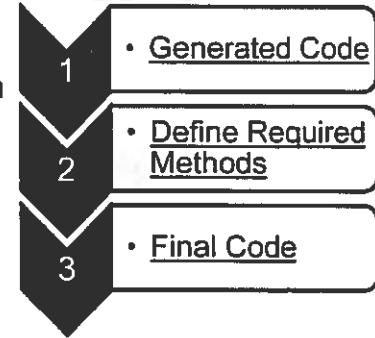
uvmgen



Usage – Command line options

■ Command: uvmgen [-L libdir] [-X] [-o fname] [-O]

- L : Takes user defined library for template generation
- X : Excludes standard template library
- o : Generate templates in specified file
- O : Overwrite if file already exists
- q : Quick mode to generate complete environment



■ User must set one of the below standard library path before running uvmgen script:

- UVM_HOME
- VCS_HOME

■ User can exclude standard library as below:

- uvmgen -X -L <User template library path>

Here -X is used to exclude the standard library. In this case user must give their own template library path using -L.

2-64

Environment Generation (Verbose Mode)

■ Option 1: to Create Complete Environment

- Detailed questions to guide environment creation
- Typically used in beginning of project

```
Do you want to create Agents?  
Select(y/Y/n/N) [Default: n]:y  
  
Enter Master agent data  
Enter name of master agent: mas  
  
Enter name of sequencer: seq1  
  
Enter name of driver: drv1  
  
Enter name of monitor: mon1  
  
Enter name of interface: intf1  
  
Enter name of the transaction: tri  
  
Enter Slave agent data  
Enter name of slave agent: slv  
  
Enter name of sequencer: seq2
```

2-65

Directory Structure

```
-proj
|
|- README
|
|- top_env
| |
| |- doc/
| |
| |- examples/
|
|- hdl/ -> top program and environment
| |
| |- top_env_top.sv
| |- top_env.sv
|
|- env/ -> environment related files
| |
| |- slv.sv
| |- mas.sv
| |- top_env.ralf
| |- top_env_ral_env.sv
|
|- include/
|
|- src
```

```
-src/ -> Transactions, components and configurations
|
| |- slv_drv2.sv
| |- slv_mon2.sv
| |- slv_intf2.sv
| |- slv_seq2.sv
| |- mas_drv1.sv
| |- mas_tr1.sv
| |- mas_seq1.sv
| |- mas_intf1.sv
| |- mas_mon1.sv
| |- top_env_cov.sv
| |- scbd.sv
| |- top_env_cfg.sv
| |- mon_2cov.sv
| |- ral_multiplexed.sv
|
|- tests/ -> testbench module and testcases
| |
| |- top_env_tb_mod.sv
| |- top_env_test.sv
|
|- run/ ---> Makefile
```

2-66

Individual Template Generation

■ Option 2: to Generate Individual Template

- 0) Physical interface declaration
- 1) Transaction descriptor
- 2) Driver
- 3) Generic Master Agent
- 4) Generic Slave Agent
- 5) Monitor
- 6) RAL Adapter Sequence, single domain
- 7) RAL Adapter Sequence, multiplexed domains
- 8) Verification Environment
- 9) Verification Environment with Agents
- 10) Testcase
- 11) Agent
- 12) Program block
- 13) UVM sequence library
- 14) UVM Sequencer
- 15) Transaction descriptor testcase
- 16) UVM Scoreboard
- 17) Top Module
- 18) Coverage Template
- 19) Monitor To coverage Connector
- 20) Configuration descriptor

2-67

uvmgen Quick Mode

uvmgen also provides a quick mode for complete environment generation which asks minimum questions and user can pass all the options directly on the command line.

Use model:

To enable uvmgen in quick mode use the option `-q` with uvmgen:

```
% uvmgen -q <Other options>
```

The following are the various options which can be passed in quick mode (uvmgen will ask for that field if it is not specified and is required) :

- SE** Associate master and slave agents with environment [argument :y/n].
- RAL** Associate RAL environment [argument :y/n].
- ENV** Switch to provide name of environment [argument: env name].
- TR** Switch to provide name of transaction class, multiple transaction file can be generated using '+' operator. i.e. `-TR apb+atm`.
- BU** Switch to have the classes extending from a Business Unit Layer which extends from the UVM base class e.g. `-BU bu`

2-68

Quick Environment Generation

```
uvmgen -RAL y -ENV apb -TR apb_data+ahb_data -SE n -q  
or  
uvmgen -cfg_file cfile
```

```
-----  
WELCOME TO UVM TEMPLATE GENERATOR  
-----
```

```
UVM version uvm-1.0 is being used.
```

```
Using template from /.../etc/uvm_template/shared/lib/templates/uvm-1.0
```

```
Name of RAL adapter is "reg2tr"
```

```
-----  
Template generation completed.  
-----
```

```
Usage notes :
```

- 1) Find the generated files in "proj/apb" directory
- 2) Makefile has been placed in run "proj/apb/run" directory
- 3) Edit files and look for comments marked "ToDo:" and fill in the application-specific behavior for your function

2-69

Important uvmgen Options

- **-L liblist** : user-defined template directories
- **-d** : displays the standard template location
- **-o fname** : user-defined output file names
- **-O** : Overwrite the output file
- **-X (user-template-path)** : Custom templates
- **-q** : Quick mode (minimum questions)
- **-RAL [y/n]** : Enable/disable RAL in environment
- **-ENV (name)** : User environment class name
- **-TR (name)** : User transaction class name
- **-BU (name)**: BU layer class name

2-70

Agenda: Day 1

DAY

1

1 OOP Inheritance Review

2 UVM Overview



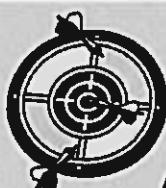
3 Modeling Transactions



4 Creating Stimulus Sequences



Unit Objectives

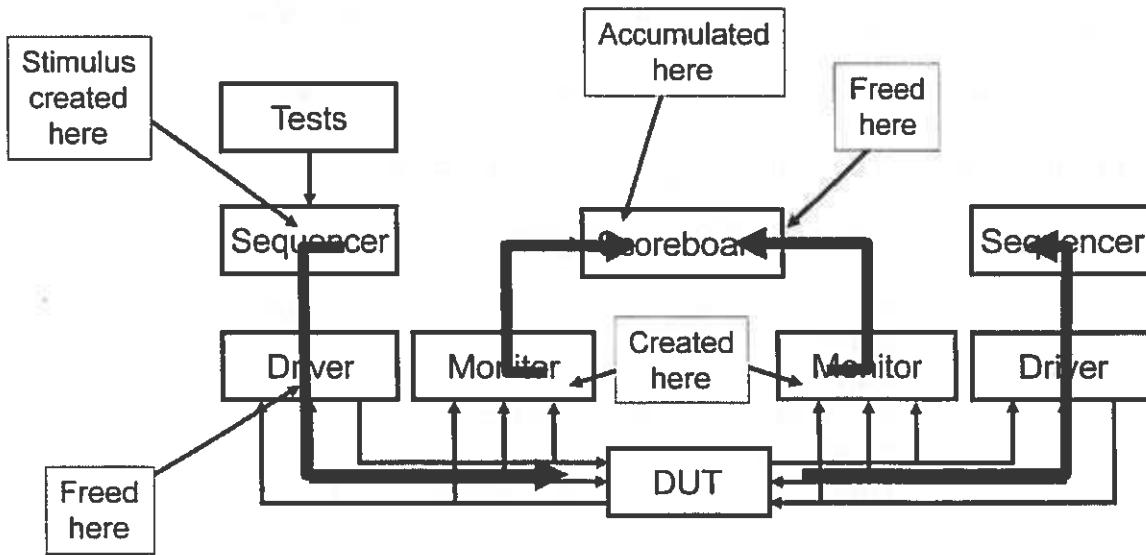


After completing this unit, you should be able to:

- Build data models by inheriting from `uvm_sequence_item`
- Use shorthand-hand macros to create `uvm_sequence_item` methods
- Configure test with modified sequence items

3-2

Transaction Flow



3-3

Modeling Transactions

- Derive from `uvm_sequence_item` base class
 - Built-in support for stimulus creation, printing, comparing, etc.
- Properties should be public 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;
    rand bit [47:0] sa, da;
    rand bit [15:0] len;
    rand bit [ 7:0] payload[$];
    rand bit [31:0] fcs;
    function new(string name = "packet");
        super.new(name);
        this.fcs.rand_mode(0);
    endfunction
endclass
```

3-4

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_t;
    rand int delay = 0;
    rand kind_t 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);
    task execute(cpu_data tr);
        repeat(tr.delay) @(sig.s.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
```

3-5

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);
  task execute(cpu_data tr);
    repeat(tr.delay) @(sig.s.drvClk);
    case (tr.kind) begin
      ...
    endcase
    if (error_condition_encountered)
      tr.status = cpu_data::ERROR;
      `uvm_info("DEBUG", tr.sprint(), UVM_HIGH)
  endtask
endclass
```

3-6

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
```

3-7

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 {
    fcs == 32'h0000_0000;
  }
endclass
```

3-8

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

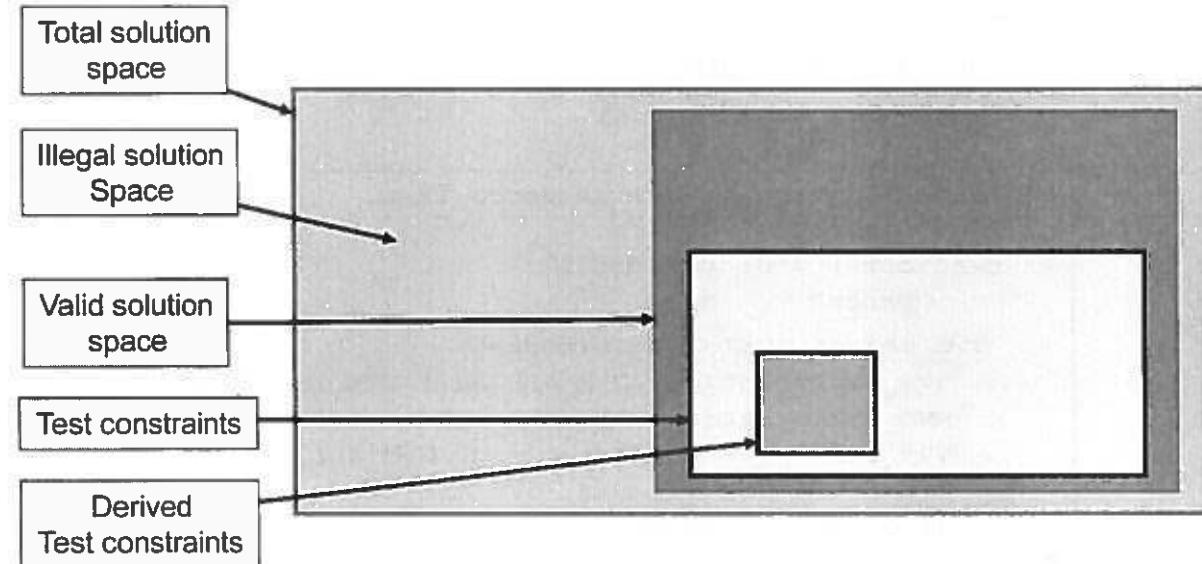
Should-obey constraints make sure errors are not injected unless you want to.

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



3-9

Transaction Class Methods

■ How create transactions manipulation method?

- Print, copy, compare, record, pack

■ Use macros to create commonly needed methods

```
'uvm_object_utils_begin(cname)
`uvm_field_*(ARG, FLAG)
`uvm_object_utils_end
```

```
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
```

3-10

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.

`uvm_field_* Field Automation Macros

- `uvm_field_* macros embed ARG properties in methods specified by FLAG
- 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
```

3-11

Examples of array macros:

```
'uvm_field_sarray_int(my_ints, UVM_ALL_ON) // Fixed array of int and bit vectors
'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 int and bit vectors
indexed with string
```

Methods Specified by FLAG

- FLAG is a collection of ON/OFF bits associated with the base class methods

```
//A=ABSTRACT Y=PHYSICAL, F=REFERENCE, S=SHALLOW, D=DEEP
//K=PACK, R=RECORD, P=PRINT, M=COMPARE, C=COPY
//----- AYFSD K R P M C
parameter UVM_DEFAULT      = 'b000010101010101;
parameter UVM_ALL_ON        = 'b000000101010101;
//Values are or'ed into a 32 bit value
parameter UVM_COPY          = (1<<0); // copy(...)
parameter UVM_NOCOPY        = (1<<1);
parameter UVM_COMPARE        = (1<<2); // compare(...)
parameter UVM_NOCOMPARE     = (1<<3);
parameter UVM_PRINT          = (1<<4); // sprint(...)
parameter UVM_NOPRINT        = (1<<5);
parameter UVM_RECORD          = (1<<6); // record(...)
parameter UVM_NORECORD       = (1<<7);
parameter UVM_PACK           = (1<<8); // pack(...)
parameter UVM_NOPACK         = (1<<9);

// Disable printing for a hidden field
`uvm_field_int(hidden_field, UVM_ALL_ON | UVM_NOPRINT)
```

3-12

The flag bits are or'ed together to control the action of the copy, compare, etc methods.

If you want to turn off an action for a field, add NO to the flag name.

```
class my_object extends uvm_object;
  int hidden_field; // field not to be printed
  `uvm_object_utils_begin(my_object)
    // The hidden_field variable is not printed by the print() and sprint() methods.
    `uvm_field_int(hidden_field, UVM_ALL_ON | UVM_NOPRINT)
  `uvm_object_utils_end
endclass
```

UVM_COPY	- Field will participate in <uvm_object::copy>
UVM_COMPARE	- Field will participate in <uvm_object::compare>
UVM_PRINT	- Field will participate in <uvm_object::print>
UVM_RECORD	- Field will participate in <uvm_object::record>
UVM_PACK	- Field will participate in <uvm_object::pack>
UVM_DEEP	- Object field will be deep copied
UVM_SHALLOW	- Object field will be shallow copied
UVM_REFERENCE	- Object field will be copied by reference

PHYSICAL – User defined policies

ABSTRACT – User defined policies

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 the ~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)
```

3-13

Transaction Methods

■ Methods supported by macros

For debugging

```
virtual function void print(uvm_printer printer = null);
virtual function string sprint(uvm_printer printer = null)
```

For making copies

```
virtual function uvm_object clone();
virtual function void copy(uvm_object rhs);
```

Used by checker

```
virtual function bit compare(uvm_object rhs,
                           uvm_comparer comparer = null);
```

Used by transactors

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

Use to record
transactions for
debugger

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

■ User must NOT override these methods

- Exception is clone() which has a default implementation but can be overridden if required

3-14

Creating these method by hand is tedious and error prone. For example, if you add a new variable to a class, you just need to declare it, and then use the uvm_field macros to add it to all the methods. If you did not use the macros, you have to manually process the fields with do_xxx() methods as shown in slide 3-16 - a good way to create bugs.

Method Descriptions

methods	function
clone	The clone method creates and returns an exact copy of this object. Calls create followed by copy
copy	Returns a deep copy of this object
print	Deep-prints this object's properties in a format and manner governed by the given <i>printer</i> argument
sprint	Same as print, but returns a string
compare	Compare method deep compares this data object with the object provided in the rhs
pack/unpack	Bitwise-concatenate object's properties into an array of bits, bytes, or ints.
record	Deep-records this object's properties according to an optional <i>recorder</i> policy

3-15

Customization of Methods

- Methods can be customized via `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);
```

3-16

```
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
```

For examples of customization of methods, take a look at the `$VCS_HOME/doc/examples/sv/uvm` directory.

Using Transaction Methods

```
packet pkt0, pkt1, pkt2;
bit bit_stream[];
pkt0 = packet::type_id::create("pkt0", this);
pkt1 = packet::type_id::create("pkt1", this);
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_fatal("MISMATCH", {"\n", pkt0.sprint(), pkt2.sprint()});
end                                         // sprint() returns string for logging
pkt0.pack(bit_stream);                     // pack content of pkt0 into bit_stream array
pkt2.unpack(bit_stream);                   // unpack bit_stream array into pkt2 object
```

3-17

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

Modify Constraint in Transactions by Type

```
class packet extends uvm_sequence_item;
    rand bit[3:0] sa, da;
    rand bit[7:0] payload[];
    constraint valid {payload.size inside {[1:10]};}
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

class test_da_3_type extends test_base;
    `uvm_component_utils(test_da_3_type)
    function new(string name, uvm_component parent);
        super.new(name, parent);
    endfunction
    virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase);
    //set_type_override("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 adding constraint

Preferred override (compile-time check)

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

3-18

```
// 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()` method uses string as arguments resulting in type check done at run-time.

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

Transaction Replacement Results

Simulate with:

```
simv +UVM_TESTNAME=test_da_3_type
```

```
UVM_INFO @ 0: uvm_test_top.env.i_agent.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:
factory.print()



See note

3-19

In the uvm-1.1 implementation of **uvm_pkg**, one has direct access to the **uvm_factory** singleton object via the **factory** handle.

Moving forward, this access will be deprecated.

The proper way to access the **uvm_factory** singleton object will be:

```
uvm_factory my_factory=uvm_factory::get(); // my_factory is a name of your choice
```

And, the call to print the content of the **uvm_factory** singleton object will then be:

```
my_factory.print();
```

In the future UVM source code release, if you use the factory handle directly you will see something like the following:

UVM_WARNING: reporter [UVM/PKG/DEPR_FACTORY_VAR] the **uvm_pkg::factory** variable will be deprecated in a future revision of the UVM Library, due to concerns with stability at initialization time, as well as the fact that it was not properly name-protected. More information can be found at: <http://www.eda.org/mantis/view.php?id=4379>. Please update your code to use **uvm_factory::get()** instead.

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("env.i_agent.seqr.*", "packet", "packet_da_3");
  // set_inst_override_by_type("env.i_agent.seqr.*", 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_agent.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_agent.seqr.*	packet_da_3

3-20

Parameterized Transaction Class

- Parameterized transaction requires a different macro

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

```
class my_data #(width=48) extends uvm_sequence_item;
  rand bit [width-1:0] data;
  `uvm_object_param_utils_begin(my_data #(width))
    `uvm_field_int(data, UVM_ALL_ON)
  `uvm_object_utils_end
endclass
```

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

```
class my_component extends uvm_component;
  task run_phase(uvm_phase phase);
    my_data d = my_data#():type_id::create("d", this);
  endtask
endclass
```

3-21

You should use typedef to create the specific type of use for the class.

```
class my_component extends uvm_component;
  typedef my_data#() my_data_type;
  `uvm_object_utils(my_component)
  task run_phase(uvm_phase phase);
    my_data_type d = my_data_type::type_id::create("d", this);
  endtask
endclass
```

Unit Objectives Review

Having completed this unit, you should be able to:

- **Build data models by inheriting from `uvm_sequence_item`**
- **Use shorthand-hand macros to create `uvm_sequence_item` methods**
- **Configure test with modified sequence items**

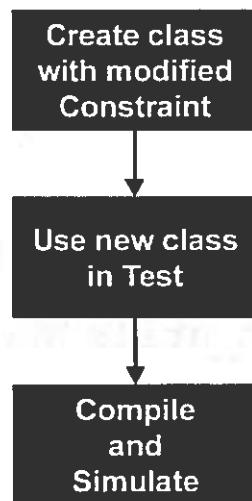
3-22

Lab 2 Introduction

Implement configurable sequence



15 min



3-23

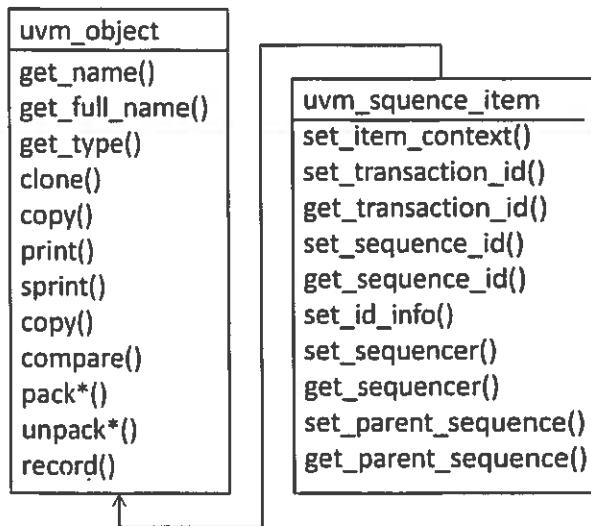
Appendix

uvm_sequence_item Class Tree

uvm_object_utils Macro

uvm_sequence_item Class Tree

uvm_sequence_item Class Tree



3-26

uvm_object_utils Macro

uvm_object_utils Macro

`uvm_object_utils[_begin] (cname)

- Implements the following:

```
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

Agenda: Day 1

DAY

1

1 OOP Inheritance Review

2 UVM Overview



3 Modeling Transactions



4 Creating Stimulus Sequences



Synopsys 40-I-054-SSG-004

© 2013 Synopsys, Inc. All Rights Reserved

4-1

Unit Objectives



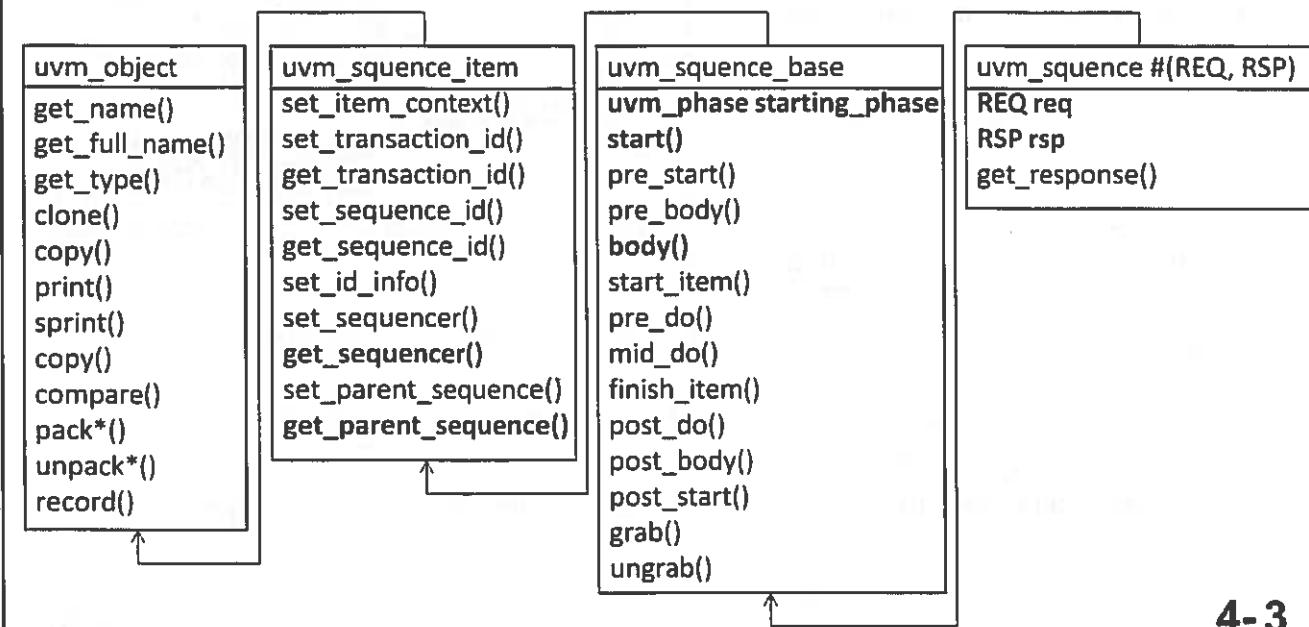
After completing this unit, you should be able to:

- Create sequences using base classes
- Create complex sequences
- Configure sequences
- Setting sequences to execute at different phases

4-2

uvm_sequence Class Common Members

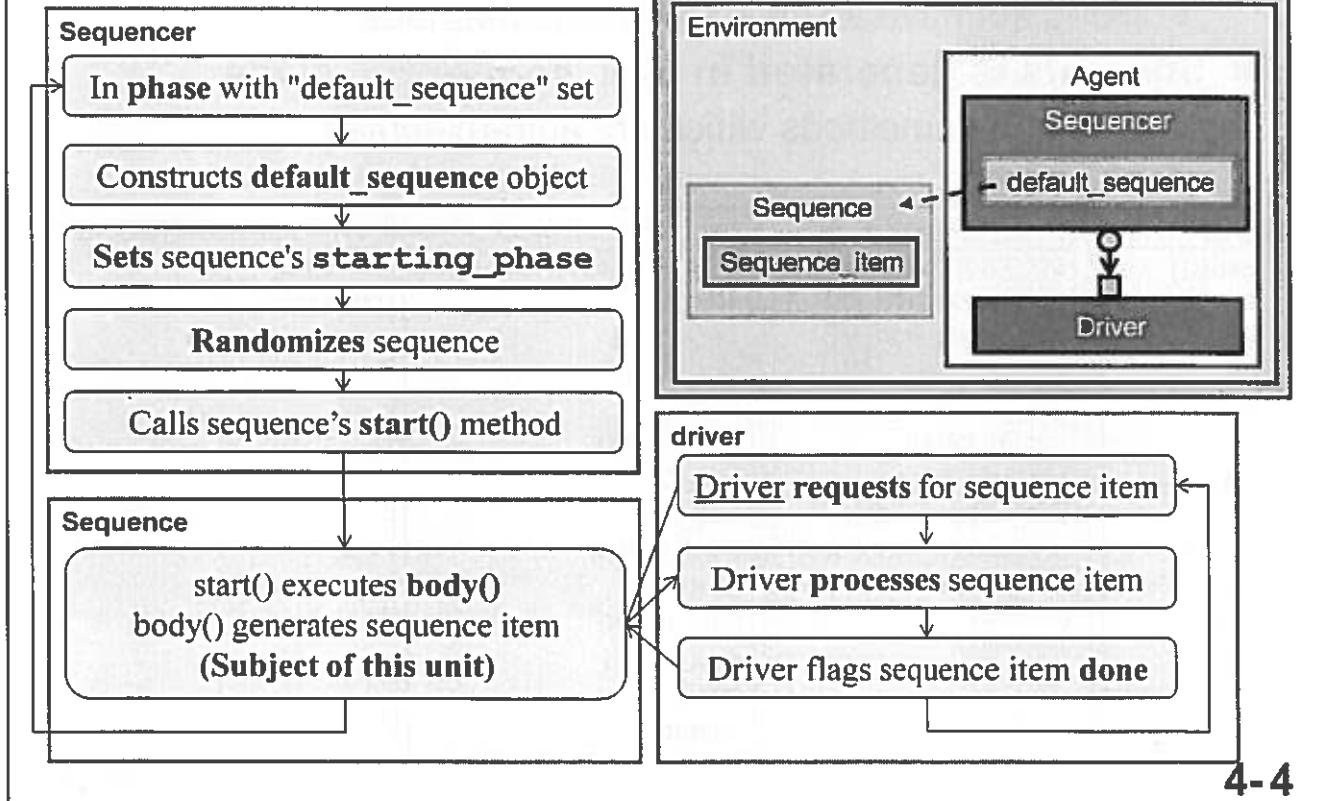
- Transaction base class is `uvm_sequence_item`
 - Does not have any auto-executed methods
- Stimulus is generated in `uvm_sequence` class
 - Does have methods which are auto-executed



4-3

Implicit Sequence Execution Protocol

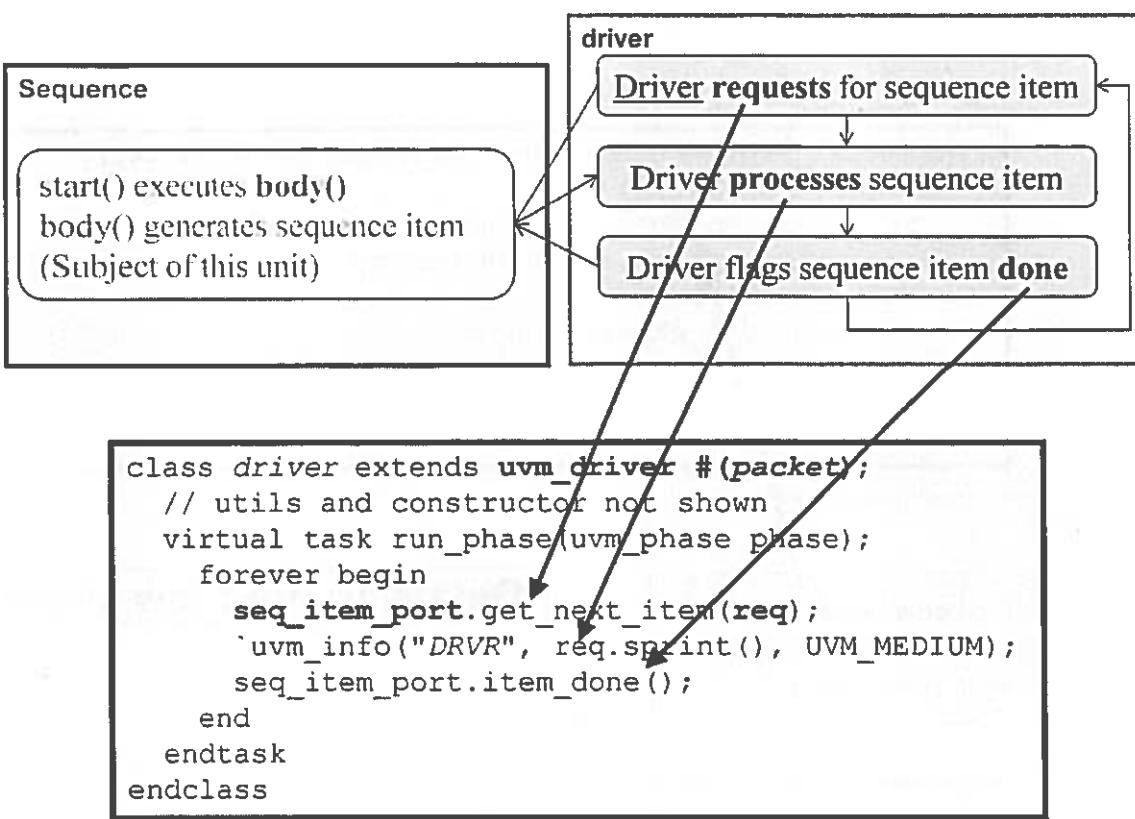
■ In the configured phase:



starting_phase is the phase in which the sequencer runs the configured sequence by calling the **start()** method of the sequence. It is a handle to a **uvm_phase** object.

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

Mapping Protocol to Code (1/2)



4-5

Mapping Protocol to Code (2/2)

```
class user_sequence extends uvm_sequence #(packet);
    virtual task body();
        if (starting_phase != null)
            starting_phase.raise_objection(this, "starting");
        `uvm_do(req);
        if (starting_phase != null)
            starting_phase.drop_objection(this, "done");
    endtask
endclass
```

Sequence

start() executes body()
body() generate sequence item
(Subject of this unit)

Details in next few slides

4-6

Sequence Class Requirements

- User sequence extends from `uvm_sequence` class
- Sequence code resides in `body()` method
 - `'uvm_do` creates, randomizes and passes transaction to driver
- Sequence need to raise and drop phase objection to allow sequence to span execution over time
 - `starting_phase` set by sequencer (see slide 4-4)

```
class user_sequence extends uvm_sequence #(packet);  
  // macro and constructor not shown  
  virtual task body();  
    if (starting_phase != null)  
      starting_phase.raise_objection(this, "starting");  
      `uvm_do(req);  
    if (starting_phase != null)  
      starting_phase.drop_objection(this, "done");  
  endtask  
endclass
```

Optional debug message

Object raising objection

4-7

A sequence consists of one or more sequence items. Each sequence item for the sequence is created in the `body()` task, usually with pre-defined UVM macros such as `'uvm_do()`. This macro randomizes the sequence item and makes it available for driver to access.

The `uvm_sequence` class has a handle `req` that is typed to the sequence item parameter. You should use this built-in handle as the sequence item reference.

The `raise_objection()` / `drop_objection()` calls are to prevent the Run Time phase that the sequence is executed in from terminating before the sequence completes. Without these calls, if a Run Time phase consumes time and objection was not raised, then that phase will execute but terminates in 0 time. When the phase terminates, all child thread of that phase will be killed.

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 answered (objections dropped).

Reminder: there is an optional second argument to the objection method:

```
starting_phase.raise_objection(this, $sformatf("Starting %s", this.get_name()));
```

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

```
UVM_INFO @ 0.0s: main [OBJTN_TRC] Object uvm_test_top.env.sequencer.simple_sequence  
raised 1 objection(s) (Starting user_sequence): count=1 total=1
```

Sequence Class Callbacks

■ Sequence method flow when start() is executed:

```
start() {  
    pre_start()  (task) UVM 1.1  
    pre_body()   (task) if call_pre_post==1 for sub-sequences  
    body()       (task) YOUR STIMULUS CODE  
    post_body()  (task) if call_pre_post==1 for sub-sequences  
    post_start() (task) UVM 1.1  
}
```



See note

```
class user_sequence extends uvm_sequence #(packet);  
    virtual task start();  
    virtual task pre_start();*  
    virtual task pre_body();  
    virtual task body();  
    virtual task post_body();  
    virtual task post_start();*  
endclass
```

4-8

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. This nested sequence's pre_body() and post_body() will not be executed. Also, because it is a child sequence, it will have no access to starting_phase. This is why you must always check for the existence of the starting_phase before raising or dropping objection.

For forward compatibility, you should NOT use pre_body() and post_body() for raising and dropping of objection. The reason is that in uvm-1.1 onwards, there is a new set of task before and after the pre_ and post_body(). If you drop objection in the post_body() method, you may cause the phase to terminate too early if there is an implementation of the post_start() method that requires simulation time.

CAUTION:

In uvm-1.2, there is a possibility that the execution order of pre_start/pre_body and post_body/post_start may change.

Recommendation:

Treat pre_body and post_body as deprecated.

User Sequence with Callback

Simplify sequence implementation:

- Create base sequence to raise and drop objections
- Create user sequence extending from base sequence

```
class base_sequence extends uvm_sequence #(packet);
  // macro and constructor not shown
  virtual task pre_start();
    if (get_parent_sequence() == null && starting_phase != null)
      starting_phase.raise_objection(this);
  endtask
  virtual task post_start(); Minimizes excessive objections
    if (get_parent_sequence() == null && starting_phase != null)
      starting_phase.drop_objection(this);
  endtask
endclass

class user_sequence extends base_sequence; // macro and constructor
  virtual task body();
    `uvm_do(req);
    // if constraint is required: `uvm_do_with(req, {da == 3});
  endtask
endclass
```

4-9

Caution:

In UVM 1.0, `pre_start()` and `post_start()` methods do not exist.

When `seq.start()` executes in UVM 1.1, the `pre_start()` will be executed first, followed by `body()` then `post_start()`. Both `pre_start()` and `post_start()` are noop tasks if user code is not implemented.

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);
```

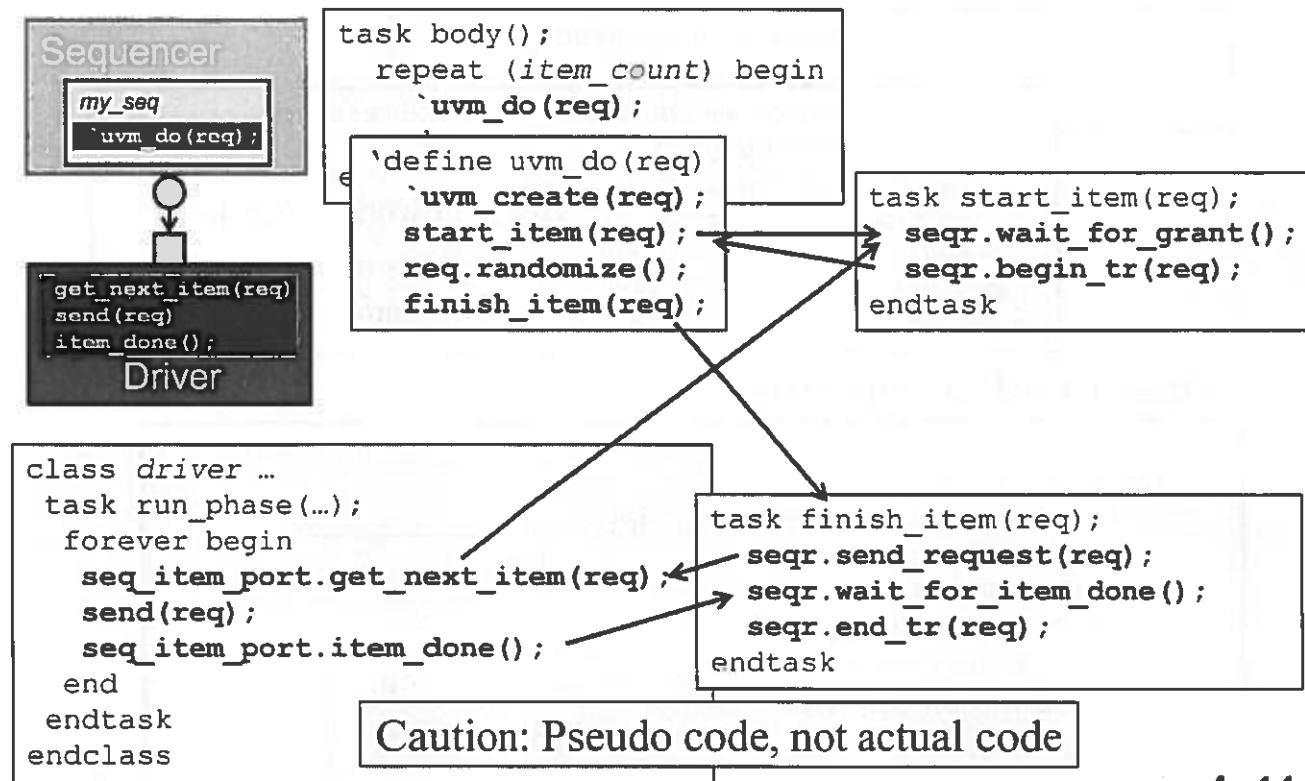
- User can manually execute the embedded methods:

```
virtual task body();
// Instead of `uvm_do_with(req, {sa == 3;})
// User can populate own item:
`uvm_create(req); // constructs item linked with sequencer
start_item(req); // wait for parent sequencer to get request from driver
req.sa = 3; // populate item with user value
...
finish_item(req); // use parent sequencer to pass item to driver and wait for done
endtask
```

4-10

`uvm_do Macro Interaction Detailed

■ Pseudo code illustration:



4-11

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

- 1 – The sequencer executes `body()` method of the sequence
- 2 – The sequence creates the transaction, then waits for its parent sequencer to grant permission to start the transaction.
- 3 – The granting process starts when driver calls `get_next_item(req)` on the TLM port connected to the sequencer (`seq_item_port`).
- 4 – The sequencer responds to the driver request by granting the sequence permission to continue.
- 5 – The sequence, then, marks the transaction as started (time stamped).
- 6 – The sequence randomizes the transaction next 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 responds by marking the transaction as ended (ending time stamp) and continues on to the next iteration within `body()`.

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 t_sequence extends uvm_sequence#(transaction);
  virtual task body();
    repeat(10) begin
      `uvm_do(req);
    end
  endtask
endclass
```

Does not work for randc!
randc requires randomization
on the same object

- User need to implement :

```
virtual task body();
  `uvm_create(req);
  repeat(10) begin packet req_copy;
    start_item(req);
    req.randomize(); —————— Randomize same object
    `uvm_create(req_copy);
    req_copy.copy(req);
    finish_item(req_copy); —————— Pass a copy of the
  end
  endtask
```

object to driver

4-12

If you like macro, you can create your own macro:
(Only for sequence_item classes. Additional code needed for sequence classes)

```
`define my_do_randc(req, class_name) \
begin \
  class_name req_copy; \
  start_item(req); \
  if(!req.randomize()) \
    `uvm_fatal("RAND_ERR", req.sprint()); \
  `uvm_create(req_copy); \
  req_copy.copy(req); \
  finish_item(req_copy); \
end
```

Then, the sequence code becomes:

```
virtual task body();
  `uvm_create(req); // constructs item object once
  repeat(10) begin
    `my_do_randc(req, packet);
  end
endtask
```

User Can Implement Scenario Sequence

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

Use array of rand sequence items to create scenario

4-13

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()** task contains steps similar to **uvm_do**, but skips the randomize step. Instead **body()** takes the prebuilt array of random packets and sends them to the TLM port.

Nested Sequences

■ Sequences can be nested

```
class noise_sequence extends base_sequence; // other code
virtual task body();  
  
class burst_sequence extends base_sequence; // other code
virtual task body();  
  
class congested_sequence extends base_sequence; // other code
virtual task body();  
  
class nested_sequence extends base_sequence;
// 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
```

'uvm_do macro also applies to sequences

4-14

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 1.1, 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(m_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 execute
`uvm_do(burst); // no pre/post_body execution
`uvm_do(congestion); // no pre/post_body execution
endtask
```

call_pre_post defaults to 1

Executing User Sequences in Test

- Default test (*test_base*) executes default sequence specified by embedded environment

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

- Test writer can override or turn this off in tests

```
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.*.seqr.main_phase",
            "default_sequence", null); // tells seqr not to execute sequence in main phase
    endfunction
endclass
```

4-15

Implicit Sequence Execution at a Phase

■ Sequences be targeted for a chosen phase

- Typically done at the testcase level

```
class simple_seq_RST extends uvm_sequence #(...);  
class simple_seq_CFG extends uvm_sequence #(...);  
class simple_seq_MAIN extends uvm_sequence #(...);  
  
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.seqr.reset_phase",  
      "default_sequence", simple_seq_RST::get_type());  
    seq_phase::set(this, "env.seqr.configure_phase",  
      "default_sequence", simple_seq_CFG::get_type());  
    seq_phase::set(this, "env.seqr.main_phase",  
      "default_sequence", simple_seq_MAIN::get_type());  
  endfunction  
endclass
```

4-16

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.

The explicit sequence execution is shown on the next page.

Explicit Sequence Execution

■ Execute sequence via sequence's start() method:

```
class router_env extends uvm_env; // other code not shown
  function void build_phase(...); super.build_phase(...);
    uvm_config_db #(uvm_object_wrapper)::set(this, ".seqr.main_phase",
      "default_sequence", packet_sequence::get_type());
  endfunction
endclass
```

```
class test_explicit_sequence extends test_base; //other code not shown
  function void build_phase(uvm_phase phase); super.build_phase(...);
    uvm_config_db#(uvm_object_wrapper)::set(this, ".seqr.main_phase",
      "default_sequence", null);
  endfunction
  task main_phase(uvm_phase phase);
    nested_sequence seq = new();
    phase.raise_objection(this);
    if (!seq.randomize()) `uvm_fatal(...);
    seq.start(env.i_agent.seqr);
    phase.drop_objection(this);
  endtask
endclass
```

Turned off implicit sequence

Objection must be raised
in phase method

Must set executing sequencer

4-17

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.i_agent.seqr.packet_sequence",
            "item_count", 20);
    endfunction
endclass
```

Field tag within sequence Value Name path to sequence

■ Reference configuration field through get_full_name()

```
class packet_sequence extends base_sequence; // other code not shown
    int item_count = 10;
    function new(string name = "packet_sequence");
        virtual task pre_start(); super.pre_start();
            uvm_config_db#(int)::get(null, this.get_full_name(),
                "item_count", item_count);
        endtask
        virtual task body();
            repeat(item_count) begin ...; end
        endtask
    endclass
```

Full path to sequence

Field tag to retrieve value

Variable to store value unchanged if not set

4-18

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.

Instance-Based Configuration Benefits

■ Prevents field name collision in nested sequences

```
class packet_sequence ...; // many code left off
    int item_count;
    uvm_config_db#(int)::get(null, this.get_full_name(), "item_count", item_count);
endclass
class other_sequence ...;
    int item_count;
    uvm_config_db#(int)::get(null, this.get_full_name(), "item_count", item_count);
endclass
class nested_sequence ...;
    packet_sequence p_seq; other_sequence o_seq;
    virtual task body();
        `uvm_do(p_seq);
        `uvm_do(o_seq);
    endtask
endclass
class test_nested_seq extends test_base; // other code not shown
    virtual function void start_of_simulation_phase(...); ...
        uvm_config_db#(int)::set(this, "env.i_agent.seqr.nested_sequence.p_seq",
                               "item_count", 20);
        uvm_config_db#(int)::set(this, "env.i_agent.seqr.nested_sequence.o_seq",
                               "item_count", 50);
    endfunction
endclass
```

4-19

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.i_agent.seqr",
      "item_count", 20);
  endfunction
endclass
```

Name path to sequencer

■ Reference configuration field through get_sequencer()

```
class packet_sequence extends base_sequence; // macros not shown
  int item_count = 10;
  function new(string name = "packet_sequence"); // code not shown
    virtual task body();
      uvm_config_db#(int)::get(this.get_sequencer(), "",
        "item_count", item_count);
      repeat(item_count) begin ...; end
    endtask
  endclass
```

Full path to sequencer

4-20

Sequencer-Based Configuration Benefits

- Provides common configuration for all in nested sequences

```
class packet_sequence ...; // many code left off
    bit[15:0] da_enable = '1;
    uvm_config_db#(int)::get(this.get_sequencer(), "", "da_enable", da_enable);
endclass

class other_sequence ...;
    bit[15:0] da_enable = '1;
    uvm_config_db#(int)::get(this.get_sequencer(), "", "da_enable", da_enable);
endclass

class nested_sequence ...;
    packet_sequence p_seq; other_sequence o_seq;
    virtual task body();
        `uvm_do(p_seq);
        `uvm_do(o_seq);
    endtask
endclass

class test_nested_seq extends test_base; // other code not shown
    virtual function void build_phase(...); super.build_phase(...);
        uvm_config_db#(int)::set(this, "env.i_agent.seqr", "da_enable", 'h0008);
    endfunction
endclass
```

4-21

Configuring Sequences (Agent-Based)

- Provides access to agent configuration for sequences
 - Only to be used to retrieve agent configuration
- Set in test

```
class test_agent_configuration extends test_base;  
  // utils and constructor not shown  
  virtual function void build_phase(...); ...  
    uvm_config_db#(int)::set(this, "env.i_agent", "port_id", 3);  
  endfunction  
endclass
```

Agent configuration field

Name path to agent

- Retrieve agent configuration field through sequencer

```
class packet_sequence extends base_sequence; // other code not shown  
  int port_id;  
  virtual task body();  
    uvm_sequencer_base my_seqr = get_sequencer();  
    uvm_config_db#(int)::get(my_seqr.get_parent(), "", "port_id", port_id);  
  ...  
  endtask  
endclass
```

Full path to agent

Agent configuration field

4-22

Unit Objectives Review

Having completed this unit, you should be able to:

- Create sequences using base classes
- Create complex sequences
- Configure sequences
- Setting sequences to execute at different phases

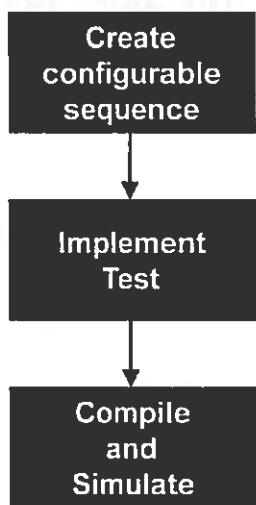
4-23

Lab 3 Introduction

Implement configurable sequence



30 min



4-24

Appendix

Sequence Priority/Weight

Sequencer-Driver Response Port

Out-Of-Order Sequencer-Driver Port

Sequence Priority/Weight

Sequence Priority/Weight

■ Sequences can be assigned priority/weight

```
class nested_sequence extends base_sequence;
    // utils macro and other code not shown
    virtual task body();
        uvm_sequencer_base seqr = get_sequencer();
        seqr.set_arbitration(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 SEQ_ARB_FIFO)

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

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()

4-27

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

When the sequencer arbitration mode is set to 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 SEQ_ARB_FIFO, which returns the entry at avail_sequences[0].

Sequencer-Driver Response Port

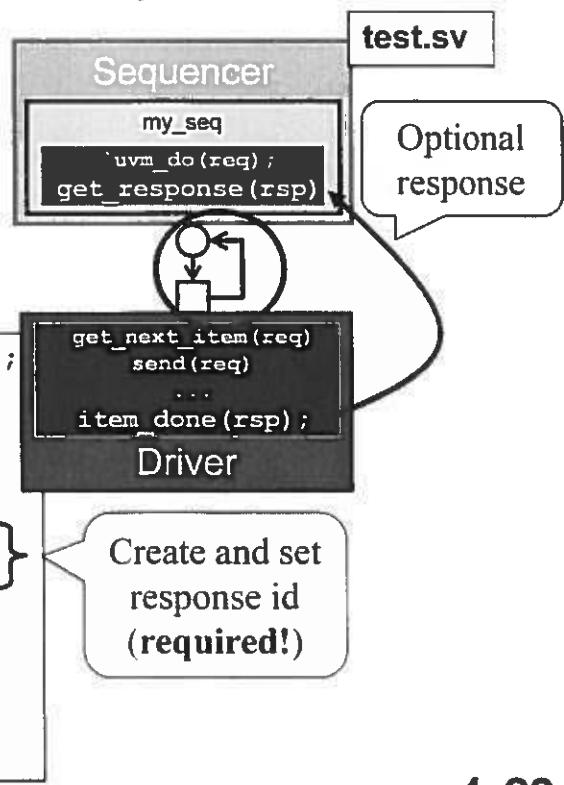
Sequencer-Driver Response Port

- Driver can send response back to sequence

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

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");
    rsp.set_id_info(req);
    // set rsp response
    seq_item_port.item_done(rsp);
  end
endtask
endclass
```

Set response



4-29

Note that the driver receives **req** but sends back **rsp**.

Out-Of-Order Sequencer-Driver Port

Sequence

```
task body();
    packet pkt_in_drv[int];  int packet_transaction_id = 0;
    repeat(item_count) begin
        wait(pkt_in_drv.size() < 5); <-- throttle traffic
        `uvm_create(req);
        req.set_transaction_id(packet_transaction_id++);
        start_item(req);
        if (!req.randomize()) ...;
        pkt_in_drv[req.get_transaction_id()] = req;
        fork
            packet in_driver = req;
            begin
                packet from_driver;
                get_response(from_driver, in_driver.get_transaction_id());
                // process response
                pkt_in_drv.delete(from_driver.get_transaction_id());
            end
            join_none
            finish_item(req);
        end
        wait(pkt_in_drv.size() == 0); <-- Block until array is empty
    endtask: body
```

unique id number for each transaction

throttle traffic

Stores what the driver is operating on

retrieve response based on transaction_id

Remove response from in-operation array

4-31

Out-Of-Order Driver

```
virtual task run_phase(uvm_phase phase);
  packet pkt_q[$];
  fork
    forever begin
      packet in_use; int index;
      wait (pkt_q.size() != 0);
      index = $urandom_range(0, pkt_q.size()-1);
      in_use = pkt_q[index];
      begin_tr(in_use); // transaction recording
      // process transaction
      pkt_q.delete(index);
      seq_item_port.put_response(in_use);
      end_tr(in_use); // transaction recording
    end
    join_none
    forever begin
      seq_item_port.get_next_item(req);
      accept_tr(req); // transaction recording
      pkt_q.push_back(req);
      seq_item_port.item_done();
    end
  endtask: run_phase
```

Wait for transaction to process

Indicate which transaction has completed processing

Indicate transaction was picked up

Create thread to process transaction

Get transaction

Store transaction in queue

■ Requires
+define+UVM_DISABLE_AUTO_ITEM_RECORDING

Agenda: Day 2

**DAY
2**

5 Component Configuration & Factory

6 Component Communication

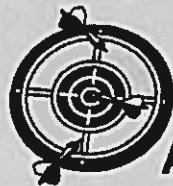


7 Scoreboard & Coverage

8 UVM Callback



Unit Objectives



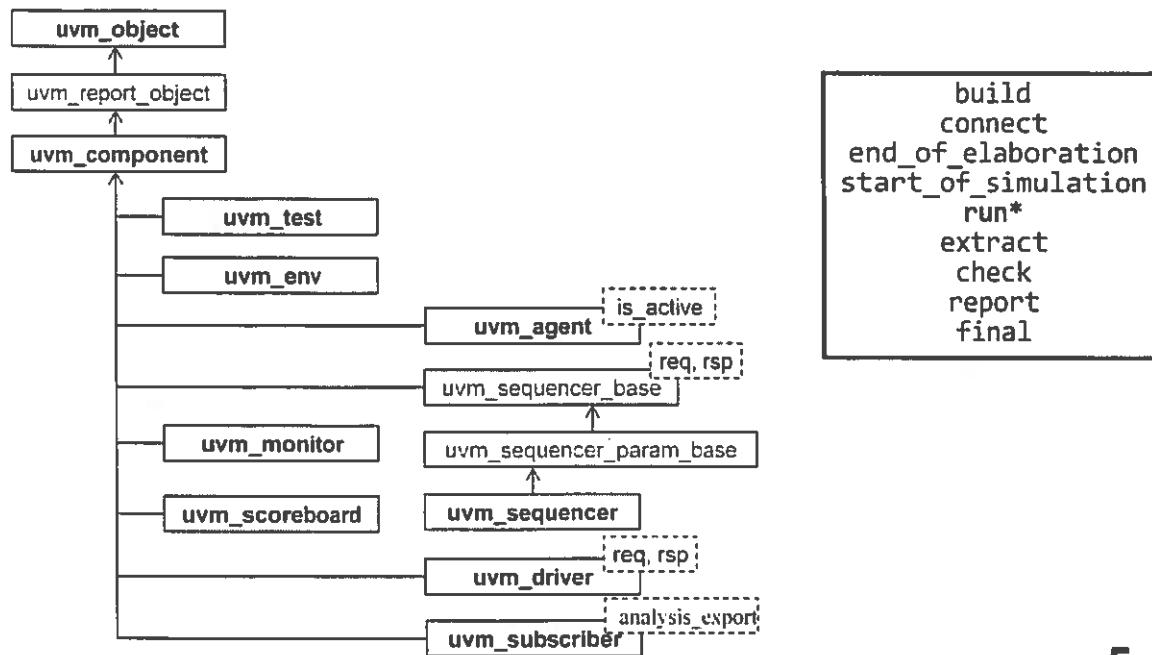
After completing this unit, you should be able to:

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

5-2

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 seqr; driver drv;

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

...
agent agnt = agent::type_id::create("agnt", 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_top` 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

Display all members

```
agt.print();
```

Get logical name

```
string str = drv.get_name();
```

Get hierarchical name

```
string str = drv.get_full_name();
```

Find one component via logical name

```
uvm_component comp;  
comp = uvm_top.find("*.seqr");
```

Logical name

Can use wildcard
* – match anything (including ".")
+ – one or more character (including ".")
? – match one character

Find all matching components

```
uvm_component comps[$];  
uvm_top.find_all("*.drv?", comps);  
foreach (comps[i]) begin  
    comps[i].print();  
end
```

5-5

Here are a few simple examples of regular expression :

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

A popular substitute 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

top\..*	top.*
top\env\..*\.\monitor	top.env.*.\monitor
\.*\.\monitor	.*.\monitor

The resource facility 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 obj;  
obj = this.get_parent();
```

Finding object via logical name

```
uvm_component obj;  
obj = vip.get_child("seqr");
```

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

`uvm_config_db#(type)::set(context, inst_name, field, value)`

Object context in which the target reside
Hierarchical instance name in context
Value to set
Tag to set value

`uvm_config_db#(type)::get(context, inst_name, field, var)`

Object context in which the target resides
Hierarchical instance name in context
Variable to store value unchanged if not set
Tag to get value

5-7

`uvm_config_db#()::get()` method returns a 1 if when 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

- Should target agent
 - ◆ Agent then configure 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_agent[10]", "port_id", 10);  
  endfunction  
  
class i_agent extends uvm_agent;  
  // constructor not shown  
  int port_id = -1; // default value  
  `uvm_component_utils_begin(i_agent)  
    `uvm_field_int(port_id, UVM_DEFAULT)  
  `uvm_component_utils_end  
  virtual task build_phase(...); ...  
    uvm_config_db #(int)::get(this, "", "port_id", port_id);  
    uvm_config_db #(int)::set(this, "*", "port_id", port_id);  
  endtask  
endclass
```

Component configuration
is done in build_phase

Set port_id value in
configuration database for agent

Agent retrieves configuration

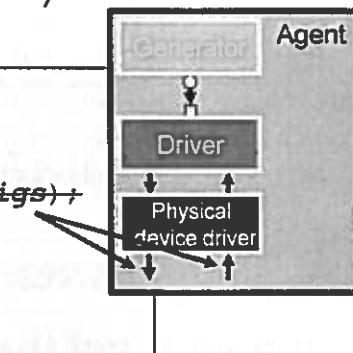
Then, set child component configuration

5-8

Other Examples: Physical Interface

- For physical layer connection, don't pass virtual interface via constructor argument
 - Not supported by proxy class (`type_id`) `create()` method

```
class driver extends uvm_driver#(packet);
    virtual router_tb_if.driver sigs;
    ...
    function new(..., virtual router_tb_if.sigs);
        this.sigs = sigs;
    ...
endfunction
endclass
```



- Use set/get methods in `uvm_config_db`

5-9

Configuring Component's DUT Interface

■ In component, use `uvm_config_db::get()`

```
class driver extends ...; virtual router_tb_io sigs; // simplified code
function void build_phase(uvm_phase phase); // other code not shown
  if (!uvm_config_db#(virtual router_tb_io)::get(this, "", "sigs", sigs))
    `uvm_fatal("CFGERR", "Driver DUT interface not set");
endfunction
endclass
```

■ In test, use `uvm_config_db::set()` to target agent

```
class test_base extends ...; // simplified code
function void build_phase(uvm_phase phase); // other code not shown
  uvm_config_db#(virtual router_tb_io)::set(this, "env.i_agent[*]",
                                             "sigs", router_test_top.sigs);
endfunction
endclass
```

Access signal via XMR, macro or port listed interfaces

■ In agent, get then set child components' interface

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

5-10

Dynamic Control Variable Example

■ Component/sequence dynamic variable control

```
class network_sequence extends uvm_sequence #(network_transaction);
    // constructor and object_utils not shown
    int SNR = 100; // user configurable
    virtual task body();
        repeat(item_count) begin
            uvm_config_db #(int)::get(get_sequencer(), "", "SNR", SNR);
            `uvm_info("SNR", $sformatf("SNR is: %0d\t", SNR), UVM_MEDIUM);
            `uvm_do_with(req, {this.SNR == local::SNR});
        end
    endtask
endclass
```

Get stimulus configuration
when needed

```
class test_noise extends test_base;
    // utils macro and constructor not shown
    virtual task run_phase(uvm_phase phase);
        #1ms;
        uvm_config_db #(int)::set(this, "env.i_agent[5].seqr", "SNR", 60);
        #10ms;
        uvm_config_db #(int)::set(this, "env.i_agent[5].seqr", "SNR", 10);
    endtask
endclass
```

Set stimulus configuration
when needed

5-11

Global UVM Resource

- **uvm_config_db** sets object's local configurations
- For global configuration, use UVM resource

```
uvm_resource_db#(d_type)::set("scope", "key", value, accessor);
```

Scope where resource reside

Content of resource

Data type

Name of resource

Object making call
(for debugging)

- Retrieval of the resources can be done in two ways

- Read by name
- Read by type

Variable of data type

```
uvm_resource_db#(d_type)::read_by_name("scope", "key", type_var, accessor);  
uvm_resource_db#(d_type)::read_by_type("scope", type_var, accessor);
```

Variable of data type

5-12

Dynamic Control Variable Example (Global)

■ Component/sequence dynamic variable control

```
class network_sequence extends uvm_sequence #(network_transaction);
  // constructor and object_utils not shown
  int SNR = 100; // user configurable
  virtual task body();
    repeat(item_count) begin
      uvm_resource_db #(int)::read_by_type("SNR", SNR, this);
      `uvm_info("SNR", $sformatf("SNR is: %0d\\%", SNR), UVM_MEDIUM);
      `uvm_do_with(req, {this.SNR == local::SNR});
    end
  endtask
endclass
```

Read from global resource

```
class test_noise extends test_base;
  // utils macro and constructor not shown
  virtual task run_phase(uvm_phase phase);
    #1ms;
    uvm_resource_db #(int)::set("SNR", "", 60, this);
    #10ms;
    uvm_resource_db #(int)::set("SNR", "", 10, this);
  endtask
endclass
```

Set SNR values at
simulation time
where values
should be applied

5-13

Additional Needs: Manage Test Variations

- Tests need to introduce class variations, e.g.
 - Adding constraints
 - Modify the way data is sent by the driver
- Instance based variation or global
- Control object allocation in entirety or for specific objects
- 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?

Type of object determines memory allocated for the instance

```
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
```

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

Impossible to add new members or modify constraint later

```
class my_packet extends packet; ...;  
int serial_no;  
virtual function void set_serial_no(...);  
virtual function int get_serial_no();  
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 scoreboard. 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 seqr;
    driver drv;
    ...
    virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        seqr = new("seqr", 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.sigs.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(...);

Proxy class can create objects specified in test with overrides

5-17

Transaction Factory

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

Required!

Macro defines a proxy class called `type_id`

An instance of proxy class is registered in `uvm_factory`

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

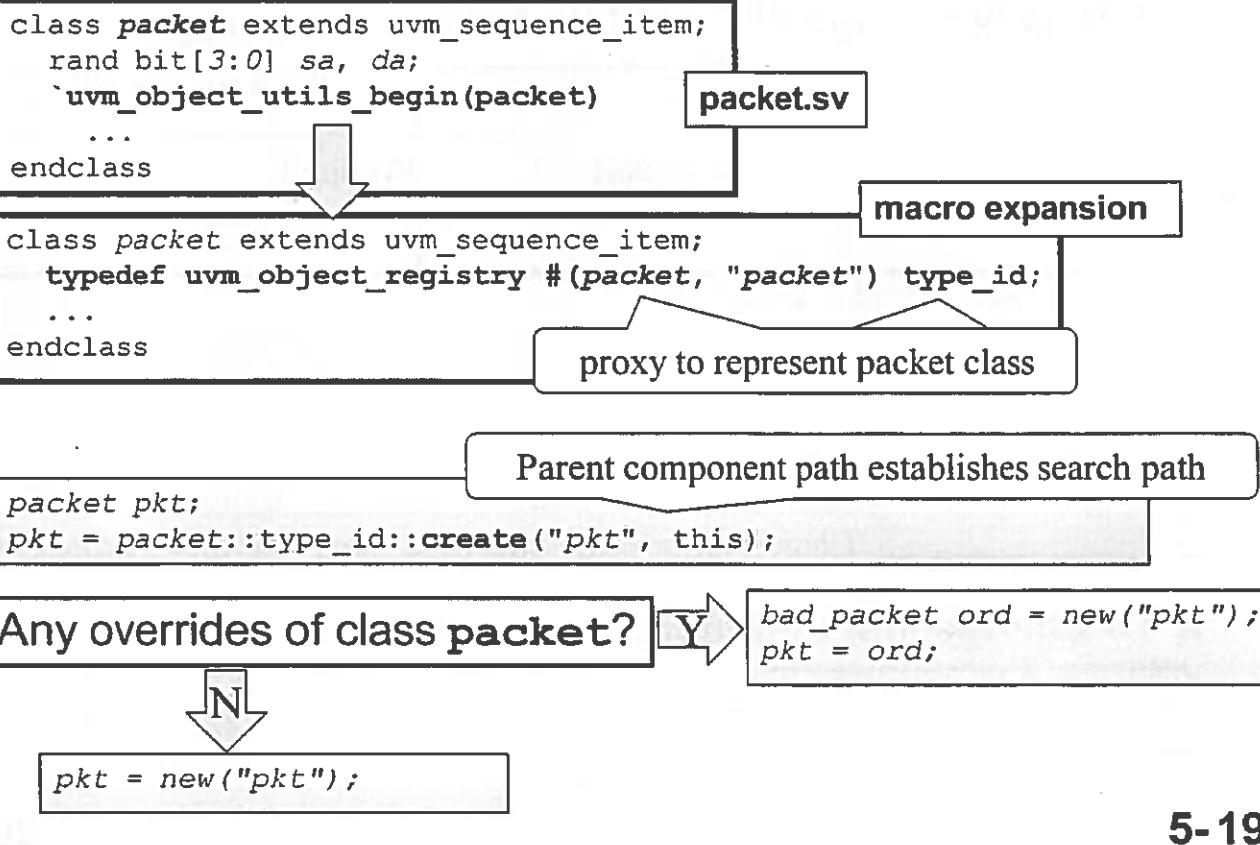
```
class monitor extends uvm_monitor;
    task run_phase(uvm_phase phase);
        forever begin
            packet pkt;
            pkt = packet::type_id::create("pkt", this);
            ...
        end
    endtask
endclass
```

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



5-19

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.

Customize Transaction in Test

■ Globally change all “packet” type

```
class my_packet extends packet;  
    int serial_no;  
    `uvm_object_utils_begin(my_packet)  
    ...  
endclass
```

Make modifications to existing transaction

Required!
Creates proxy class

```
class packet_test extends test_base;  
    `uvm_component_utils(packet_test)  
    virtual function void build_phase(uvm_phase phase);  
        super.build_phase(phase);  
        set_type_override_by_type(packet::get_type(), my_packet::get_type());  
    endfunction  
endclass
```

Change all existing “packet” proxy to “my_packet”

■ Or change just instances found in search path

```
set_inst_override_by_type("*mon(pkt", packet::get_type(),  
                           my_packet::get_type());
```

Search path

5-20

Component Factory

■ Similar mechanism for components

```
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 handle required!

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

5-21

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

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

driver.sv

```
class driver extends uvm_driver #(packet);
  typedef uvm_component_registry #(driver, "driver") type_id;
  ...
endclass
```

macro expansion

```
driver drv;
drv = driver::type_id::create("drv", this);
```

Any overrides of class **driver**
under "this" parent scope?

newDrv ord;
ord = new("drv", this);
drv = ord;



```
drv = new("drv", this);
```



5-22

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_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 override, construct an object of the default type, ie. **driver**.

Customize Component in Test

■ Globally change all “driver” type

```
class newDriver extends driver;  
  `uvm_component_utils(newDriver)  
  virtual task run_phase(...);  
    if (dut.sigs.done == 1)  
      ...  
  endtask
```

Required!
Creates proxy class

Make behavioral
modifications

```
class component_test extends test_base;  
  `uvm_component_utils(component_test)  
  virtual function void build_phase(uvm_phase phase);  
    super.build_phase(phase);  
    set_type_override_by_type(driver::get_type(), newDriver::get_type());  
  endfunction  
endclass
```

Change all existing “driver”

■ Or change just instances found in search path

```
set_inst_override_by_type("*.i_agent[5].drv", driver::get_type(),  
                           newDriver::get_type());
```

Search path

5-23

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 the overrides in the factory

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

■ Example:

```
+uvm_set_inst_override=driver,newDriver,* .drv0
```

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

No space character!

5-24

Visually Inspect Structural Correctness

- Topology of the test can be printed with

```
uvm_top.print_topology()
```

```
UVM_INFO @ 64250.0ns: reporter [UVMTOP] UVM testbench topology:  
-----  


| Name             | Type                   | Size | Value    |
|------------------|------------------------|------|----------|
| uvm_test_top     | test_base              | -    | @510     |
| env              | router_env             | -    | @517     |
| i_agent          | 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 |
| seqr             | uvm_sequencer          | -    | @557     |
| rsp_export       | uvm_analysis_export    | -    | @564     |
| ...              |                        |      |          |



```
function void final_phase(uvm_phase phase);
 uvm_top.print_topology();
endfunction
```


```

5-25

uvm_top is a global instance of a **uvm_root** class automatically generated by UVM

Can Choose Print Format

■ Topology can also be printed in tree format

```
UVM_INFO @ 64250.0ns: reporter [UVMTOP] UVM testbench topology:  
uvm_test_top: (test_base@510) {  
    env: (router_env@517) {  
        i_agent: (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  
            }  
            seqr: (uvm_sequencer@557) {  
                rsp_export: (uvm_analysis_export@564) {  
                    recording_detail: UVM_FULL  
                }  
            }  
            ...  
            function void final_phase(uvm_phase phase);  
                uvm_top.print_topology(uvm_default_tree_printer);  
            endfunction
```

5-26

The tree format is more verbose than the table format, but does not truncate names

Visually Inspect Factory Overrides

- You should always check the overrides with
`factory.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.i_agent*.seqr.*	packet_da_3

Type Overrides:

Requested Type	Override Type
driver	delayed_driver

```
function void final_phase(uvm_phase phase);
    uvm_top.print_topology(uvm_default_tree_printer);
    factory.print();
endfunction
```

5-27

Parameterized Component Class

■ Parameterized component requires a different macro

```
'uvm_component_param_utils_begin(cname)
  `uvm_field_*($ARG, $FLAG)
`uvm_component_utils_end

class my_driver #(width=8) extends uvm_driver #(packet);
  rand bit [width-1:0] data;
  `uvm_component_param_utils_begin(my_driver #(width))
    `uvm_field_int(data, UVM_DEFAULT)
  `uvm_component_utils_end
endclass
```

■ Creation of object requires parameter

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

Use typedef to minimize typo

5-28

Unit Objectives Review

Having completed 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**

5-29

Appendix

uvm_component_utils Macro
Variable Length Configuration
UVM Configuration Data Base API
UVM Configuration Debugging

uvm_component_utils Macro

uvm_component_utils Macros

`uvm_component_utils[_begin] (cname)

- Implements the following:

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

class uvm_component_registry #(type T=uvm_component, string Tname=<unknown>) extends uvm_object_wrapper;
typedef uvm_component_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); <-- Macro creates a proxy class which
end                                is registered in the uvm_factory
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
```

5-32

Variable Length Configuration

Configuring Variable Length Members

■ The following does not compile

```
class packet_sequence extends base_sequence; // some code left off
    bit valid_da[16];
    virtual task body();
        uvm_config_db#(bit)::get(get_sequencer(), get_name(),
                                  "valid_da", valid_da);
        ...
    endtask
endclass
```

Configuration can not handle arrays

■ The following works but not suitable if width changes

```
class packet_sequence extends base_sequence; // some code left off
    bit[15:0] valid_da;
    virtual task body();
        uvm_config_db#(bit[15:0])::get(get_sequencer(), get_name(),
                                       "valid_da", valid_da);
        ...
    endtask
endclass
```

What if size changes?

5-34

Configuring Variable Length Members

- Use `uvm_bitstream_t` to minimize `set()` and `get()` changes when bit width changes

```
class packet_sequence extends base_sequence; // some code left off
    bit[31:0] valid_da;
    virtual task body();
        uvm_config_db#(uvm_bitstream_t)::get(get_sequencer(), get_name(),
                                                "valid_da", valid_da);
    endtask
endclass
```

Configuration remains the same even if `valid_da` bit width changes

```
class test_bitstream extends test_base; // some code left off
    virtual function void start_of_simulation(uvm_phase phase); ...
        bit[31:0] valid_da;
        std::randomize(valid_da);
        uvm_config_db#(uvm_bitstream_t)::set(this,
                                                "env.i_agent.seqr.packet_sequence", "valid_da", valid_da);
    endtask
endclass
```

Configuration remains the same even if bit width changes

Caveat: `uvm_bitstream_t` is 4096 bits wide, consuming a lot of memory.

5-35

UVM Configuration Data Base API

uvm_config_db::set() - (1/2)

```
uvm_config_db#(int)::set(  
.cntxt(this),  
.inst_name("env.i_agent*.seqr"),  
.field_name("item_count"),  
.val(15));
```

The `cntxt` field is important to set the precedence of various `set()` calls for the same resource. If you use null at lower levels of component hierarchy it sets the precedence of the `set` to `uvm_root`. It is recommended that you always use `this`.

1. `uvm_config_db#(T)` contains an array of pools, one pool per context (scope of configuration) per config type in `m_rsrc[cntxt]`
2. Each pool is a parameterized object `uvm_pool#(string KEY, T)`, where `KEY = {inst_name, field_name}`
3. Each pool stores one resource per key of the config type per context
4. Each resource stores three important elements for retrieval
 - i. `string name = field_name`
 - ii. `string scope = {cntxt.get_full_name(), ".", inst_name}`
 - iii. `T val = value passed in the uvm_config_db::set() call`
5. The resource is then added to the global resource pool under the name `field_name` (array `rtab`) and separately under the type `T` (array `ttab`)
6. For each name or type there is a queue of resources stored with priority. The scope is used to retrieve the matching resource of a particular name.

```
class uvm_pool #(type KEY=int, T=uvm_void) extends uvm_object;  
protected T pool[KEY];  
endclass
```

5-37

uvm_config_db::set() - (2/2)

```

uvm_config_db#(int)::set(this, "env.i_agent*.seqr", "item_count", 15); //this = uvm_test_top

class uvm_config_db#(T) extends uvm_resource_db;
//local config class - all methods static
//creates pool per context
//!!!pseudo code!!!

static uvm_pool#(string, uvm_resource#(T)) m_rsc[cntxt];
static function bit set(uvm_component cntxt,
    string inst_name,
    string field_name,
    T val);

inst_name = {cntxt.get_full_name(),".",inst_name};
string key = {inst_name,field_name};
if(!exists(m_rsc[cntxt]))
    uvm_pool p0 = new; //new uvm_pool object
m_rsc[cntxt] = p0; //one pool per context
r0 = new(field_name, inst_name); //resource object
p0[key] = r0; //for local lookup
r0.val = val;//store value in resource object
//uvm_resources is handle to global pool singleton
uvm_resources.set(r0); //store resource in global pool

```

cntxt	pool
uvm_test_top	p0
uvm_test_top.env...	p1

class uvm_pool#(KEY,T) :	
key	resource
uvm_test_top..item_count	r0

```

class uvm_resource_pool ;
//singleton global pool
//details in earlier slides

```

rtab

Name	Queue
item_count	rq

priority



```

class uvm_resource#(T);
//resource container
string name; //uvm_object
protected string scope;
T val;

```

5-38

uvm_config_db::get() - (1/2)

```
uvm_config_db#(int)::get(  
//this = uvm_test_top.env  
.cntxt(this),  
.inst_name("i_agent.seqr"),  
//!!!do not use globs in inst_name!!!  
.field_name("item_count"),  
.val(item_count));
```

The cntxt field is important to set the precedence of various get() calls for the same resource. If you use null it sets the precedence of the get to uvm_root. It is recommended that you always use this.

1. Call `uvm_resource_pool::lookup_regex_names(...)`
retrieve queue of resources that matches scope
`"uvm_test_top.env.i_agent.seqr"`
and name ("item_count")
and type_handle (`uvm_resource#(int)`)
`rq = lookup_regex_name(scope, name, type_handle);`
`//calls q = lookup_name(scope, name, type_handle, rpterr);`
 - A. The lookup includes regex matches – uses DPI call `uvm_re_match()`
 - B. retrieve resource with highest priority
`rsrc = get_highest_precedence(q);`
2. Set local variable item_count to the resource value
`val = rsrc.read(cntxt); //item_count gets value of resource`
3. Return success or failure status

uvm_config_db::get() - (2/2)

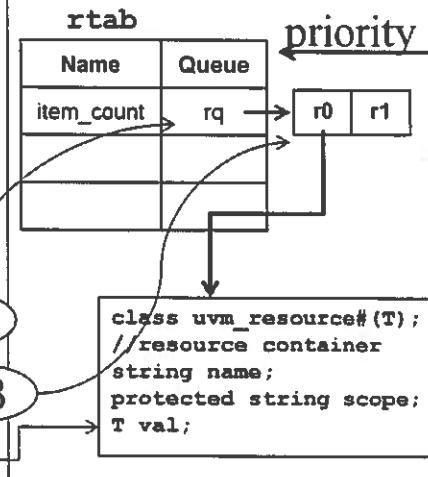
```
uvm_config_db#(int)::get(this, //e.g. this = uvm_test_top.env
  "i_agent.seqr",
  "item_count",
  item_count) // local variable;

class uvm_config_db#(T) extends uvm_resource_db;
//local config class
//all methods static
//creates pool per context
//!!!!pseudo code!!!
static uvm_pool#(string, uvm_resource#(T)) m_rsc[cntxt];

static function bit get(uvm_component cntxt,
  string inst_name,
  string field_name,
  T val);

  uvm_resource_types::rsrc_q_t rq;
  uvm_resource_base r;
  inst_name = {cntxt.get_full_name(), .inst_name}
  //call lookup_name as described earlier
  rq = rp.lookup_regex_names(inst_name, field_name,
    uvm_resource#(T)::get_type());
  r = uvm_resource#(T)::get_highest_precedence(rq);
  val = r.read(cntxt); //read value in resource object
  return 1; //success!!
```

```
class uvm_resource_pool ;
//singleton global pool
//details in earlier slides
```



5-40

UVM Configuration Debugging

Debugging configuration issues (1/4)

■ Common issues

- configuration is strongly typed
- mismatched types between the set and get calls will not flag an error during get.

```
uvm_config_db#(int)::set::(this,"env","item_count",10);  
uvm_config_db#(int unsigned)::get(this,"","item_count", item_count);
```

- mismatched field names and/or scopes due to typos and change in hierarchy will not flag an error

5-42

Debugging configuration issues (2/4)

■ Always check return status

- when doing a `read_by_name/_type()` or `get()` check the return status and print an error message

```
if(! uvm_config_db#(int unsigned)::get(this,"","item_count", item_count))
  `uvm_info("CFGERR", $sformatf("item_count not available for %s",
                                this.get_full_name()), UVM_MEDIUM);
```

- use run-time switch available to trace all config sets and gets

```
+UVM_CONFIG_DB_TRACE - works only with uvml.1  
+UVM_RESOURCE_DB_TRACE - works only with uvml.1
```

- use tracing controls in test code

- `uvm_config_db_options::turn_tracing_on() // turn tracing on`
- `uvm_config_db_options::turn_tracing_off() //turn tracing off`
- `uvm_config_db_options::is_tracing() //check status of tracing`

5-43

Debugging configuration issues (3/4)

■ Global handle available to UVM resource pool

`uvm_resources`

■ Dumping resource data

- Dump all resources in the resource pool using `dump()`

```
uvm_resources.dump(.audit(1)); //audit==1 displays resource access details
```

Sample output:

```
default_sequence [/^uvm_test_top\.env\.i_agent\.seqr\.main_phase$/] : (class
uvm_pkg::uvm_object_wrapper) ?

-----
uvm_test_top.env.i_agent reads: 0 @ 0.0ns writes: 1 @ 0.0ns
uvm_test_top.env.i_agent.seqr reads: 1 @ 0.0ns writes: 0 @ 0.0ns
```

- Dump all 'gets' done using `uvm_resource_db::read_by_name()/_type()`

```
uvm_resources.dump_get_records();
```

5-44

Debugging configuration issues (4/4)

■ Find all unused resources (on which no gets were performed)

- see code in notes section

```
uvm_resources.find_unused_resources();
```

Sample Output of code below:

```
====Unused Resources====
```

```
delay [/^uvm_test_top].env$/] : (int) 5
```

Name	Type	Size	Value
delay	<unknown>	-	@510

```
uvm_test_top.env reads: 0 @ 0.0ns writes: 1 @ 0.0ns
```

5-45

```
//generally put this code in start_of_simulation_phase() and/or final_phase()
if (uvm_report_enabled(UVM_HIGH, UVM_INFO, "UNUSED RSRC"))
begin
    uvm_resource_types::rsrc_q_t      q;
    uvm_resource_base    r;
//find_unused_resources() returns all resources that were written but not read.
    q    =  uvm_resources.find_unused_resources();
    `uvm_info("UNUSED RSRC", $sformatf("\n\n====Unused
Resources====\n"), UVM_HIGH)
    if (!q.size())
        `uvm_info("UNUSED RSRC", $sformatf("\n\n==== No Unused
Resources Found====\n"), UVM_HIGH)
    else
        for( int i = 0; i < q.size(); i++)
        begin
            r = q.get(i);
            r.print(); //print resource
            r.print_accessors(); //print all accessors
        end
        `uvm_info("UNUSED RSRC", $sformatf("\n====End Unused
Resources====\n\n"), UVM_HIGH)
end
```

UVM Resource and Config DB debug

- Within your code you can enable dumping
 - `resources.dump(audit);`
- Command line based debug
 - `./simv +UVM_RESOURCE_DB_TRACE +UVM_CONFIG_DB_TRACE`

```
UVM_INFO /fs/Release/linux RH4 AMD64 TD 32 debug_Engineer/etc/uvm-1.1/base/uvm_resource_db.svh(130) @ 0.0ns: reporter [CFGDB/GET] Configuration 'uvm_test_top.recording_detail' (type logic signed[4095:0]) read by uvm_test_top = null (failed lookup)
UVM_INFO /fs/Release/linux RH4 AMD64 TD 32 debug_Engineer/etc/uvm-1.1/base/uvm_resource_db.svh(130) @ 0.0ns: reporter [CFGDB/GET] Configuration 'uvm_test_top.recording_detail' (type int) read by uvm_test_top = null (failed lookup)
UVM_INFO @ 0.0ns: reporter [RNTST] Running test test_seq_lib_cfg...
UVM_INFO @ 0.0ns: reporter [UVM_CMDLINE_PROC] Applying config setting from the command line: +uvm_set_config_int=uvm_test_top.env.foo,3
UVM_INFO /fs/Release/linux RH4 AMD64 TD 32 debug_Engineer/etc/uvm-1.1/base/uvm_resource_db.svh(130) @ 0.0ns: reporter [CFGDB/SET] Configuration 'uvm_test_top.env.foo' (type logic signed[4095:0]) set by = (logic signed[4095:0]) 11
UVM_INFO /fs/Release/linux RH4 AMD64 TD 32 debug_Engineer/etc/uvm-1.1/base/uvm_resource_db.svh(130) @ 0.0ns: reporter [CFGDB/GET] Configuration 'uvm_test_top.env.recording_detail' (type logic signed[4095:0]) read by uvm_test_top.env = null (failed lookup)
UVM_INFO /fs/Release/linux RH4 AMD64 TD 32 debug_Engineer/etc/uvm-1.1/base/uvm_resource_db.svh(130) @ 0.0ns: reporter [CFGDB/GET] Configuration 'uvm_test_top.env.recording_detail' (type int) read by uvm_test_top.env = null (failed lookup)
```

5-46

Agenda: Day 2

**DAY
2**

5 Component Configuration & Factory

6 Component Communication

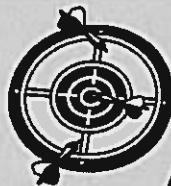


7 Scoreboard & Coverage

8 UVM Callback



Unit Objectives



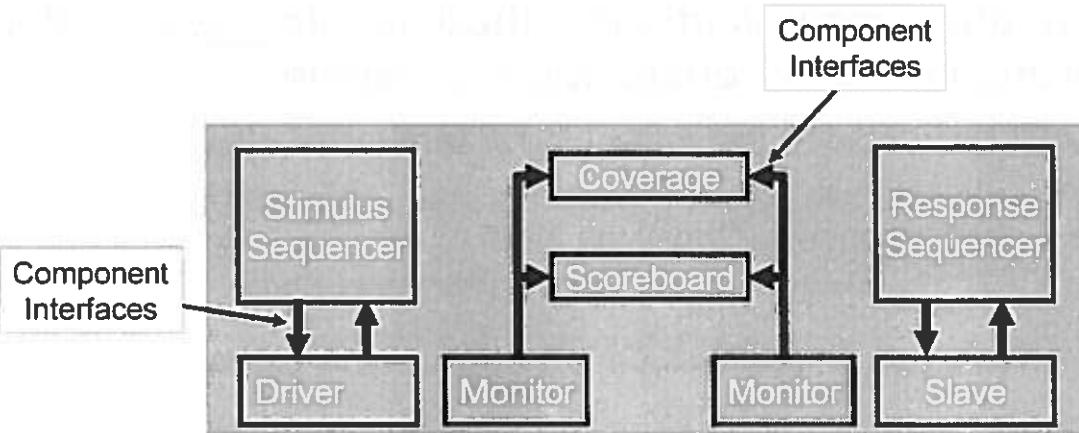
After completing this unit, you should be able to:

- **Describe and implement TLM port/socket for communication between components**
- **Describe and implement UVM notification for synchronization between behavioral units (components and sequences)**

6-2

Component Communication: Overview

- Need to exchange transactions between components of verification environment
 - Sequencer → Driver
 - Monitor → Collectors (Scoreboard, Coverage)



6-3

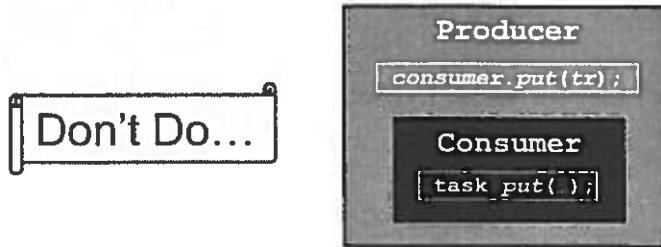
Component Communication: Method Based

- 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

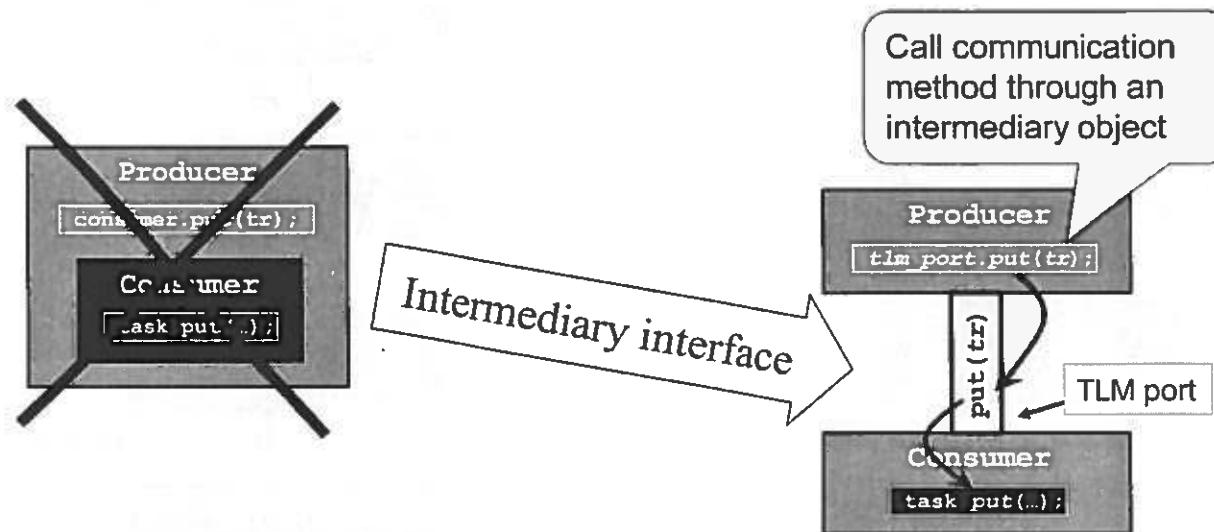
In the example at the bottom of the slide, because the Producer class uses a handle of type Consumer, it is restricted to communicating with just that type of object.

Even if the handle is of type uvm_component, the Producer must be passed this handle from a higher level such as the environment.

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 execution of the procedural communication



6-5

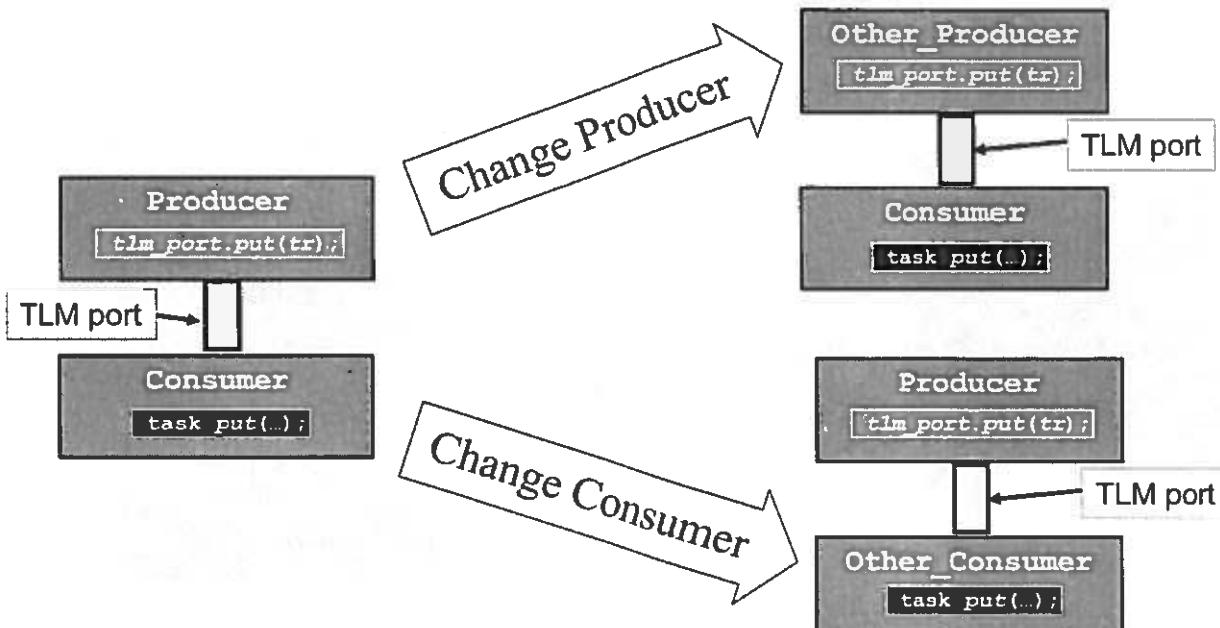
The TLM object provides a level of indirection between the producer and consumer so that they can be decoupled.

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 handled at a higher level, and can be changed during the simulation.

Component Communication: TLM

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



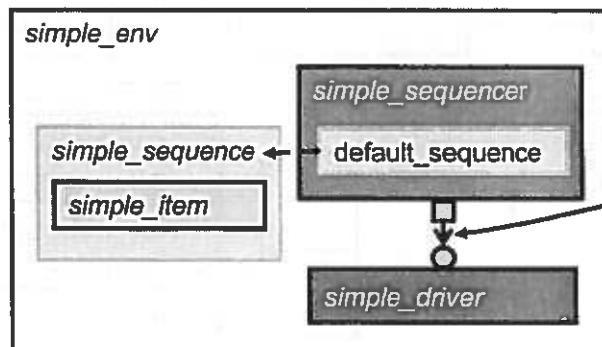
6-6

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

■ Push



■ Pull



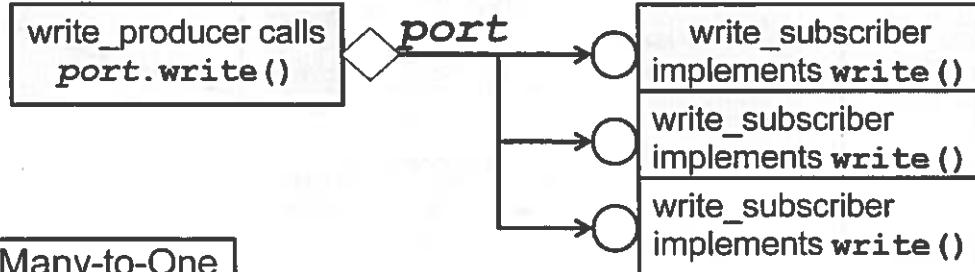
■ Fifo

One-to-One



■ Analysis (broadcast)

One-to-Many



Many-to-One

6-8

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

■ 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

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
```

6-9

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

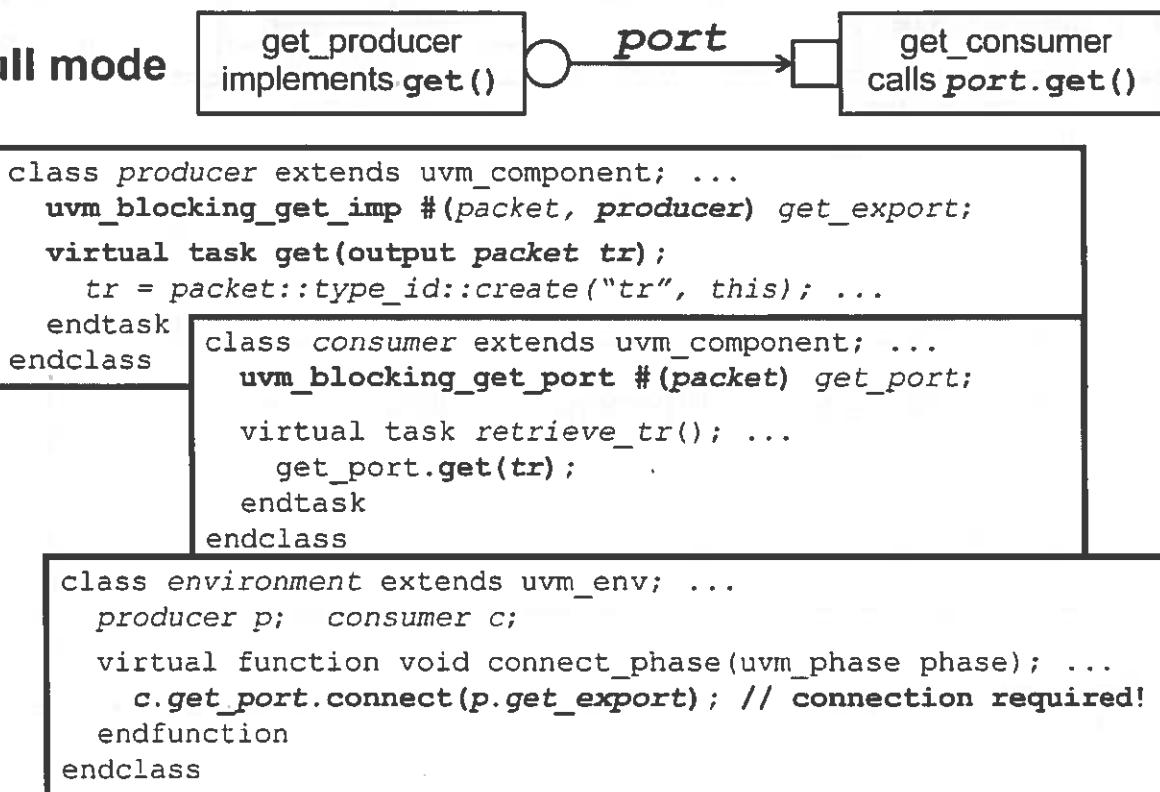
*_export

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

■ Pull mode



6-10

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

In the above diagram, the `build_phase()` method is required but not shown.

FIFO Mode

- FIFO Mode producer calls `port.put()`
- Connect producer to consumer via `uvm_tlm_fifo` consumer calls `port.get()`

```
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); // connection required!
    c.get_port.connect(tr_fifo.get_export); // connection required!
endfunction
endclass
```

6-11

Analysis Port

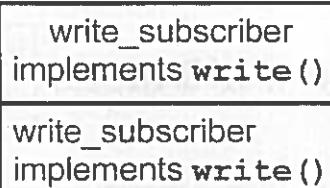
■ Analysis (broadcast)

- Analysis port can be left unconnected

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

class subscriber extends uvm_component; ...
  uvm_analysis_imp #(packet, subscriber) analysis_export;
  virtual function void write(packet tr); // cannot block
    process_transaction(tr);
  endfunction

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
```



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

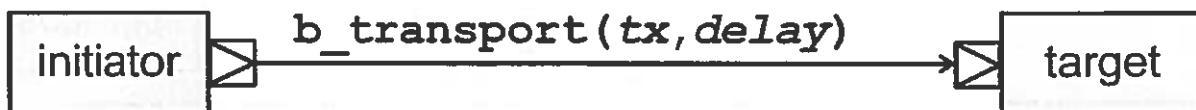
class agent extends uvm_agent; // other code not shown ...
  monitor mon;
  uvm_analysis_port #(packet) analysis_port;
  virtual function void build_phase(uvm_phase phase); ...
    this.analysis_port = new("analysis_port", this);
  endfunction
  virtual function void connect_phase(uvm_phase phase); ...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

port
must be
same
type

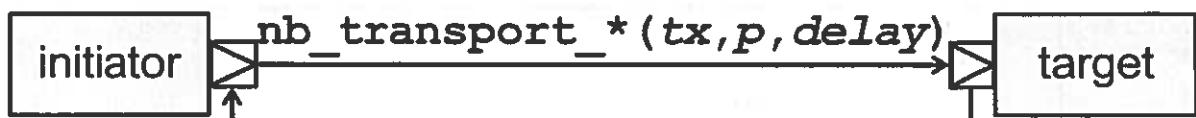
6-13

UVM TLM 2.0

■ Blocking



■ Non-Blocking



6-14

Blocking Transport Initiator



```
class initiator extends uvm_component;
    uvm_tlm_b_initiator_socket #(packet) i_socket;
    // constructor 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

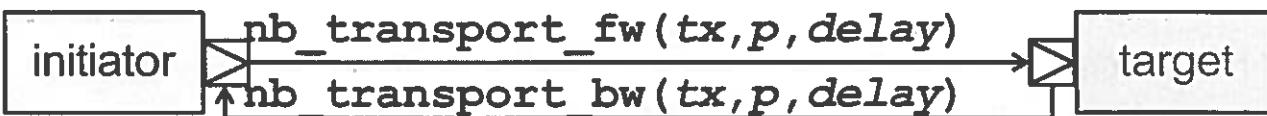
Blocking Transport Target



```
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

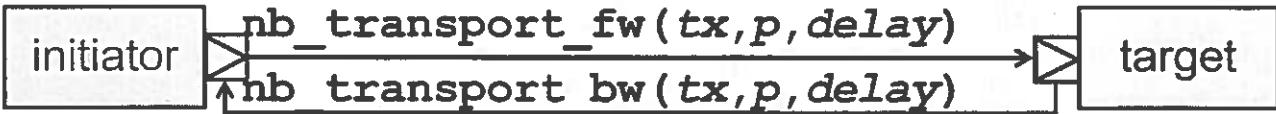
Non-Blocking Transport Initiator



```
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);
    phase.raise_objection(this);
    tx.randomize();
    sync = i_socket.nb_transport_fw(tx, p, delay);
    phase.drop_objection(this);
  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

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

Having completed this unit, you should be able to:

- **Describe and implement TLM port/socket for communication between components**
- **Describe and implement UVM notification for synchronization between behavioral units (components and sequences)**

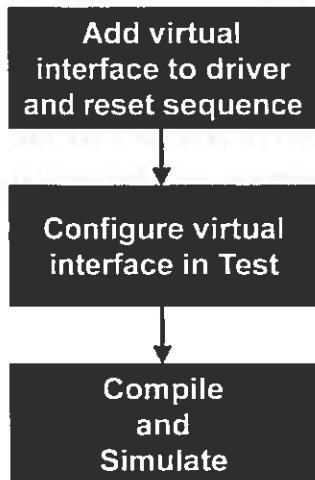
6-19

Lab 4 Introduction

Implement & configure physical device drivers



30 min



6-20

Appendix

TLM 2.0 Generic Payload

DVE Debugging Features

DVE UVM-Aware Debugging Features

TLM 2.0 Generic Payload

TLM 2.0 Generic Payload (1/4)

- TLM 2.0 Generic Payload is a generic bus read/write access transaction type

- 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;
```

6-23

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
```

6-24

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
```

6-25

TLM 2.0 Generic Payload (4/4)

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

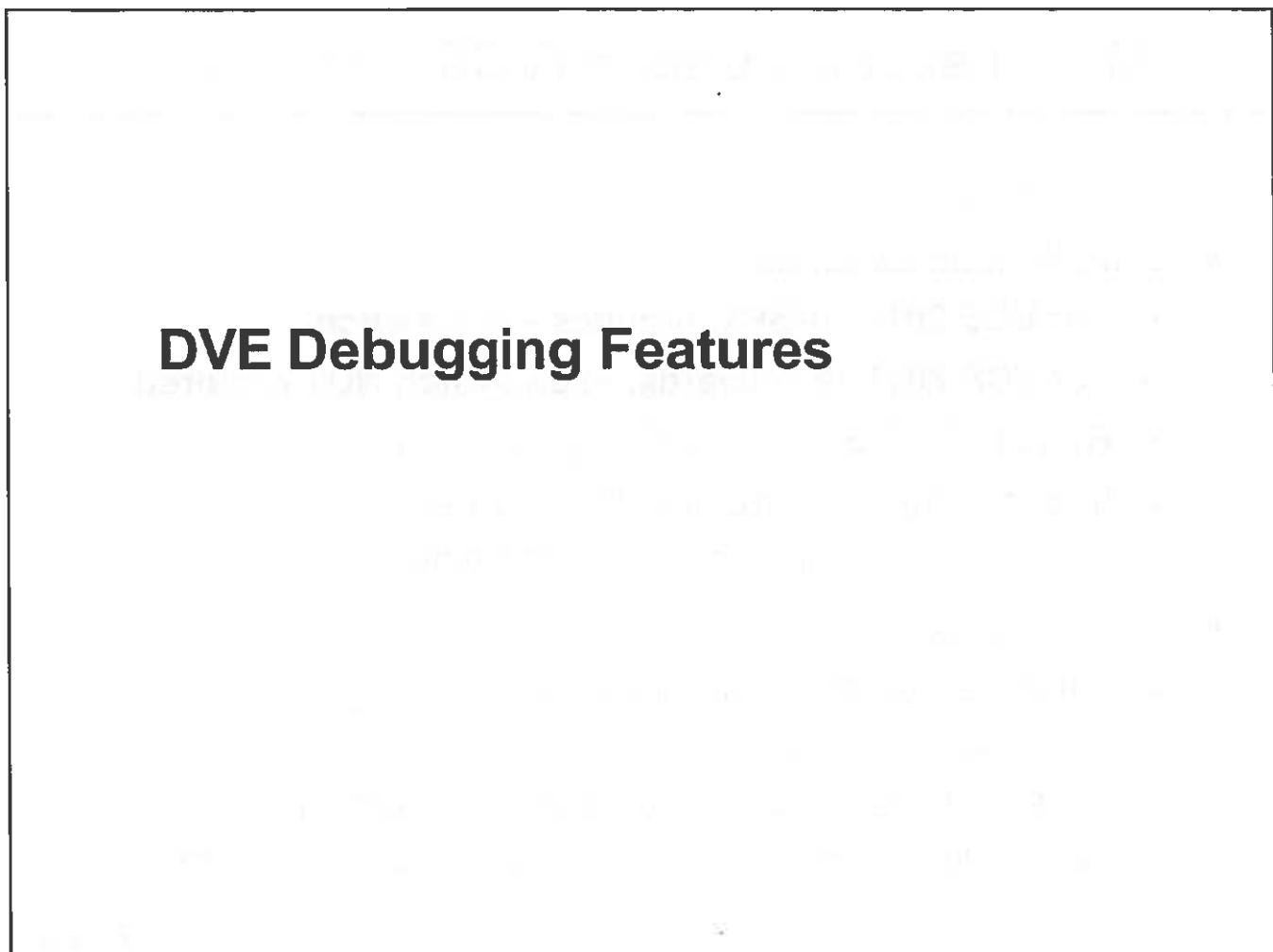
- Requires implementation of uvm_tlm_extension class

```
// 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
```

6-26

DVE Debugging Features



UVM Transaction Debug (VCS Installation)

For VCS 2011.03-SP1 onwards

■ Compile-time switches:

- For VCS 2011.03-SP1, requires `-lca` switch
- For VCS 2011.12 onwards, `-lca` switch NOT required
- Requires `-debug` or `-debug_pp` or `-debug_all`
- If recording is not desired (for regression)
 - ◆ `-tld_logoff` (compiles out recording)

■ Run-time control:

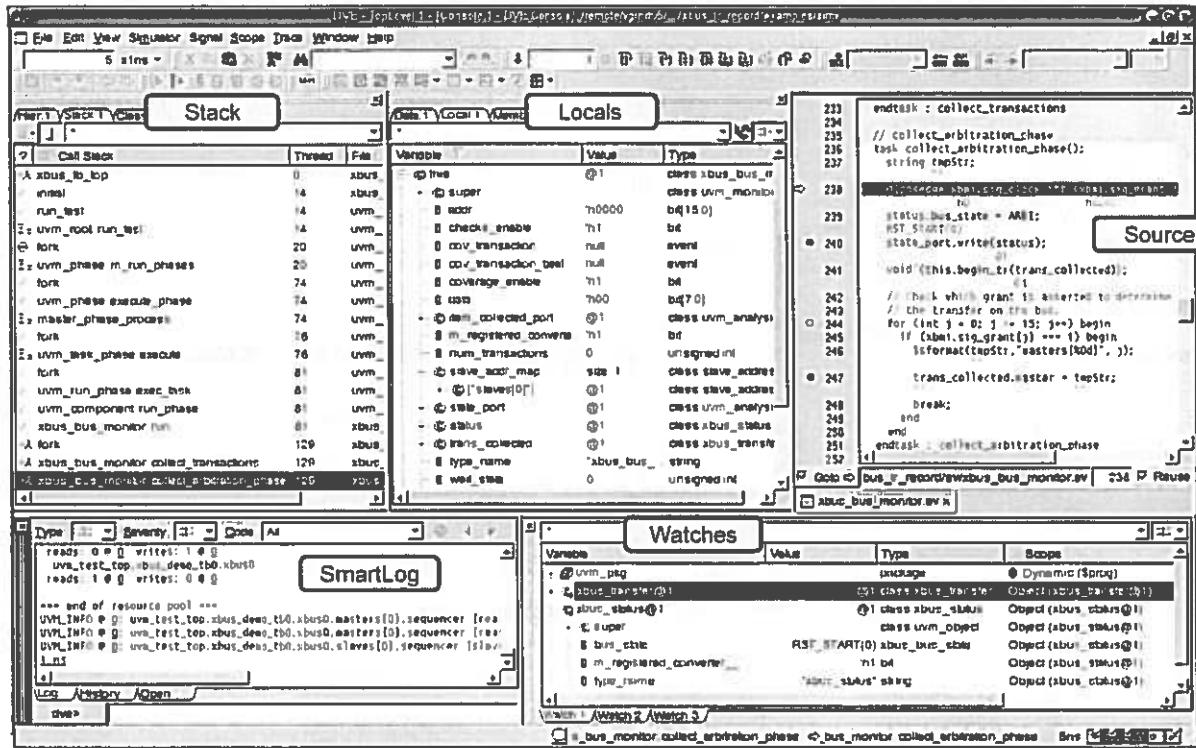
- `+UVM_TR_RECORD` for transaction recording
 - ◆ visualize transactions in time on waveform
- `+UVM_LOG_RECORD` for log message recording
 - ◆ visualize log messages within components on waveform

6-28

Additional compile-time switches if custom UVM source code is used

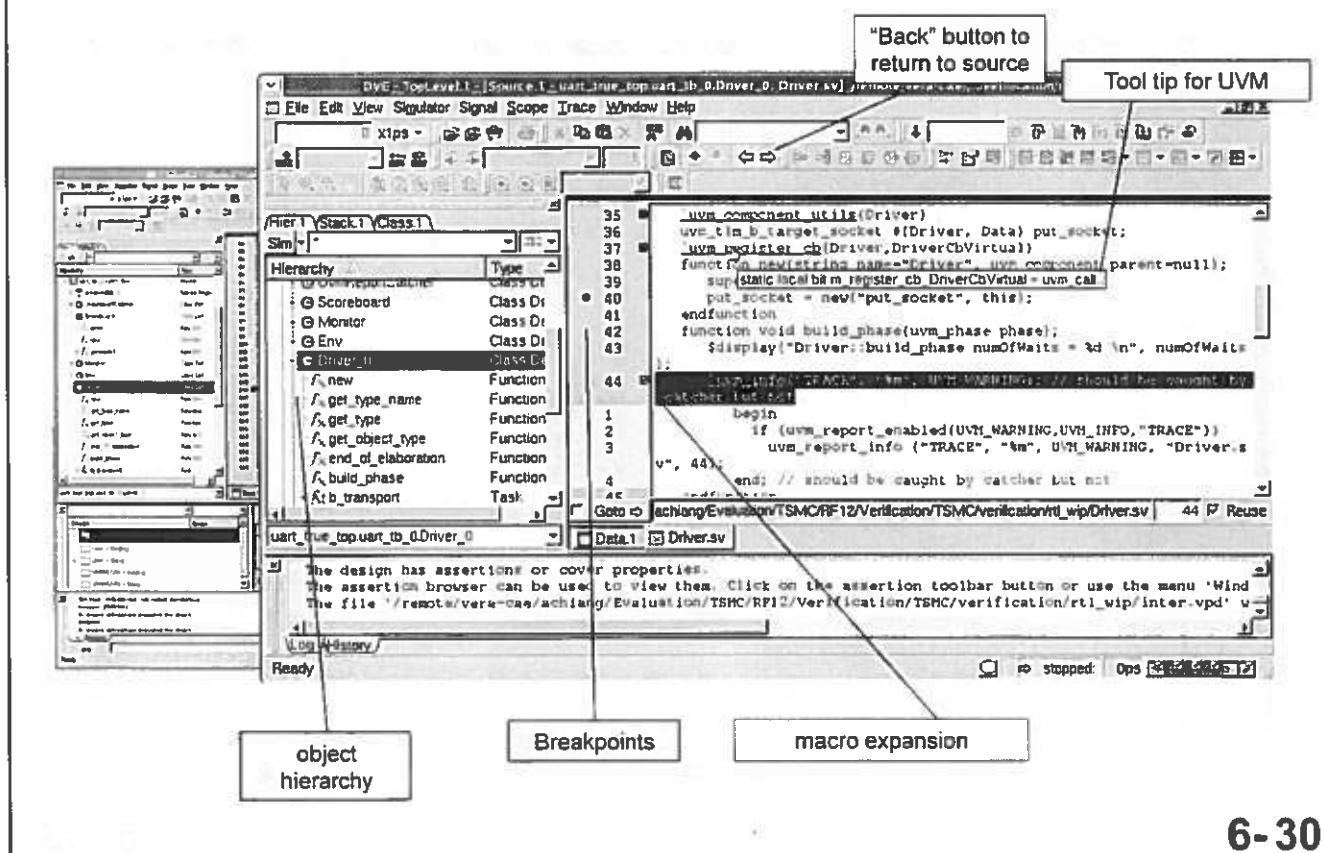
```
+incdir+${UVM_HOME}/src  
${UVM_HOME}/src/uvm_pkg.sv  
${UVM_HOME}/src/dpi/uvm_dpi.cc -CFLAGS -DVCS  
+incdir+${VCS_HOME}/etc/uvm-1.1/vcs  
${VCS_HOME}/etc/uvm-1.1/vcs/uvm_custom_install_vcs_recorder.sv
```

DVE Methodology-Aware Panes



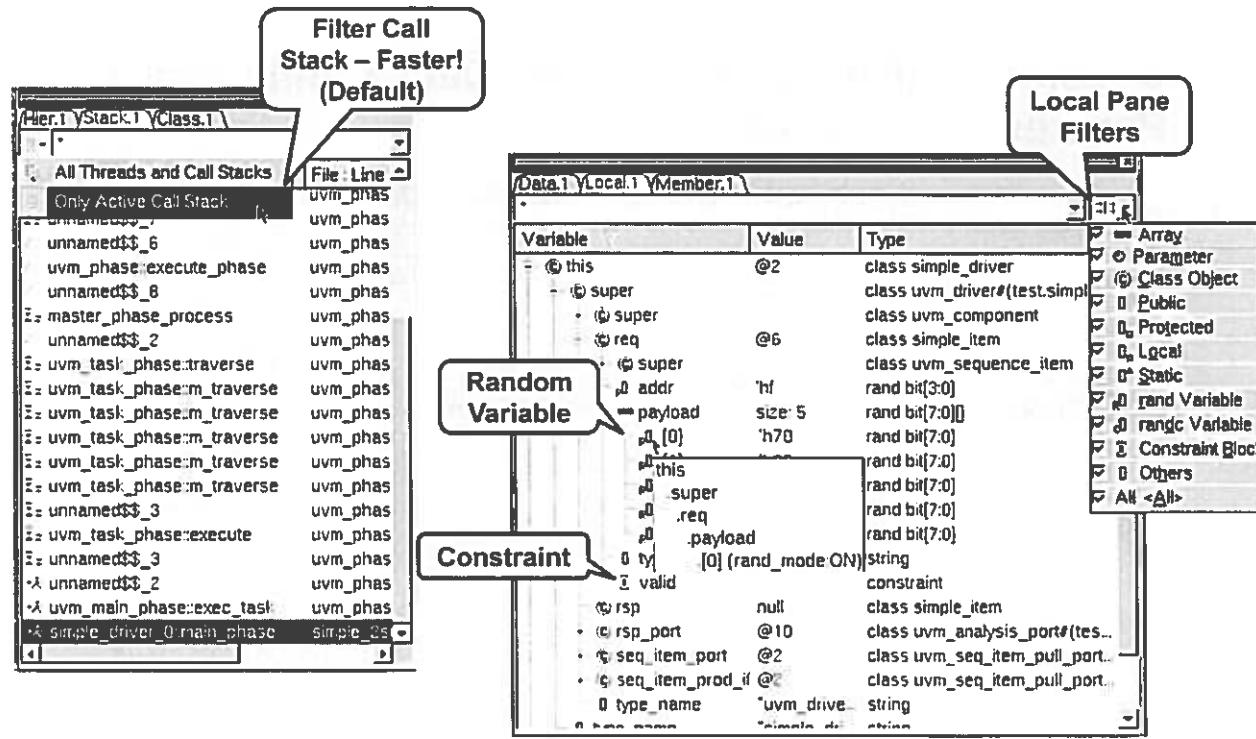
6-29

Managing Debugging Session with DVE



6-30

Local and Stack Pane Usage

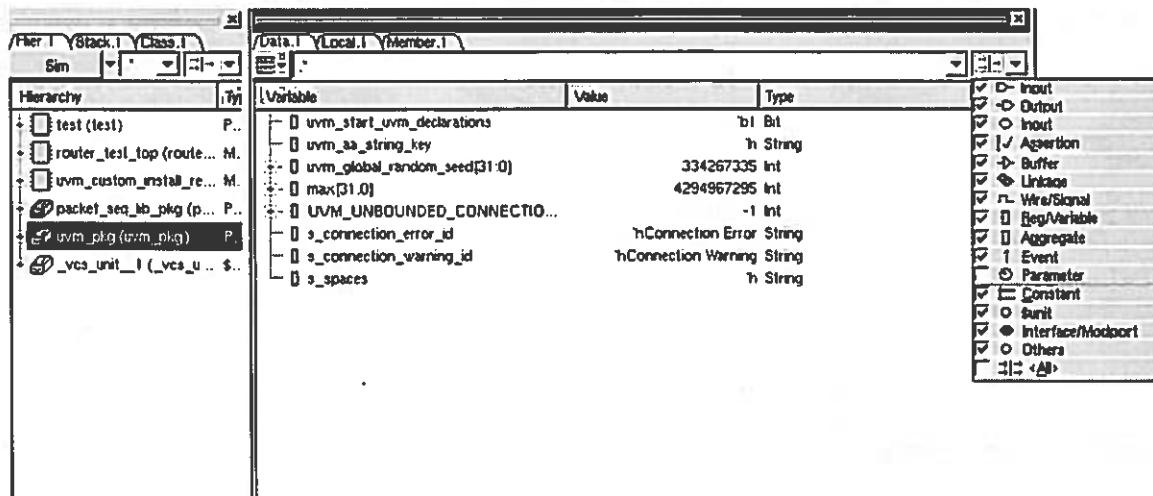


6-31

Filtering the active call stack provides additional performance with large designs. It shows only the active call stack in the stack pane.

Filtering Data

- Deselect PARAMETER to make Data.1 and Local.1 Pane more readable



6-32

Watch Object in DVE

- Send uvm_pkg.uvm_top.uvm_top_levels[0] to Watch.1 pane

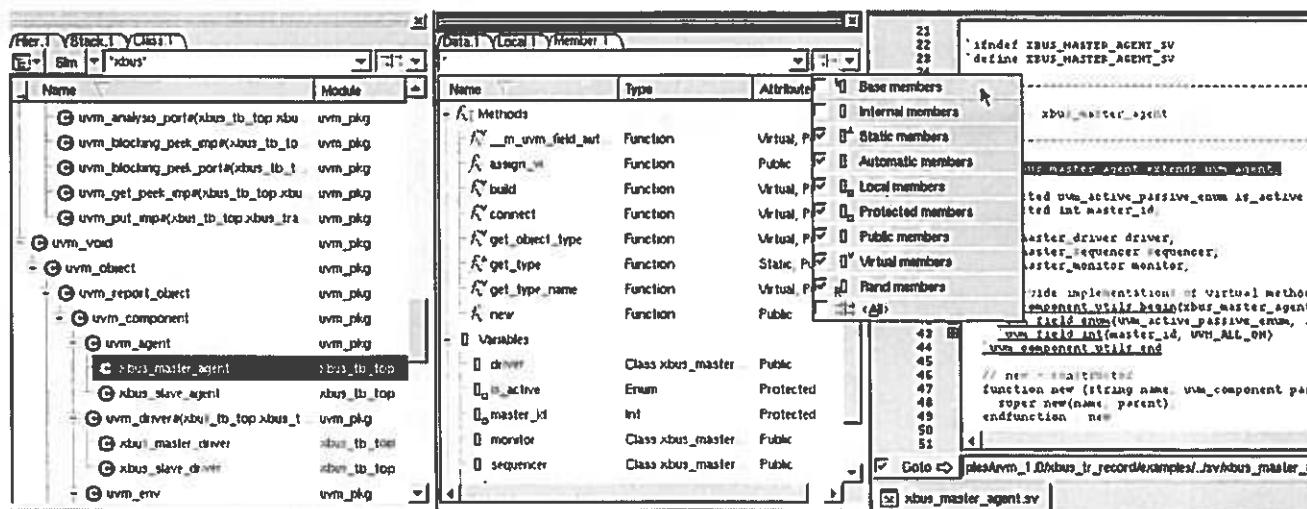
The screenshot shows the DVE (Digital Verification Environment) interface. On the left is the 'Watch.1' pane, which displays a hierarchical list of variables. One variable, 'uvm_top_levels[0]', is selected and expanded, showing its type as 'class uvm_component (test_seq_lb_cfg){}' and its value as '0x0'. The right side of the interface is the 'Watch.2' pane, which lists variables with their values, types, and scopes. The variable 'uvm_top_levels[0]' is also present here with the same details. The bottom of the interface shows tabs for 'Watch 1', 'Watch 2', and 'Watch 3', with 'Watch 2' currently active.

Variable	Value	Type	Scope
uvm_top_levels[0]	0x0	class uvm_component (test_seq_lb_cfg){}	Dynamic (\$prog)
super		class test_base	Dynamic (\$prog)
super		class uvm_test	Dynamic (\$prog)
type_name	"test_base"	string	Dynamic (\$prog)
env	0x1	class router_env	Dynamic (\$prog)
super		class uvm_env	Dynamic (\$prog)
foo	10	int	Dynamic (\$prog)
sb	null	class scoreboard	Dynamic (\$prog)
r_seqr	null	class uvm_sequencer	Dynamic (\$prog)
h_agent	null	class host_agt	Dynamic (\$prog)
regmodel	null	class ral_block	Dynamic (\$prog)
type_name	"router_env"	string	Dynamic (\$prog)
agent	size 16	class router_a	Dynamic (\$prog)
omon	size 16	class omonito	Dynamic (\$prog)
seq_cfg	0x1	class uvm_seq	Dynamic (\$prog)
super		class uvm_obs	Dynamic (\$prog)
type_name	"uvm_sequence_k"	string	Dynamic (\$prog)
selection_mode	UVM_SEQ_LIB	uvm_sequenc	Dynamic (\$prog)
min_random_count	1	unsigned int	Dynamic (\$prog)
max_random_count	1	unsigned int	Dynamic (\$prog)
type_name	"test_seq_lb_cfg"	string	Dynamic (\$prog)

6-33

Class Browser in DVE

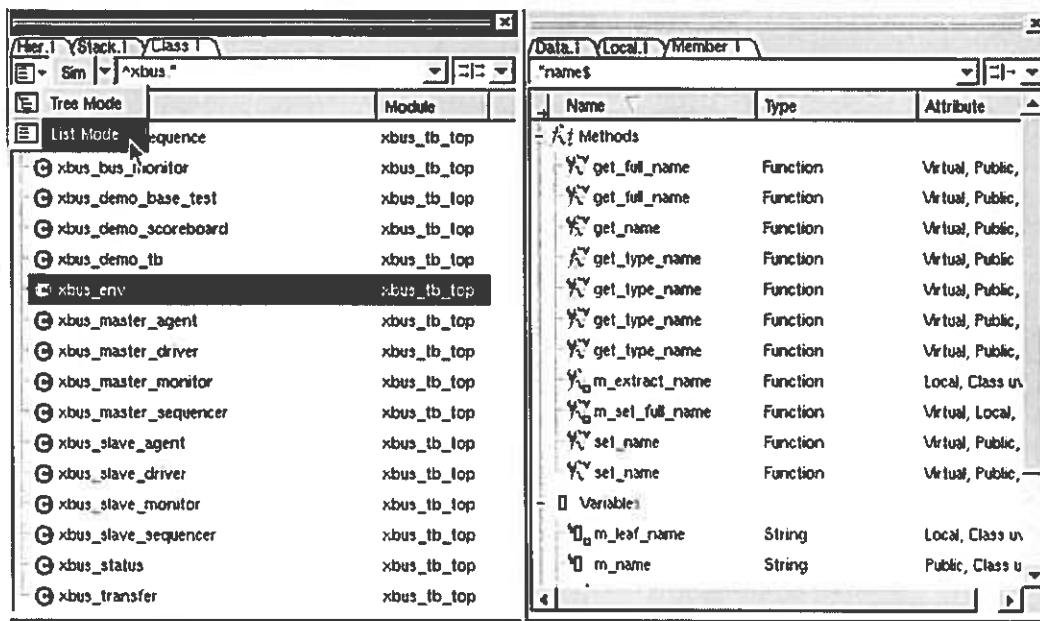
- All classes as derivation tree or flat list
- All methods, variables and constraints (optionally including base classes)
- Filtering on class names, member names and types



6-34

Filtering Classes

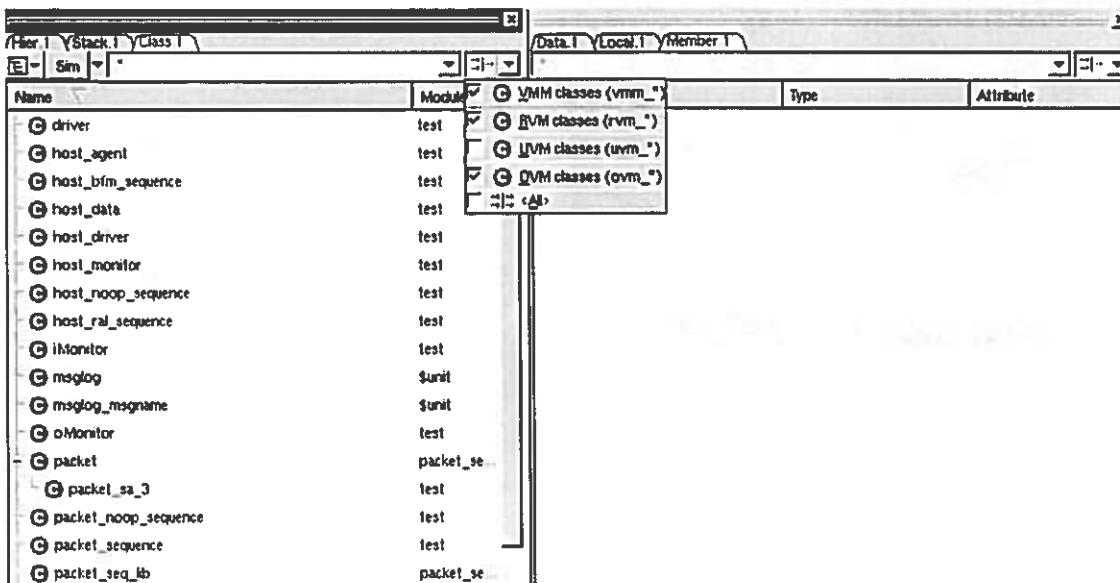
- Select List mode in Class Browser for easier filtering
- Use regex rather than wildcarding for more powerful pattern matching



6-35

Hiding Methodology classes

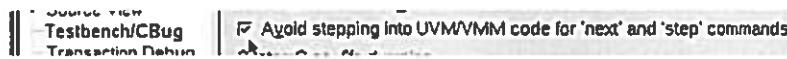
- Deselect UVM to make Class Browser more readable



6-36

Other Testbench Tricks

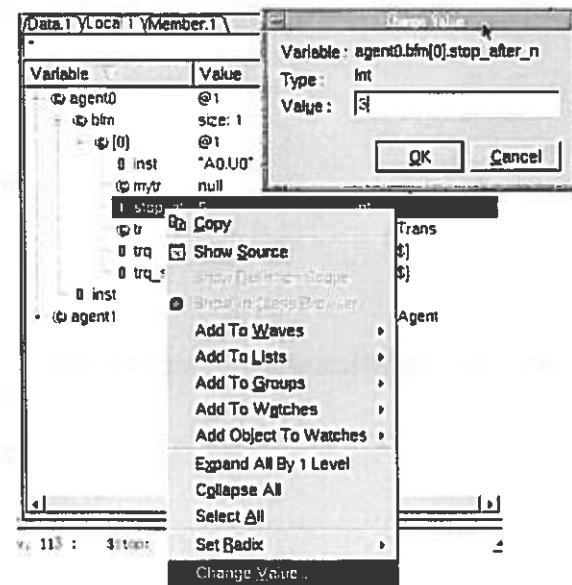
■ Preference to not step into UVM/VMM source



■ Set dynamic object values

■ Structs/Mailbox also visible

Variable	Value	Type
- @this	@1	class myclass
- @p1		struct packed ps
- @a	'hx	reg[2 0]
- @aps		struct packed
- @a	'hx	reg[2 0]
- @bl	'h00	byte
- @b	'h0	bit[2 0]
+ @ps_nested		struct packed
+ @mc	size 4	class myclass[3 0]



6-37

Thread Debug

Select Thread mode from drop-down

The screenshot shows the Thread Debug interface with two main windows:

- Call Stack**: A table showing the call stack for the current thread. It includes columns for File:Line and Object Id.
- Threads**: A table showing all threads and their scopes. It includes columns for Thread, Scope, File:Line, and Object Id.

In the 'Threads' window, the selected thread is highlighted with a yellow background. The call stack for this thread is also displayed in a separate window below it.

File:Line	Object Id
xbus_tb_top.sv:35	
initial.sv:65	
run_test.sv:40	
uvm_root.sv:392	
uvm_root.sv:395	
uvm_phase.sv:1771	
uvm_phase.sv:1772	
uvm_phase.sv:1084	
uvm_phase.sv:1157	
uvm_phase.sv:1166	
uvm_task.sv:137	
uvm_task.sv:150	
uvm_common_phases.sv:245	
uvm_component.sv:2287	
xbus_slave_driver.sv:51	
xbus_slave_driver.sv:52	
xbus_slave_driver.sv:63	
xbus_slave_driver.get_and_drive.sv:99	

Scope	File:Line	Object Id
uvm_sequence_base.start.sv:287	uvm_sequence_base.sv:287	
uvm_sequencer_base.wait_for_item_done.sv:850	uvm_sequencer_base.sv:850	xbus
xbus_slave_monitor.run.sv:137	xbus_slave_monitor.sv:137	
xbus_slave_monitor.collect_data_phase.sv:197	xbus_slave_monitor.sv:197	
xbus_slave_driver.run.sv:51	xbus_slave_driver.sv:51	
xbus_slave_driver.reset_signals.sv:71	xbus_slave_driver.sv:71	xbus
xbus_slave_driver.request_to_transfer.sv:156	xbus_slave_driver.sv:156	
uvm_sequence_base.start.sv:287	uvm_sequence_base.sv:287	
uvm_sequencer_base.wait_for_item_done.sv:850	uvm_sequencer_base.sv:850	xbus
xbus_master_monitor.run.sv:113	xbus_master_monitor.sv:113	
xbus_master_monitor.collect_data_phase.sv:167	xbus_master_monitor.sv:167	xbus
xbus_master_driver.run.sv:56	xbus_master_driver.sv:56	
xbus_master_driver.reset_signals.sv:78	xbus_master_driver.sv:78	xbus
xbus_master_driver.read_byte.sv:139	xbus_master_driver.sv:139	xbus
xbus_bus_monitor.run.sv:197	xbus_bus_monitor.sv:197	
xbus_bus_monitor.collect_data_phase.sv:298	xbus_bus_monitor.sv:298	xbus
xbus_bus_monitor.observe_reset.sv:206	xbus_bus_monitor.sv:206	

Threads only mode allows visualizing call stack for selected thread

6-38

DVE Instance Object IDs vs. UVM Instance ID

- **VCS Object ID represents the Nth object instance of a class**

- Always refers to the same object in memory, regardless of handles
- Each class instance will have a unique object ID: <className>@<ID#>
- Object IDs are the foundation for dynamic debug
 - ◆ Dynamic waveforms, object browser, etc.

- **Use Object IDs for breakpointing**

- More reusable than a global_id
- Less affected by types moving in code

- **Easy comparison of object handles**

```
class Trans; ...
endclass

class Packet; ...
endclass

Trans tr;
Packet pkt;

initial begin
    tr = new(); //Trans@1
    tr = new(); //Trans@2
    pkt = new(); //Packet@1
    ...

```

- **UVM instance id (m_inst_id) is a unique value**

- VCS Object_id AND UVM m_inst_id are useful
- Both serve different purposes

6-39

Object IDs convey more meaning to the user (n'th instance of a class)

Fullname for object IDs is: <scope>:<className>@<ID#>

Class Instance Object IDs

An object ID always refer to the same object in memory

The screenshot shows two windows. On the left is the 'Object ID Value' window, which displays a hierarchical tree of variables and their types. A specific entry for 'req' is highlighted, showing its type as 'class simple_item'. On the right is a code editor window displaying SystemVerilog code for a 'simple_driver' class. In the code, there is a line of code: 'req_item_port.get_next_item(req);'. A callout box from the 'Object ID Value' window points to this line, indicating that the object ID '@2' refers to the same object in memory.

Object ID Value

variable	Value	Type
this	@2	class simple_driver
super		class uvm_driver #(test.s)
super		class uvm_component
req	@2	class simple_item
super		class uvm_sequence_item
addr	'hb	rand bit[3:0]
payload	size: 4	rand bit[7:0][0]
type_name	"simple_item"	string
valid		constraint
rsp	null	class simple_item
rsp_port	@10	class uvm_analysis_port
seq_item_p	@2	class uvm_seq_item_pull
seq_item_pr	@2	class uvm_seq_item_pull
type_name	"uvm_driver"	string
type_name	"simple_driver"	string
phase	@2 [uvm_driver #(REQ RSPI)]	package
uvm_pkg		package
UVM_UNBU	-1	int
ninth render	@1	class uvm front

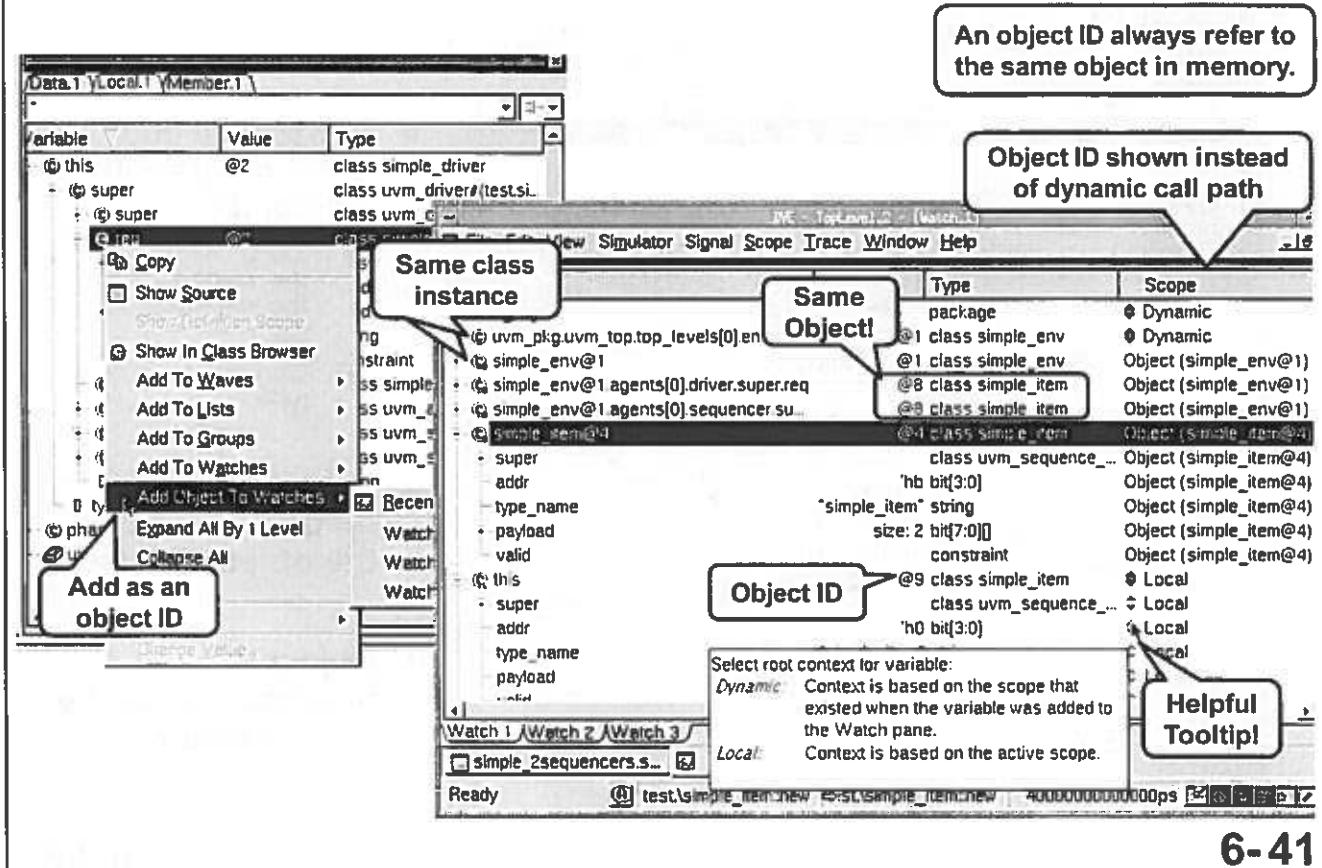
```
class simple_driver extends uvm_driver #(simple_item);
    `uvm_component_utils(simple_driver)
    function new(string name, uvm_component parent);
        super.new(name, parent);
        items = test.simple_items();
    endfunction

    virtual task main_phase(uvm_phase phase);
        forever begin
            seq_item_port.get_next_item(req);
            phase.raise_objection(this);
            @2
            send();
            seq_item_port.item_done();
            phase.drop_objection(this);
        end
    endtask

    virtual task send();
        // Implementation of send()
    endtask
endclass
```

6-40

Class Instance Object IDs



6-41

Drag & Drop to watch pane behavior still the same – adds Dynamic watchpoint.

Tooltip over “Dynamic” in scope will show dynamic stack path.

Every class instance has a unique ID number based on the order in which the object was created:

<classname>@<ID#>

For example, *simple_item@10* means the 10th *simple_item* object that was created.

Listing the above: *uvm_pkg.uvm_top.top_levels[0]* refers to the 1st instance of the class *simple_test*
simple_test@1 refers to the 1st instance of the class *simple_test* (the same as
uvm_pkg.uvm_top.top_levels[0])

simple_env@1 refers to the 1st instance of the class *simple_env*

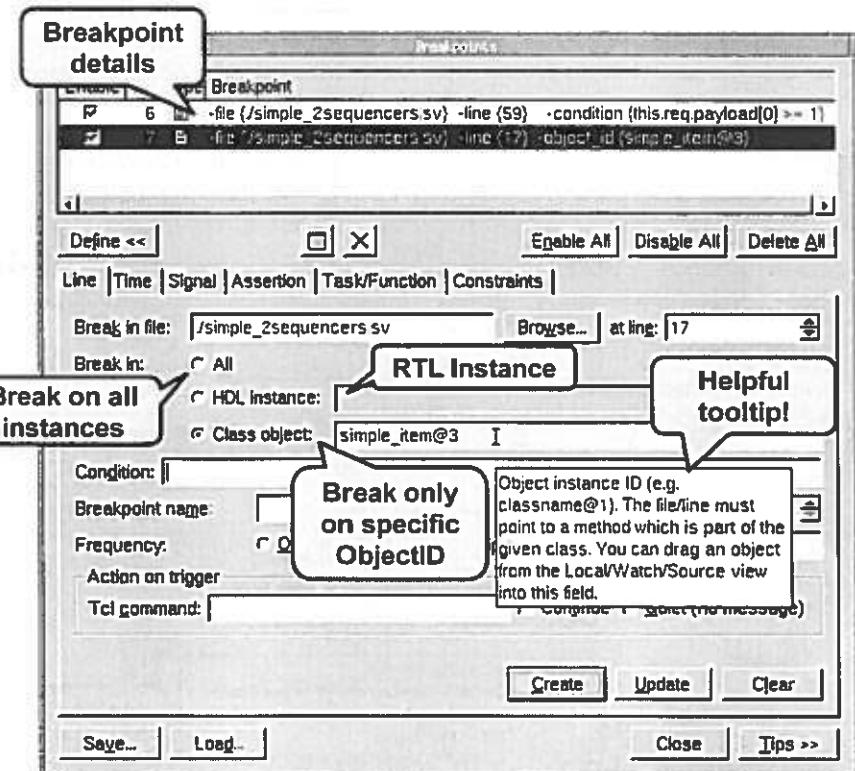
simple_env@1 refers to the 1st instance of the class *simple_env*

simple_item@3 refers to the 3rd instance of the class *simple_item*

this is local, referring to the this in the current local pane – currently the 2nd instance of the class *simple_driver*

this.super.req is a dynamic variable referring to the 6th instance of *simple_item*

Breakpoints – Object IDs



- **Customize line breakpoints with Object ID**

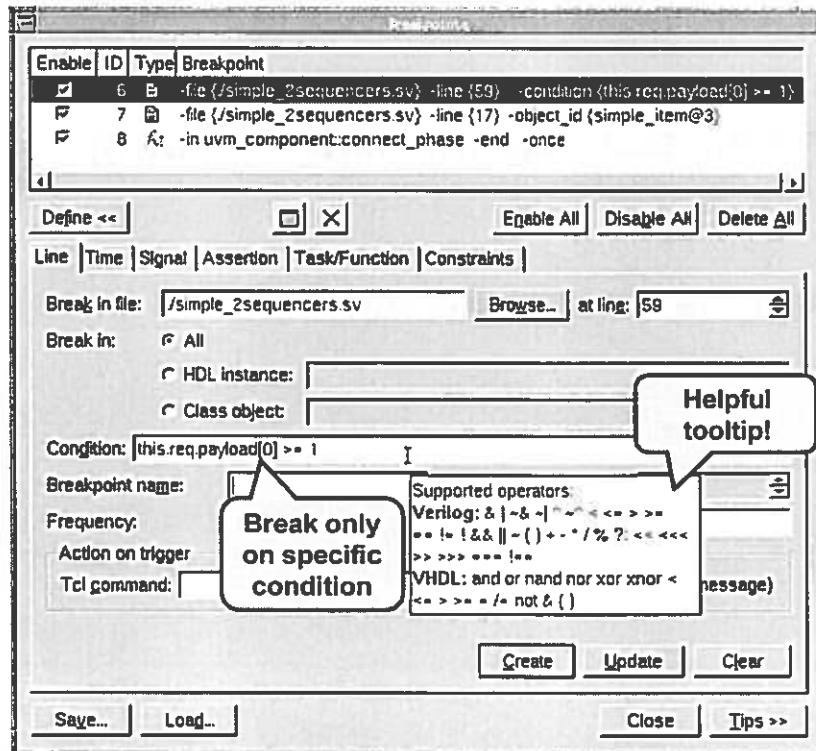
- Object ID type must be the same as the class the line breakpoint is set in
- Can also be a derived type

- **Create a breakpoint on object ID before the object exists!**

- **Object ID breakpoints can be saved/restored**

6-42

Breakpoints - Conditions



Setting the condition field

- Can reference objects using absolute or relative paths:
 - `uvm_pkg.uvm_test.top_level.s[0].env.agents[0].driver.req`
 - `this.req`
 - `req`
- Object IDs cannot be used in condition
- Condition must be valid from current scope
 - Variables in condition must be visible
 - Object must exist and reference must be valid
 - Trick to setting condition: Break in the scope, then set the condition
 - Or use an absolute path

6-43

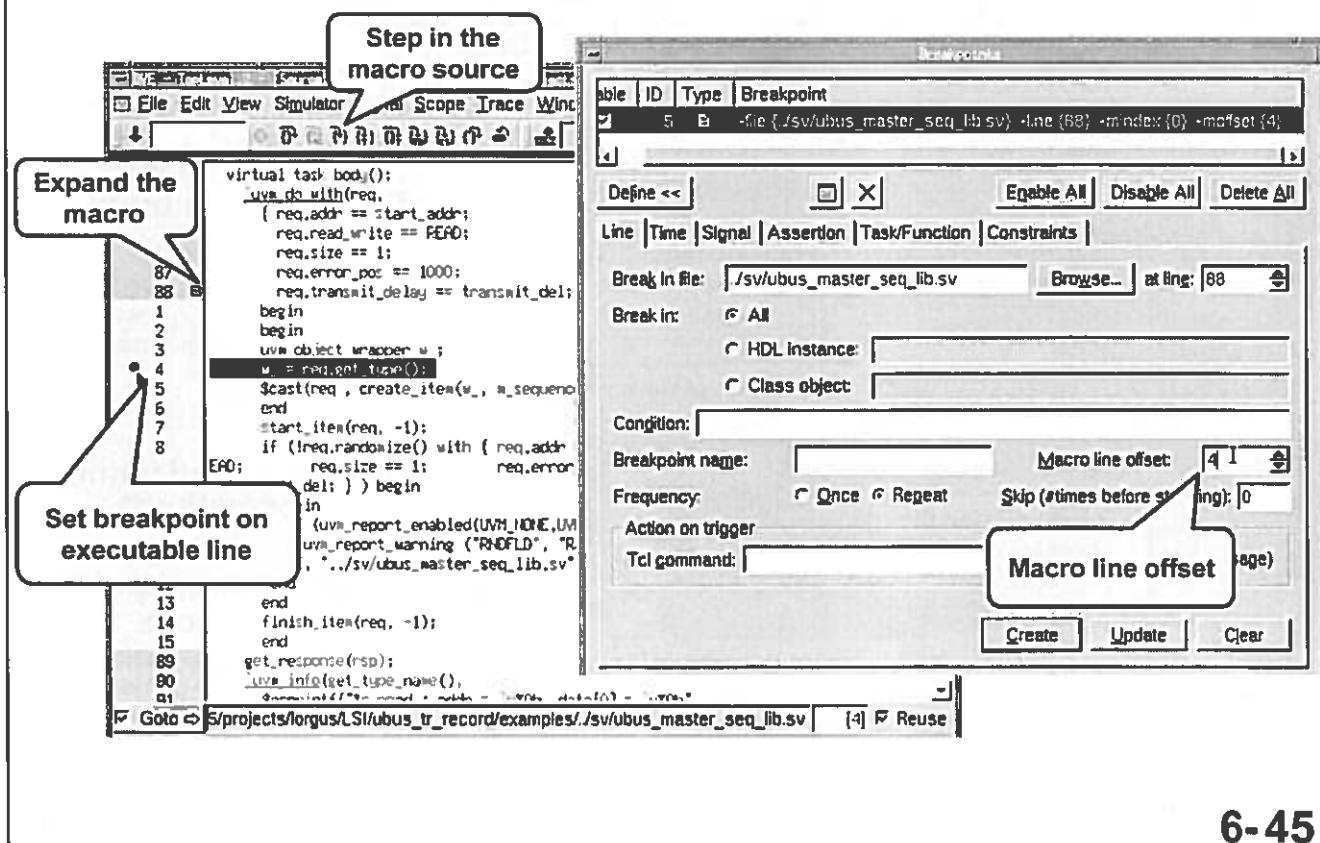
Per Instance Breakpoints –i.e m_name

```
stop -file ./my_file -line 23 -repeat -command {if {
[get m_name] != "uvm_test_top.env_i.a2_i"} {run;}}
```

- Also can be done in the breakpoint window

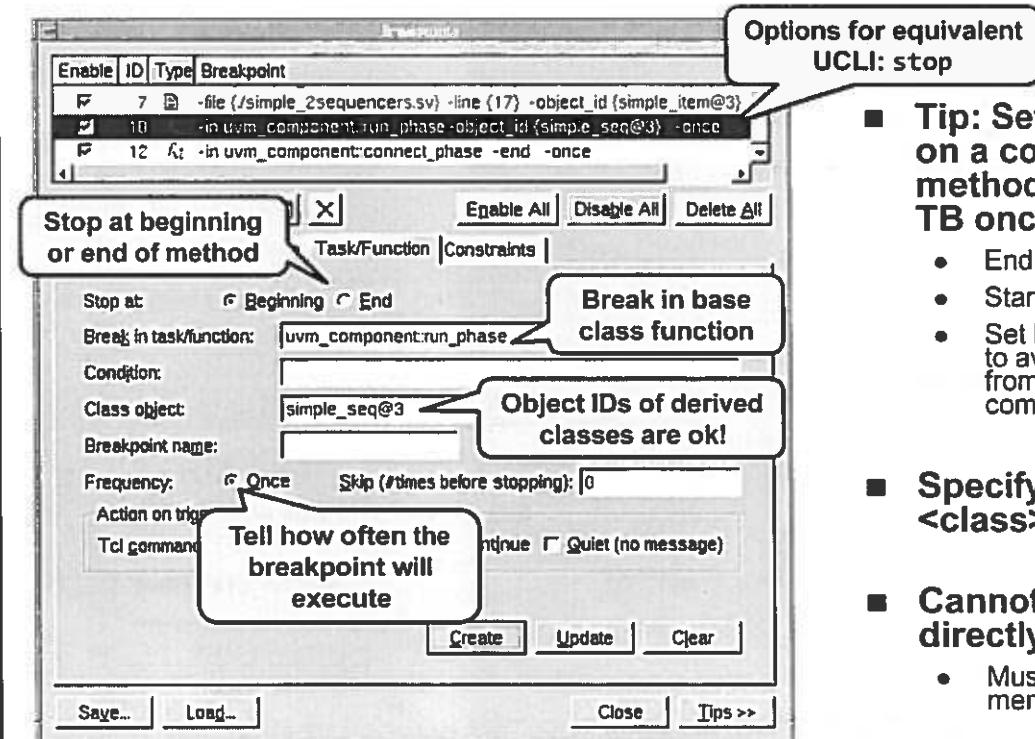
6-44

Macro Breakpoints



6-45

Breakpoints - Methods



- **Tip: Set a breakpoint on a common phase method to step into TB once env is build**
 - End of build_phase
 - Start of connect_phase
 - Set Frequency to "Once" to avoid multiple breaks from individual components
- **Specify method name: <class>::<method>**
- **Cannot set breakpoint directly in source**
 - Must use breakpoint menu or UCLI command

6-46

When using a standard methodology such as UVM or VMM, you can quickly get to a point where the entire testbench hierarchy is built and available by setting a breakpoint in a common phase. The trick is to use the frequency

Step into a Randomize Call

■ Entry mechanism:

- **step -solver**
- Randomize call executed atomically
- Triggers population of debug database in memory
- Execution stops before post-randomize() call
- Control transfers to Constraint Debug Mode
 - ◆ Constraint Dialog pops up



■ Exit mechanism: **next/continue**

- Execute randomize call but debug database not created

6-47

Static View: Constraints in Class Browser

Left Screenshot (Context Menu for c1):

- Show Source
- Show in Class Browser
- Automatic Relocation
- Show Definition Scope
- c1**
- Implication**
- Constraint Ordering**
- ForEach**
- Distribution**
- Constraint**
- Constr If**
- Regular Constraint**
- All constraints**
- Expand All**
- Collapse All**

Right Screenshot (Expanded View of c1):

Name	Type	Attribute
c1	Constraint	Class base
x dist {2 / 1, 13 / 99}	soft;	
((x + w) * v) < 16;		
(w > 8'h14)	soft;	

Properties of c1:

- Name:** x dist {2 / 1, 13 / 99}; soft;
- Type:** Distribution
- Lifetime:** Automatic
- From:** Constraint c2

6-48

Dynamic View: Local Pane

The screenshot shows the Dynamic View Local pane with two main windows. The left window displays the stack trace: 'Ver.1\Stack\Class.t' with scopes 'Scope', 'File', and 'Line'. The right window shows local variables for 'Data1\Local\Member.t' with columns 'Variable', 'Value', and 'Type'. A tooltip for 'bobimplicationsize' indicates it is a constraint of type 'rand int' with value 'super'. The code on the right is:

```
40 module ram();
41   int a;
42
43   d di=new();
44   initial begin
45     a = 2;
46     di.bobimplicationsize
47       constraint_mode(0);
48       di.randomize();
49
50     a = 3;
51     di.bobimplicationsize
52       constraint_mode(1);
53       di.randomize();
54   end
55 endmodule :ram
```

6-49

Solver/Relation Inside Constraint Dialog

Show Variables Related Through Constraints

The screenshot shows the Synopsys SystemVerilog IDE interface. On the left, the 'Solver/Relation' dialog is open, displaying variables and their initial values. A context menu is open over variable 'w'. The menu options include 'Show Source', 'Show In Class Browser', 'Show Relation', 'Expand All', and 'Collapse All'. On the right, the 'constraint.sv' file is shown with code related to constraints c1 and c2. Below it, another 'Solver/Relation' dialog shows variables 'x', 'w', 'v', 'y', and 'z' with their initial values. A context menu is open over variable 'v', listing 'Show Source', 'Show In Class Browser', 'Show In Solver', 'Expand All', and 'Collapse All'.

```
constraint c1 {
    x == 2 soft;
}
constraint c2 {
    x dist [2 : 1 , 13 : 99] soft;
    w + v + y < 16;
    w > 20 soft;
}
endclass: base

class item extends base;
constraint c1 {
    x == 4 soft;
    y == x;
}
endclass : item

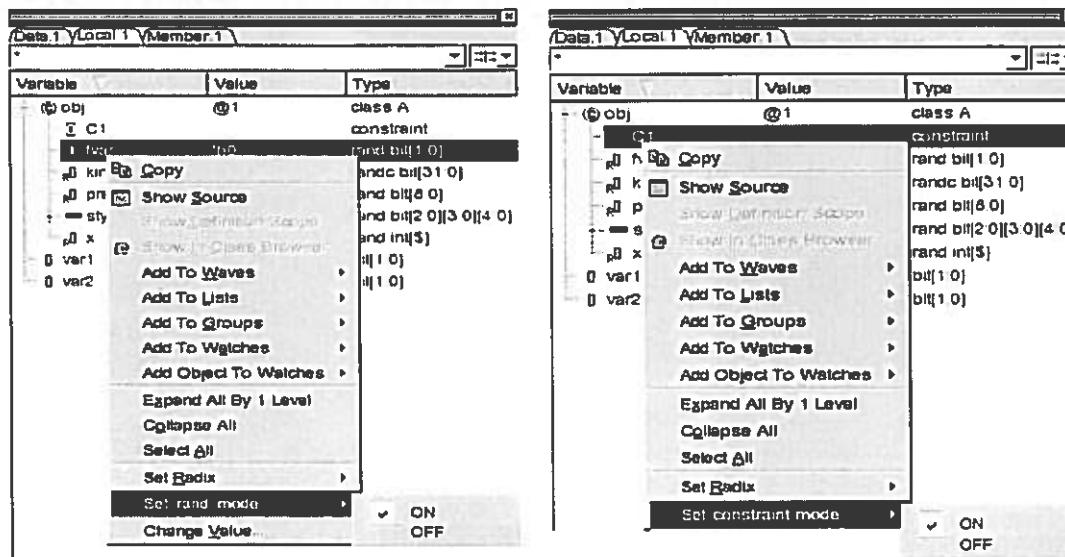
item item0;

initial begin
    item0 = new;
    repeat (10) begin
        item0.randomize();
    end
end
endmodule
```

6-50

Interactive Constraint Debug

- Modification of constraint_mode and rand_mode from within the local pane



6-51

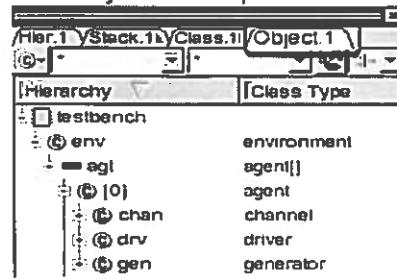
DVE UVM-Aware Debugging Features

UVM-Aware Features in DVE

Object Hierarchy Browser

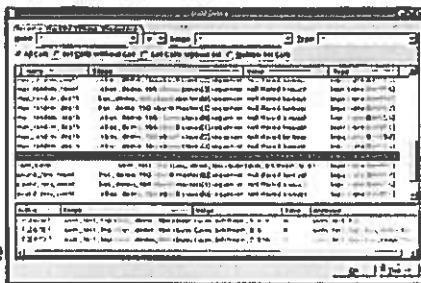
VMM-UVM-OVM Objects/Components

Hierarchy of
Classes and
Objects



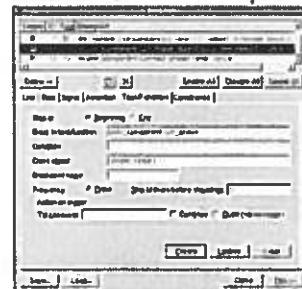
UVM Window

Resource
Factory
Phase
Sequence



Predefined Breakpoints

UVM Phase Tab and Breakpoint Dialog



Transaction Recording

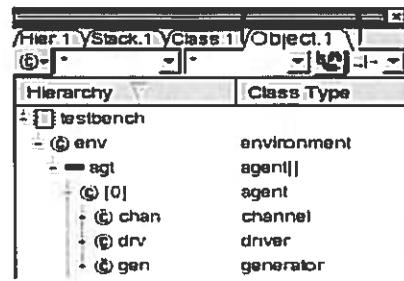


6-53

Object Hierarchy Browser

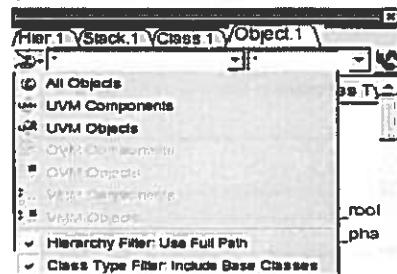
■ Display and navigate all dynamic variables

- Testbench hierarchy view
- Object creation time
- Object thread ids
- Object references
- Dynamic memory profiling
- Linked with Member Pane value annotation



■ General mode and methodology-aware mode

- All Objects
- UVM (Objects/Components)
- OVM (Objects/Components)
- VMM (Objects/Components)



6-54

Testbench Class Browser

Classes with Inheritance

The screenshot shows a class browser interface with two main panes. The left pane displays a tree view of UVM classes under 'User Classes'. The right pane shows the 'DEITAL (local) Member' list for a selected class, with annotations pointing to specific sections.

User Classes

- uvvm_void
- uvvm_object
- steve_address_map_info
- uvvm_callback
- uvvm_report_object
- (instances: 20)
- uvvm_component
- uvvm_agent
 - xbus_master_agent
 - xbus_slave_agent
- uvvm_driver#(bus_tb_top,x)
- abul_tb_top
- abus_master_driver
- abus_tb_top
- abus_slave_driver
- uvvm_env
- xbus_demo_tb
- abus_tb_top
- xbus_env
- uvvm_monitor
 - xbus_bus_monitor
 - (instances: 1)
 - bus_monitor
 - xbus_master_monitor
 - (instances: 2)
 - monitor
 - monitor
- uvvm_coreboard
- uvvm_sequencer_base
- uvvm_sequit
- abus_demo_base_test

Class Objects

Methods

- R__m_uvm_field_automati
- A assign_si
- A check_transfer_data_size
- A check_transfer_size
- A check_which_slave
- A collect_address_phase
- A collect_arbitration_phase
- A collect_data_phase
- A collect_transactions
- R getObjectType
- R get_type
- R get_type_name
- A new
- A observe_reset
- A perform_transfer_checks
- A perform_transfer_coverage
- A run
- A set_slave_configs

Variables

- B _addr
- B _check_enable
- B _cov_transaction
- B _cov_transaction_beat
- B _coverage_enable
- B _data
- B stem_collected_port
- B* m_registered_converter
- B _num_transactions

Methods with all Attributes

Annotations point to the 'Methods' and 'Variables' sections of the member list, indicating they are associated with inheritance.

```

endfunction : check_which_slave

// perform_transfer_checks
function void perform_transfer_checks();
    check_transfer_size();
    check_transfer_data_size();
endfunction : perform_transfer_checks

// check transfer size
function void check_transfer_size();
    if (trans_collected.read_write != NOP)
        assert_(transfer_size == assert(trans.collected.size == 0) || trans.collected.size == 8) else begin
            `uvm_error(get_type_name(),
                "invalid transfer size")
            begin
                if (uvvm_report_enabled(UVM_NONE))
                    uvvm_report_error (get_type_name(),
                        "Transfer size field / data size mismatch")
            end
        end
endfunction : check_transfer_size

// check transfer data size
function void check_transfer_data_size();
    if (trans_collected.size != trans.collected.size)
        `uvm_error(get_type_name(),
            "Transfer size field / data size mismatch")
endfunction : check_transfer_data_size

// perform_transfer_coverage
function void perform_transfer_coverage();
    if (trans_collected.read_write != NOP)
        begin
            cov_transaction;
            for (int unsigned i = 0; i < trans.collected.size(); i++)
                addr = trans.collected.addr + i;
                data = trans.collected.data(i);
        end
endfunction : perform_transfer_coverage

```

6-55

Testbench Object Variables

Explore all variables of current stack frame

'super' for base classes

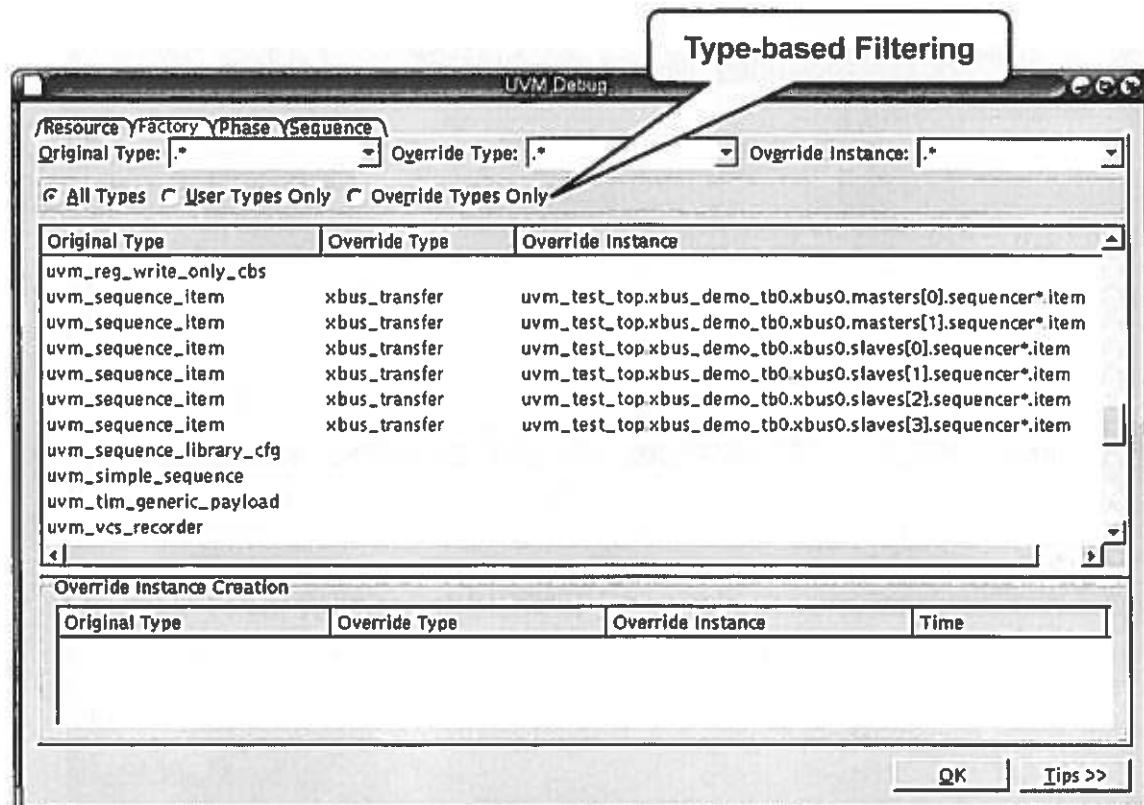
Expand objects with no limits

Object ID is key for identification

Variable	Value	Type
- @this	01	class xbui_bus_monitor
- @super		class uvm_monitor
- @super		class uvm_component
- @super		class uvm_report_object
- @super		class uvm_object
- @__m_uvm_status_container	01	class uvm_status_container
- @ m_inst_count	1115	int
- @ m_inst_id	578	int
- @ m_leaf_name	"bus_monitor"	string
- @ use_uvm_seeding	'h1	bit
- @ m_rh	075	class uvm_report_handler
- @ enable_stop_interrupt	0	int
- @event_pool	06	class uvm_object_string_pool#(uvm_pkg::uvm_event)
- @ m_build_done	'h1	bit
- @ m_children	size: 2	class uvm_component [uvm_port_component#(uvm_pkg::uv
- @ m_children_by_handle	size: 2	class uvm_component [uvm_port_component#(uvm_pkg::uv
- @ m_config_set	'h1	bit
- @ m_current_phase	025	class uvm_phase
- @ m_domain	02	class uvm_domain
- @super		class uvm_phase
- @super		class uvm_object
- @__m_uvm_status_container	01	class uvm_status_container
- @ m_inst_count	1115	int
- @ m_inst_id	170	int
- @ m_leaf_name	"uvm"	string
- @ use_uvm_seeding	'h1	bit
- @ m_end_node	011	class uvm_phase
- @ m_lmp	null	class uvm_phase
- @ m_jump_bkwd	'h0	bit

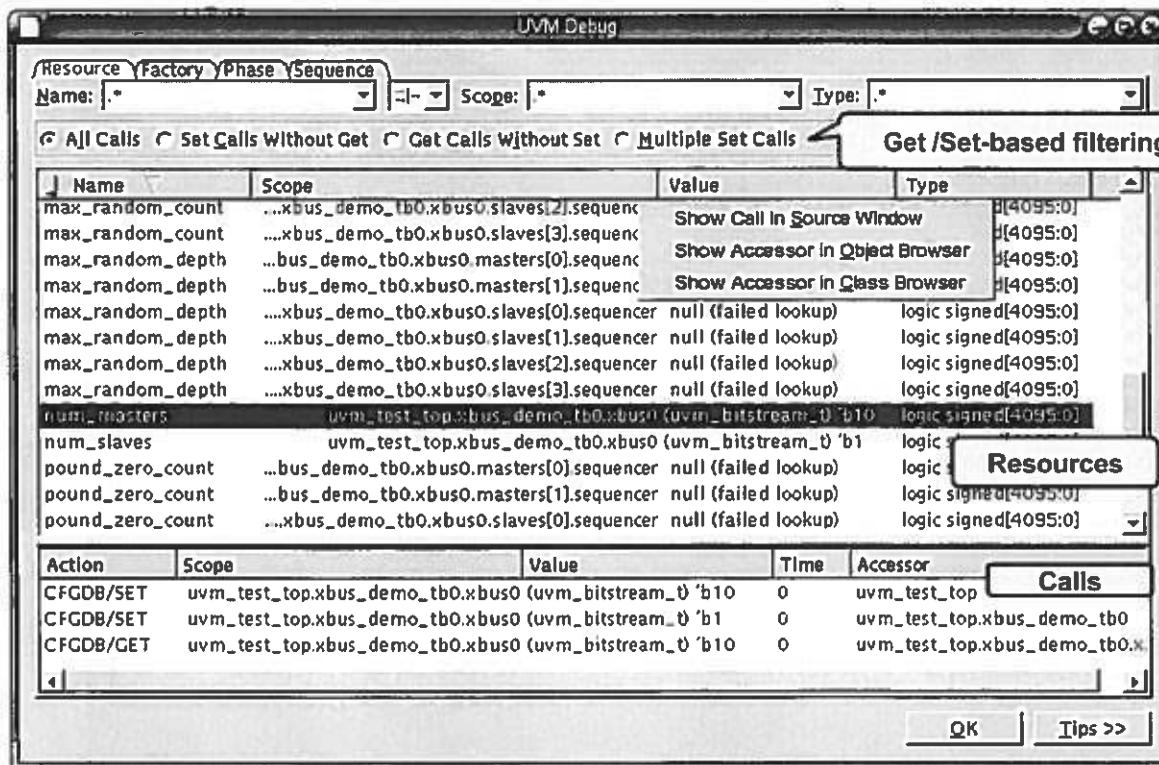
6-56

UVM Factory Debug



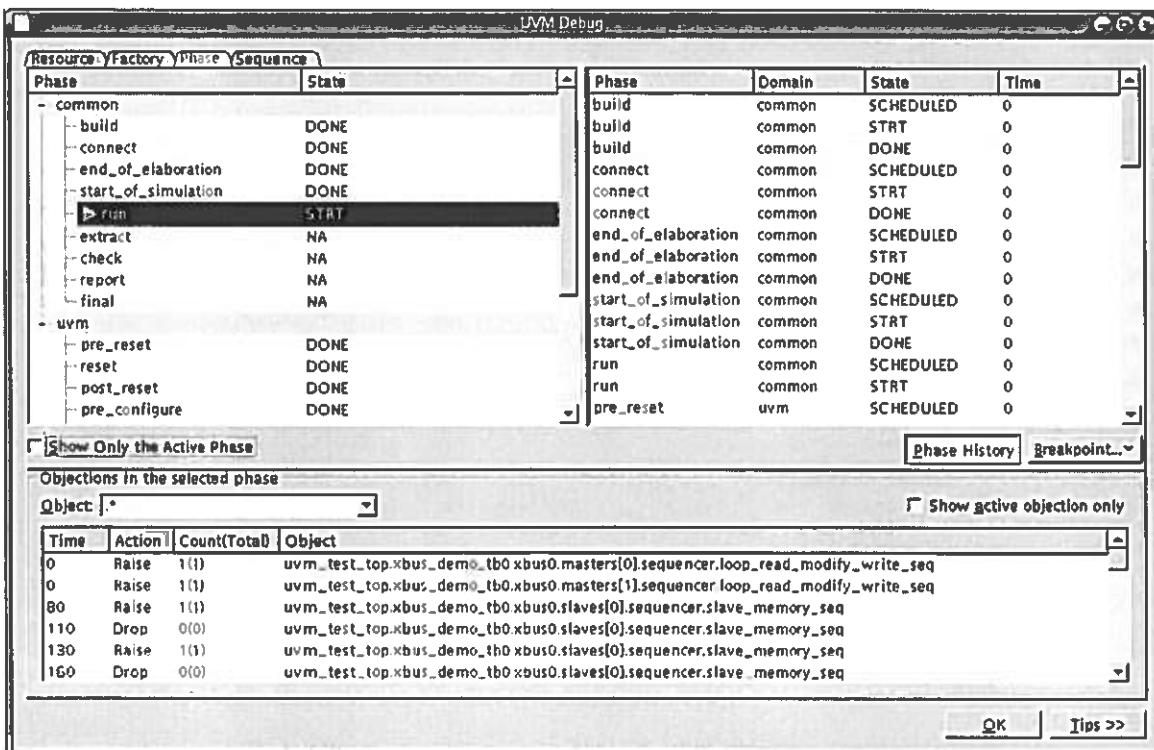
6-57

UVM Resource Debug



6-58

UVM Phase/Objection Debug



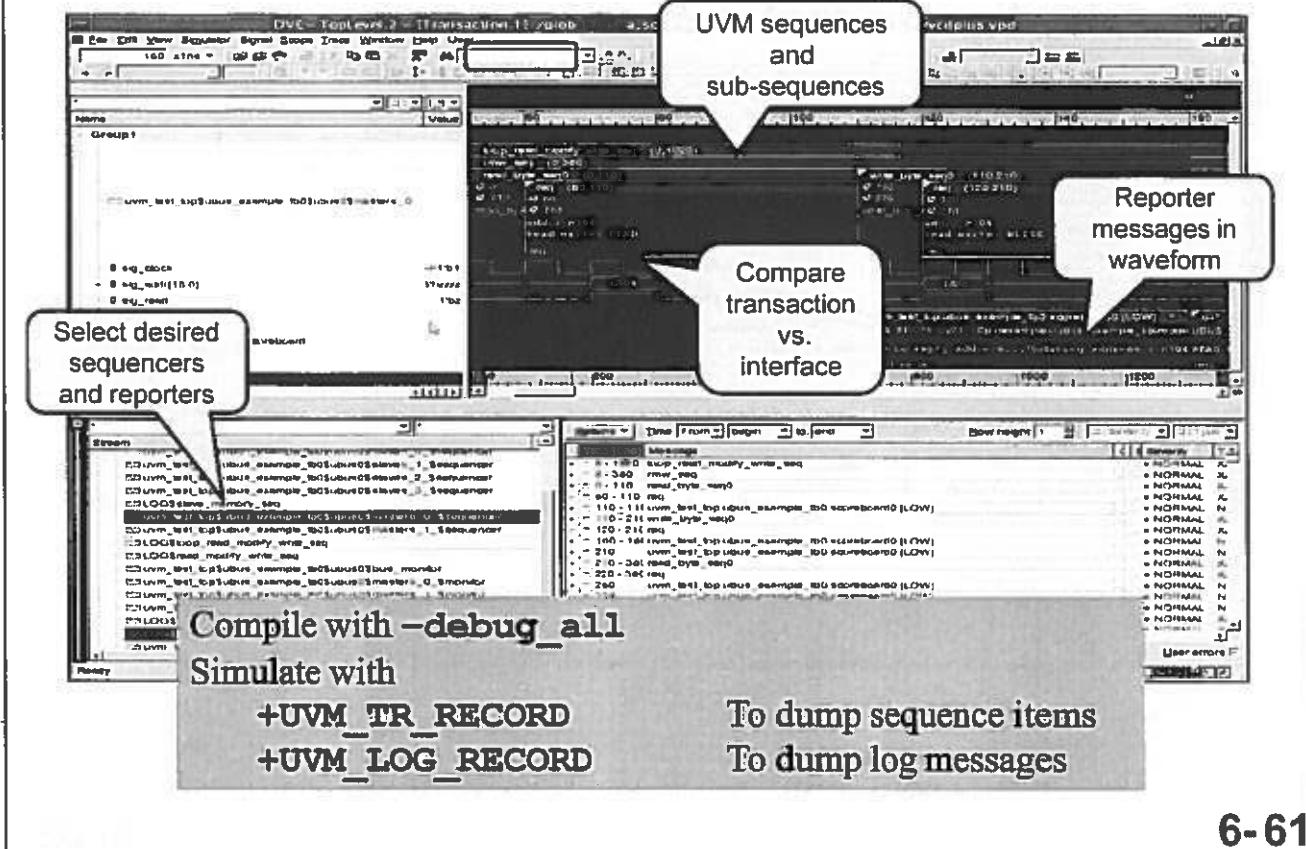
6-59

UVM Sequence Debug

Sequence	Sequence ID	Start	Finish	Phase	Thread	Sequencer	Sequencer ID
--loop_read_modify_write_seq	loop_read_modify_write_seq@1	0		run(common)	96	...masters[0].sequencer xbus_master_sequencer@1	
- rmw_seq	read_modify_write_seq@1	0			215	...masters[0].sequencer xbus_master_sequencer@1	
- - read_byte_seq0	read_byte_seq@1	0	110		240	...masters[0].sequencer xbus_master_sequencer@1	
- - - req	xbus_transfer@12	60	110		242	...masters[0].sequencer xbus_master_sequencer@1	
- - write_byte_seq0	write_byte_seq@1	110	210		240	...masters[0].sequencer xbus_master_sequencer@1	
- - - req	xbus_transfer@16	120	210		1955	...masters[0].sequencer xbus_master_sequencer@1	
- - read_byte_seq0	read_byte_seq@3	210			240	...masters[0].sequencer xbus_master_sequencer@1	
- - - req	xbus_transfer@20	220			1975	...masters[0].sequencer xbus_master_sequencer@1	
--loop_read_modify_write_seq	loop_read_modify_write_seq@2	0		run(common)	116	...masters[1].sequencer xbus_master_sequencer@2	
- rmw_seq	read_modify_write_seq@2	0			219	...masters[1].sequencer xbus_master_sequencer@2	
- - read_byte_seq0	read_byte_seq@2	0	160		241	...masters[1].sequencer xbus_master_sequencer@2	
- - - req	xbus_transfer@13	60	160		243	...masters[1].sequencer xbus_master_sequencer@2	
- - write_byte_seq0	write_byte_seq@2	160	260		241	...masters[1].sequencer xbus_master_sequencer@2	
- - - req	xbus_transfer@18	170	260		1965	...masters[1].sequencer xbus_master_sequencer@2	
- - read_byte_seq0	read_byte_seq@4	260			241	...masters[1].sequencer xbus_master_sequencer@2	
- - - req	xbus_transfer@22	270			1985	...masters[1].sequencer xbus_master_sequencer@2	
- - slave_memory_seq	slave_memory_seq@1	0		run(common)	138	...s0.slaves[0].sequencer xbus_slave_sequencer@1	
- - slave_memory_seq	slave_memory_seq@2	0		run(common)	160	...s0.slaves[1].sequencer xbus_slave_sequencer@2	
- - - req	xbus_transfer@9				227	...s0.slaves[1].sequencer xbus_slave_sequencer@2	
- - slave_memory_seq	slave_memory_seq@3	0		run(common)	182	...s0.slaves[2].sequencer xbus_slave_sequencer@3	
- - - req	xbus_transfer@10				231	...s0.slaves[2].sequencer xbus_slave_sequencer@3	
- - slave_memory_seq	slave_memory_seq@4	0		run(common)	204	...s0.slaves[3].sequencer xbus_slave_sequencer@4	
- - - req	xbus_transfer@11				235	...s0.slaves[3].sequencer xbus_slave_sequencer@4	

6-60

UVM Transaction Debug



6-61

Smart Log

Quickly find the root cause of an error

Complete Simulation Output

Hyperlink to Source and Time

Filter by Type

Filter by Severity

Filter by ID

The screenshot displays the Smart Log interface with several filter panels and a main log window.

- Complete Simulation Output:** A large window showing UVM INFO messages from the simulation. One message is highlighted: "UVM_INFO xb.us_demo_scoreboard.sv(10) @ 110: uvm_test_top.xbus_demo_tb0.scoreboard0 [xbus_demo_scoreboard] READ to empty address... Updating address : 12 with data : fd".
- Hyperlink to Source and Time:** A callout pointing to the highlighted message in the log, indicating it links to source code and timestamp information.
- Filter by Type:** A panel with dropdowns for Type (All), Severity (All), and Code (All). It lists categories like UVM-INFO, UVM-CRITICAL, UVM-ERROR, UVM-WARNING, UVM-FATAL, UVM-UNKNOWN, and UVM-NO_TYPE.
- Filter by Severity:** A panel with dropdowns for Severity (All), Code (All), and Type (All). It lists levels: Info (54), Warning (50), Error (45), Fatal (2), and No Severity.
- Filter by ID:** A panel with dropdowns for Code (All), Type (All), and Severity (All). It lists specific IDs: RHTST (2), loop_read_modify_write_seq (1), read_modify_write_seq (1), slave_memory_eq (8), test_2m_4s (2), xbus_bus_monitor (18), and xbus_demo_scoreboard (18).
- Main Log Window:** Shows a list of UVM INFO messages. The highlighted message is:


```
UVM_INFO xb.us_demo_scoreboard.sv(10) @ 110: uvm_test_top.xbus_demo_tb0.scoreboard0 [xbus_demo_scoreboard] READ to empty address... Updating address : 12 with data : fd
UVM_INFO xb.us_demo_scoreboard.sv(22) @ 160: uvm_test_top.xbus_demo_tb0.scoreboard0 [xbus_demo_scoreboard] READ to existing address... Checking address : 12 with data : fd
UVM_INFO xb.us_demo_scoreboard.sv(22) @ 210: uvm_test_top.xbus_demo_tb0.scoreboard0 [xbus_demo_scoreboard] WRITE to existing address... Updating address : 12 with data : fe
UVM_INFO xb.us_demo_scoreboard.sv(22) @ 260: uvm_test_top.xbus_demo_tb0.scoreboard0 [xbus_demo_scoreboard] WRITE to existing address... Updating address : 12 with data : fe
UVM_INFO xb.us_demo_scoreboard.sv(22) @ 310: uvm_test_top.xbus_demo_tb0.scoreboard0 [xbus_demo_scoreboard] READ to existing address... Checking address : 12 with data : fe
UVM_INFO xb.us_demo_scoreboard.sv(22) @ 360: uvm_test_top.xbus_demo_tb0.scoreboard0 [xbus_demo_scoreboard] READ to existing address... Checking address : 12 with data : fe
UVM_INFO xb.us_demo_scoreboard.sv(22) @ 410: uvm_test_top.xbus_demo_tb0.scoreboard0 [xbus_demo_scoreboard] READ to existing address... Checking address : 12 with data : fe
UVM_INFO xb.us_demo_scoreboard.sv(22) @ 460: uvm_test_top.xbus_demo_tb0.scoreboard0 [xbus_demo_scoreboard] READ to existing address... Checking address : 12 with data : fe
UVM_INFO xb.us_demo_scoreboard.sv(22) @ 510: uvm_test_top.xbus_demo_tb0.scoreboard0 [xbus_demo_scoreboard] READ to existing address... Checking address : 12 with data : fe
UVM_INFO xb.us_demo_scoreboard.sv(22) @ 560: uvm_test_top.xbus_demo_tb0.scoreboard0 [xbus_demo_scoreboard] READ to existing address... Checking address : 12 with data : fe
UVM_INFO xb.us_demo_scoreboard.sv(22) @ 610: uvm_test_top.xbus_demo_tb0.scoreboard0 [xbus_demo_scoreboard] WRITE to existing address... Updating address : 12 with data : ff
UVM_INFO xb.us_demo_scoreboard.sv(22) @ 660: uvm_test_top.xbus_demo_tb0.scoreboard0 [xbus_demo_scoreboard] WRITE to existing address... Updating address : 12 with data : ff
UVM_INFO xb.us_demo_scoreboard.sv(22) @ 710: uvm_test_top.xbus_demo_tb0.scoreboard0 [xbus_demo_scoreboard] READ to existing address... Checking address : 12 with data : ff
UVM_INFO xb.us_demo_scoreboard.sv(22) @ 760: uvm_test_top.xbus_demo_tb0.scoreboard0 [xbus_demo_scoreboard] READ to existing address... Checking address : 12 with data : ff
UVM_INFO xb.us_demo_scoreboard.sv(22) @ 810: uvm_test_top.xbus_demo_tb0.scoreboard0 [xbus_demo_scoreboard] WRITE to existing address... Updating address : 12 with data : 0
```

6-62

Agenda: Day 2

**DAY
2**

5 Component Configuration & Factory

6 Component Communication

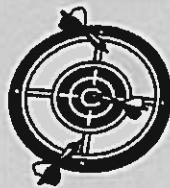


7 Scoreboard & Coverage

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:

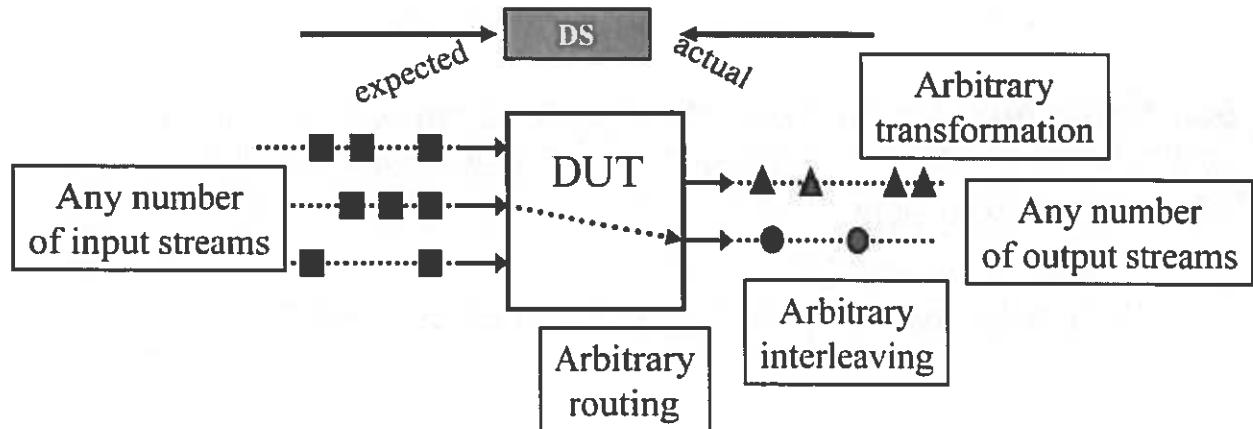
- In-order expects
- Data transformation
- Built in Analysis Exports in the Comparator classes

7-3

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

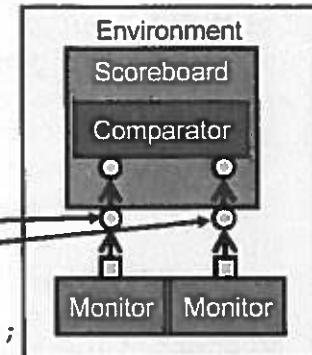


7-4

Scoreboard Implementation

■ Use `uvm_in_order_class_comparator` for checking

```
class scoreboard extends uvm_scoreboard; // utils macro and constructor
  typedef uvm_in_order_class_comparator #(packet) cmpr_type;
  cmpr_type cmpr;
  uvm_analysis_export #(packet) before_export;
  uvm_analysis_export #(packet) after_export;
  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    cmpr = cmpr_type::type_id::create("cmpr", 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(cmpr.before_export);
    after_export.connect(cmpr.after_export);
  endfunction
  virtual function void report_phase(uvm_phase phase);
    `uvm_info("Scoreboard Report",
      $sformatf("Matches = %0d, Mismatches = %0d",
        cmpr.m_matches, cmpr.m_mismatches), UVM_MEDIUM);
  endfunction
endclass
```



7-5

The example above omitted the following declaration, utility macros, and constructor:

```
'uvm_component_utils(scoreboard)

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

Scoreboarding: Monitor

- Monitors supplies scoreboard with expected and actual transactions

```
class iMonitor extends uvm_monitor;
    virtual router_io#(packet) sigs;
    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.get_parent(), "", "router_io", sigs))
            `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. It consists of an Environment block containing a Scoreboard and a Comparator. A Monitor block is connected to the scoreboard via an analysis port. The monitor also has a DUT interface and provides observed transactions to the collector via a TLM analysis port.

Annotations in the code:

- Embed analysis port: Points to the line `uvm_analysis_port #(packet) analysis_port;`
- Get DUT interface: Points to the line `virtual router_io#(packet) sigs;`
- Pass observed transaction to collector components via TLM analysis port: Points to the line `analysis_port.write(tr);`

7-6

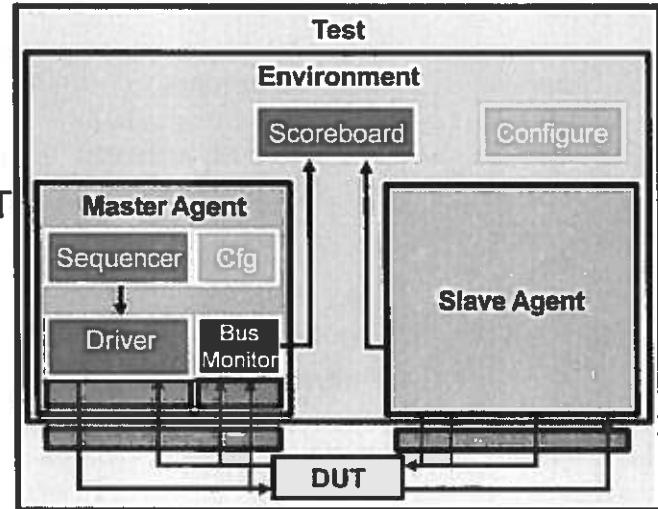
The example above omitted the following declaration, utility macros, and constructor:

```
'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 interfaces with DUT
 - ◆ Instantiates a driver, sequencer and monitor
 - Passive:
 - ◆ Operates passively
 - ◆ Only monitor instantiated and configured



7-7

An agent is group of UVM components to implement a protocol, such as AHB, PCI, etc.

An agent contains:

A Sequencer to generate a stream of constrained random transactions.

A Driver that gets transactions from the sequencer and drives them into the DUT

A Monitor that watches the DUT and creates a stream of transactions

A Configuration that holds agent-specific components

UVM Agent Example

```
class master_agent extends uvm_agent;
    uvm_analysis_port #(packet) analysis_port;
    // utils macro and constructor not shown
    sequencer seqr;
    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); is_active flag is built-in
        if (is_active == UVM_ACTIVE) begin
            seqr = packet_sequencer::type_id::create("seqr",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); Pass-through
        if(is_active == UVM_ACTIVE)
            drv.seq_item_port.connect(seqr.seq_item_export);
    endfunction: connect_phase
endclass
```

Create sequencer and driver if active

Connect sequencer to driver

7-8

The example above omitted the following constructor and utility macros:

```
'uvm_component_utils(master_agent)
```

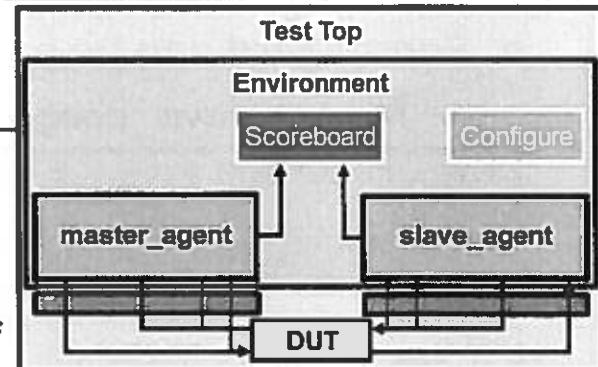
```
function new(input string name, uvm_component parent);
    super.new(name, parent);
endfunction
```

Using UVM Agent in Environment

■ Agent simplifies environment

- Easier to maintain and debug

```
class router_env extends uvm_env;
    master_agent m_agent;
    slave_agent s_agent;
    scoreboard sb;
    // utils and constructor not shown
    virtual function void build_phase(...);
        super.build_phase(phase);
        m_agent = master_agent::type_id::create("m_agent", this);
        s_agent = slave_agent::type_id::create("s_agent", this);
        sb = scoreboard::type_id::create("sb", this);
        uvm_config_db#(uvm_active_passive_enum)::set(this, "m_agent",
                                                       "is_active", UVM_ACTIVE);
        uvm_config_db#(uvm_active_passive_enum)::set(this, "s_agent",
                                                       "is_active", UVM_ACTIVE);
    endfunction
    virtual function void connect_phase(uvm_phase phase);
        m_agent.analysis_port.connect(sb.before_export);
        s_agent.analysis_port.connect(sb.after_export);
    endfunction
endclass
```



7-9

The example above omitted the following constructor:

```
'uvm_component_utils(router_env)

function new(input string name, uvm_component parent);
    super.new(name, parent);
endfunction
```

Parameterized Scoreboard

■ Scoreboard can be parameterized

- Must use `uvm_component_param_utils` macro

```
class scoreboard #(type T = packet) extends uvm_scoreboard;  
  ...  
  uvm_in_order_class_comparator #(T) comparator;  
  `uvm_component_param_utils_begin(scoreboard #(T))  
    `uvm_field_object(comparator, UVM_PRINT | UVM_COPY)  
  `uvm_component_utils_end  
  ...  
endclass
```

Macro end stays the same

■ In environment, set parameter or use default

```
class router_env extends uvm_env;  
  typedef scoreboard #(packet) pkt_scoreboard;  
  pkt_scoreboard sb;  
  function void build_phase(uvm_phase phase); ...  
    sb = pkt_scoreboard::type_id::create("sb", this);  
  ...  
endfunction  
endclass
```

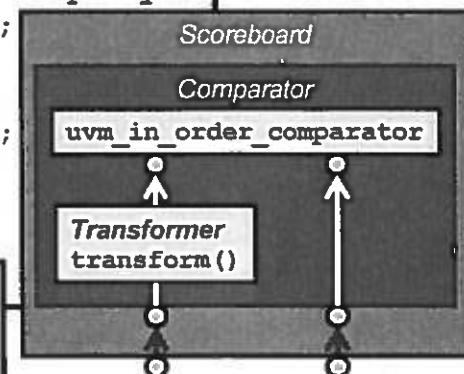
7-10

Scoreboard: Transformed Transaction

■ If transformation is required

```
class scoreboard extends uvm_scoreboard; ...
  uvm_analysis_export #(packet) before_export;
  uvm_analysis_export #(output_type) after_export;
  typedef uvm_algorithmic_comparator
    #(packet, output_type, transformer) cmpr_type;
  cmpr_type cmpr; transformer trns;
  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    trns = transformer::type_id::create("trns", this);
    cmpr = new("cmpr", this, trns); // cannot use proxy
    before_export = new("before_export", this);
    after_export = new("after_export", this);
  endfunction
  function void connect_phase(uvm_phase phase);
    before_export.connect(cmpr.before_export);
    after_export.connect(cmpr.after_export);
  endfunction ...
class transformer extends uvm_object; ...
  function output_type transform (packet tr);
    // perform transform
  endfunction
endclass
```

Algorithmic
comparator is
not UVM
compliant



7-11

Scoreboard: Out-Of-Order

- User can implement out-of-order scoreboard
 - Need a transaction queue to store in-coming transactions

```
'uvm_analysis_imp_decl(_before)
`uvm_analysis_imp_decl(_after)
class scoreboard #(type T = packet) extends uvm_scoreboard;
  typedef scoreboard #(T) this_type;
  `uvm_component_param_utils(this_type)
  uvm_analysis_imp_before #(T, this_type) before_export;
  uvm_analysis_imp_after #(T, this_type) after_export;
  int m_matches = 0, m_mismatches = 0, m_orphaned = 0;
  T pkt_list[$];
  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfunction
  virtual function void build_phase(uvm_phase phase);
    super.build_phase();
    before_export = new("before_export", this);
    after_export = new("after_export", this);
  endfunction
// continued on next slide
```

7-12

Scoreboard: Out-Of-Order

- Narrow down potential matches with a tag search
 - Otherwise, performance may be an issue

```
// continued from previous slide
virtual function void write_before(T pkt);
    pkt_list.push_back(pkt);
endfunction
virtual function void write_after(T pkt);
    int index[$];
    index = pkt_list.find_index() with (item.da == pkt.da);
    foreach(index[i]) begin
        if (pkt.compare(pkt_list[index[i]])) begin
            `uvm_info("Packet Match", pkt.get_name(), UVM_MEDIUM);
            m_matches++;
            pkt_list.delete(index[i]);
            return;
        end
    end
    `uvm_warning("Packet Not Found", {"\n", pkt.sprint()});
    m_orphaned++;
endtask
endclass
```

Do a quick search first

Do full compare on results of quick search

7-13

The heart of this scoreboard is the queue of packets, *pkt_list* and the SystemVerilog search function *find_first()* that searches for an element that satisfies the search expression. This function returns a queue with either a single entry that is the index into *pkt_list* 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 the index queue size is 0, no matching packet was found, so tell the user.

In this example, a mismatch only generates a warning. Your testbench may want to generate error messages instead.

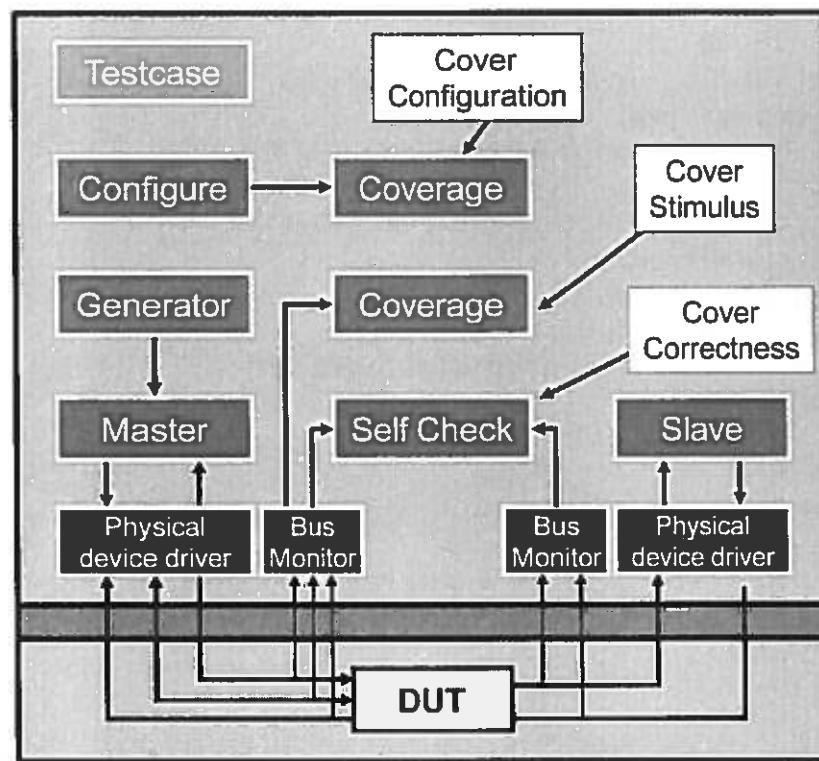
Functional Coverage

- **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

7-14

Connecting Coverage to Testbench

■ SystemVerilog Testbench Structure



7-15

Configuration Coverage

```
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;
    `uvm_component_utils_begin(config_coverage)
        `uvm_field_object(cfg, UVM_DEFAULT)
        `uvm_field_int(coverage_enable, UVM_DEFAULT)
    `uvm_component_utils_end
    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
            if (!uvm_config_db #(router_cfg)::get(this, "", "cfg", cfg) begin
                `uvm_fatal(...);
            end
            cg = new();
        end
    endfunction
    virtual function void start_of_simulation_phase(uvm_phase phase);
        if (coverage_enable)
            cg.sample(cfg);
    endfunction
endclass
```

7-16

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.

Configuration Coverage

■ Build configuration coverage component in test

```
class test_ports extends test_base; // utils and constructor not shown
  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
```

7-17

Stimulus Coverage

■ Cover monitor's observed transactions

```
covergroup pkt_cg with function sample(packet pkt);
    coverpoint pkt.sa;
endgroup: pkt_cg
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 : write
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.agent.analysis_port.connect(cov_comp.analysis_export);
    endfunction
endclass
```

subscriber class with built-in analysis port

Sample method called with monitored packet

7-18

Below is the router environment class, stripped of everything but the code for the packet coverage and output agent.

```
class router_env extends uvm_env;
    output_agent o_agent[];
    packet_coverage pkt_cov;
    `uvm_component_utils_begin(router_env)
        `uvm_field_object(cfg, UVM_DEFAULT)
    `uvm_component_utils_end
    virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        o_agent = new[cfg.ports_in_use.size];
        foreach (o_agent[i])
            o_agent[i] = output_agent::type_id::create($sformatf("o_agent%0d", i), this);
        pkt_cov = packet_coverage::type_id::create("pkt_cov", this);
    endfunction
    virtual function void connect_phase(uvm_phase phase);
        foreach (o_agent[i]) begin
            o_agent[i].analysis_port.connect(sb.after_export);
            o_agent[i].analysis_port.connect(pkt_cov.cov_export);
        end
    endfunction
endclass
```

Correctness Coverage

■ Cover verified transaction in scoreboard

```
'uvm_analysis_imp_decl(_before)
`uvm_analysis_imp_decl(_after)          Packet cover group

covergroup sb_pkt_cg with function sample(packet pkt);
  coverpoint pkt.sa;
  coverpoint pkt.da;
  cross pkt.sa, pkt.da;
endgroup: sb_pkt_cg

class scoreboard #(type T = packet) extends uvm_scoreboard;
  bit coverage_enable = 0;
  // component_utils and other code not shown
  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: write_after
endclass
```

Packet cover group

TLM imp method called
with monitored packets

7-19

Unit Objectives Review

Having completed this unit, you should be able to:

- **Build re-usable self checking scoreboards by using the in-built UVM comparator classes**
- **Implement functional coverage**

7-20

Appendix

Multi-Stream Scoreboard

Multi-Stream Scoreboard

Scoreboard: Multi-Stream

```
`uvm_analysis_imp_decl(_before)
`uvm_analysis_imp_decl(_after)
class scoreboard #(type T = packet, int num=16) extends uvm_scoreboard;
  typedef scoreboard #(T, num) this_type;
  typedef uvm_in_order_class_comparator #(T) cmpr_type;
  uvm_analysis_imp_before #(T, this_type) before_export;
  uvm_analysis_imp_after #(T, this_type) after_export;
  cmpr_type comparator[num];
  `uvm_component_param_utils(this_type)
  function new (string name, uvm_component parent);
    super.new(name, parent);
  endfunction
  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    before_export = new("before_export", this);
    after_export = new("after_export", this);
    for (int i=0; i < num; i++) begin
      comparator[i] = cmpr_type::type_id::create($sformatf("cmpr_%0d",
i), this);
    end
  endfunction
... // Continued on next page
```

7-23

Scoreboard: Multi-Stream

```
virtual function void write_before(T pkt);
    comparator[pkt.da].before_export.write(pkt);
endfunction

virtual function void write_after(T pkt);
    comparator[pkt.da].after_export.write(pkt);
endfunction

virtual function void report();
    `uvm_info("TRACE", $sformatf("%m"), UVM_HIGH);
    foreach (comparator[i]) begin
        `uvm_info("Scoreboard Report",
            $sformatf("Comparator[%0d] Matches = %0d, Mismatches = %0d", i,
            comparator[i].m_matches, comparator[i].m_mismatches), UVM_MEDIUM);
    end
endfunction

endclass
```

7-24

Agenda: Day 2

**DAY
2**

5 Component Configuration & Factory

6 Component Communication

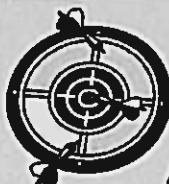


7 Scoreboard & Coverage

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

Implementing Simple Callback Operations

- Simple callback requires one to extend from existing component class

```
class new_driver extends driver;
    virtual task pre_send(...); ...
    virtual task send(...); ...
    virtual task post_send(...); ...
endclass
```

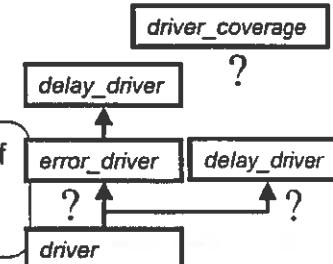
In the derived class, one can modify either the major operation or the callback methods

- Works very 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 error_driver extends driver;
class delay_driver extends driver;
class delay_driver extends error_driver;
```

What to extend from if different requirement for different test?



8-4

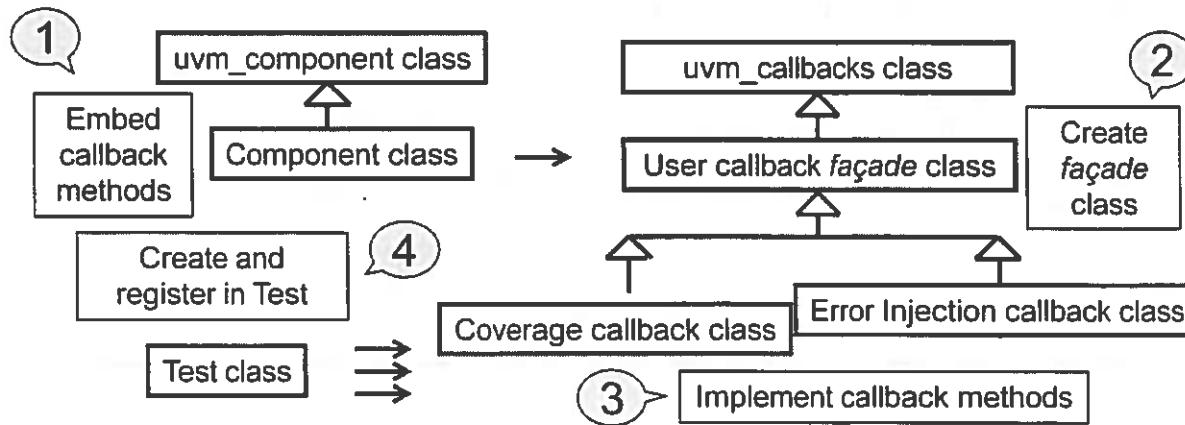
You can add new behavior to an existing component by extending the class. For example, *my_driver* can be extended to inject errors by extending the class to create *error_driver* that has a new *send()* task.

But what if you want to make further modifications, such as adding a delay to the transmission? If your new class, *delay_driver*, extends the base driver, you won't be able to write a test that injects both delays and errors. If instead your new class extends from the *error_driver* class, any delay tests will also have to set up or disable the error injection.

As you add more and more capabilities, such as extending the protocol, or inject multiple flavors of errors, your OOP hierarchy becomes unstable. UVM provides the 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 new capabilities, 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
```

Component class name

1a. Register UVM callback with component

1b. Embed UVM callback methods with `uvm_do_callbacks` macro

UVM callback class name
User must create (see next slide)

UVM callback method
User must embed in callback class (see next slide)

8-6

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;
    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

8-7

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;
    virtual task pre_send(driver drv, packet tr);
       drv.req.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 driver_err_test 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.drv, drv_err_cb);  
    uvm_callbacks #(driver, driver_callback)::display();  
  endfunction  
endclass
```

4b. Register callback objects

4c. Visually verify the registration (optional)

8-9

Driver Coverage Example

- If there are no analysis ports in driver
 - Callbacks can be the hooks for coverage also

```
typedef class driver;
class driver_callback extends uvm_callback;
// constructor not shown
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)
// 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
```

8-10

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();
        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 object in Environment
- Construct and register callback object in build 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.drv, drv_cov_cb);  
//  uvm_callbacks #(driver, driver_callback)::add(null, drv_cov_cb);  
endfunction  
endclass
```

4b. Register callback objects

Alternative: Register callback objects to all driver objects

8-12

User Callback Debug

- **Compile-time switch:**

- `+UVM_CB_TRACE_ON`

```
VCD+ Writer F-2011.12 Copyright (c) 1991-2011 by Synopsys Inc.  
UVM_INFO /global/apps5/vcs_2011.12/etc/uvm-  
1.1/base/uvm_callback.svh(631) @ 0: reporter [UVMCB_TRC] Add  
(UVM_APPEND) typewide callback uvm_report_catcher for type : callback  
uvm_report_catcher (uvm_callback@465)  
UVM_INFO @ 0.0ns: reporter [RNTST] Running test test_base...  
UVM_INFO reset_agent.sv(28) @ 0.0ns: uvm_test_top.env.r_agent [RSTCFG]  
Reset agent r_agent setting for is_active is: UVM_ACTIVE  
UVM_INFO /global/apps5/vcs_2011.12/etc/uvm-  
1.1/base/uvm_callback.svh(639) @ 0.0ns: reporter [UVMCB_TRC] Add  
(UVM_APPEND) callback sb_callback to object uvm_test_top.env.sb :  
callback sb_callback (uvm_callback@7788)
```

What was appended

Where the callback
object is appended

8-13

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 sequence::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



UVM 1.1 only

User should not call
these methods directly.
Instead, override in
sequence definition



UVM 1.1 only

8-14

Unit Objectives Review

Having completed 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

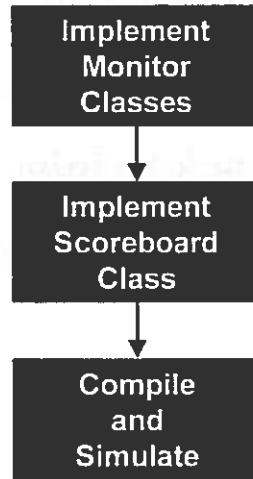
8-15

Lab 5 Introduction

Implement monitors and scoreboard



60 min



8-16

Agenda: Day 3

**DAY
3**

9 Virtual Sequence/Sequencer

10 Component Phasing Revisited



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 chosen phase with Virtual Sequences and Virtual Sequencers**
- **Manage synchronization of sequence execution within a virtual sequence**

9-2

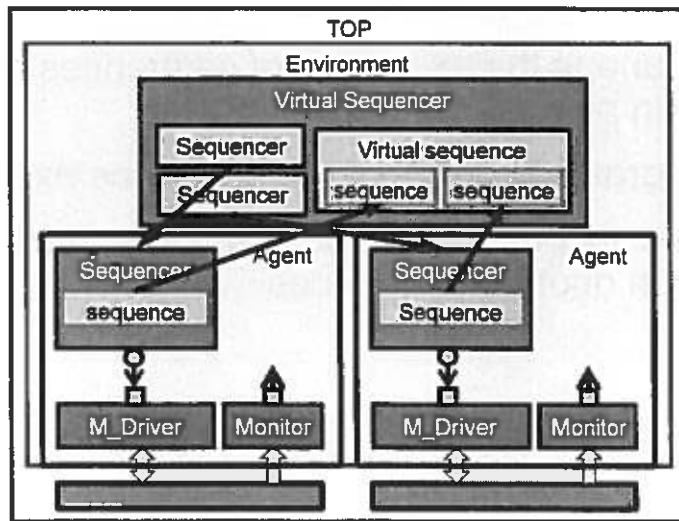
Virtual Sequences

- **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: Virtual Sequences**
 - 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

9-3

Virtual Sequence/Sequencer

- Typical sequence interacts with a single DUT interface
- When multiple DUT interfaces need to be synchronized a virtual sequence is required
- Virtual sequences in environment/test synchronize timing and transactions between different interfaces

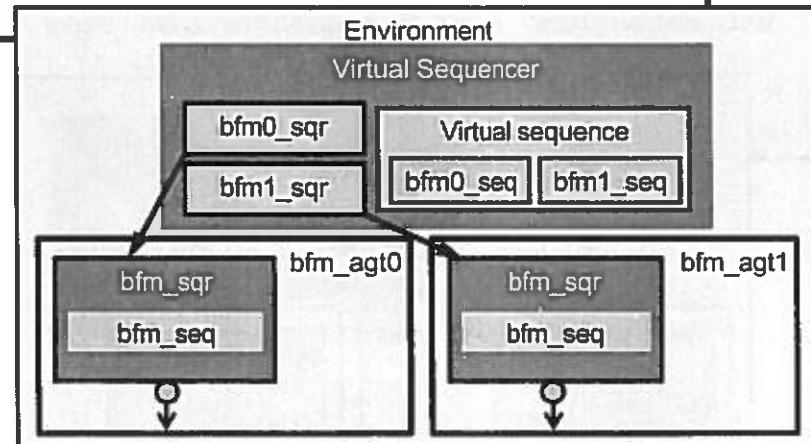


9-4

Virtual Sequencer

■ Embed sequencers managing the sequences

```
class virtual_sequencer extends uvm_sequencer; ...
  bfm_sequencer bfm0_sqr;
  bfm_sequencer bfm1_sqr;
  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfunction
endclass
```



9-5

The **virtual_sequencer** executes the **virtual_sequence** defined on the next slide.
It has handles to the sequencers in the two agents.

Virtual Sequence

- Embed sequence from different agents
- In body() method, control these sequences

```
class virtual_sequence extends uvm_sequence;
  `uvm_object_utils(virtual_sequence)
  `uvm_declare_p_sequencer(virtual_sequencer)

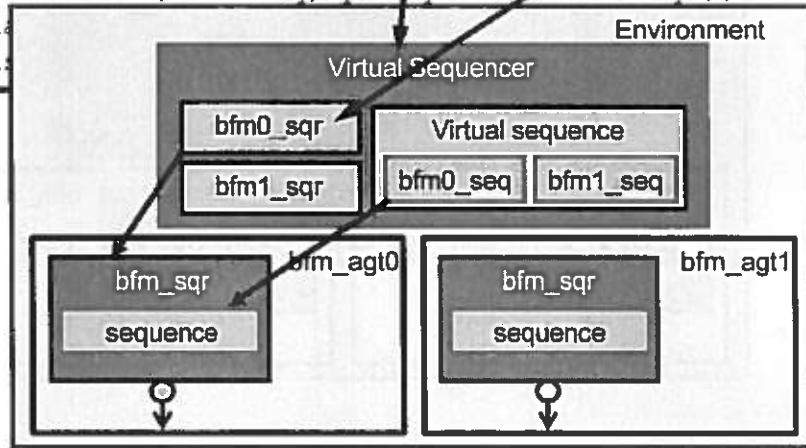
  bfm_sequence bfm0_seq;
  bfm_sequence bfm1_seq;

  virtual task body();
    `uvm_do_on(bfm0_seq, p_sequencer.bfm0_sqr);
    `uvm do on(bfm1 seq, p sequence.bfm1 sqr);

  endt
endclass
```

for accessing virtual sequencer content

can be completely different sequences for different interfaces



9-6

The following code is left off the slide:

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

Connect Sequencer to Virtual Sequencer

- Create the virtual sequencer in build phase
- Disable the controlled sequencers by setting their default_sequence to null
- Configure the virtual sequencer to execute the virtual sequence in the desired phase

```
class top_env extends uvm_env; // other code not shown
    virtual_sequencer v_sqr; // and subenv's
    virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        // creation of other objects not shown
        v_sqr = virtual_sequencer::type_id::create("v_sqr", this);
        uvm_config_db#(uvm_object_wrapper)::set(this,
            "bfm_agt0.sqr.main_phase", "default_sequence", null);
        uvm_config_db#(uvm_object_wrapper)::set(this,
            "bfm_agt1.sqr.main_phase", "default_sequence", null);
        uvm_config_db#(uvm_object_wrapper)::set(this, "v_sqr.main_phase",
            "default_sequence", virtual_sequence::get_type());
    endfunction
// continued on the next page
```

Disable child sequencers

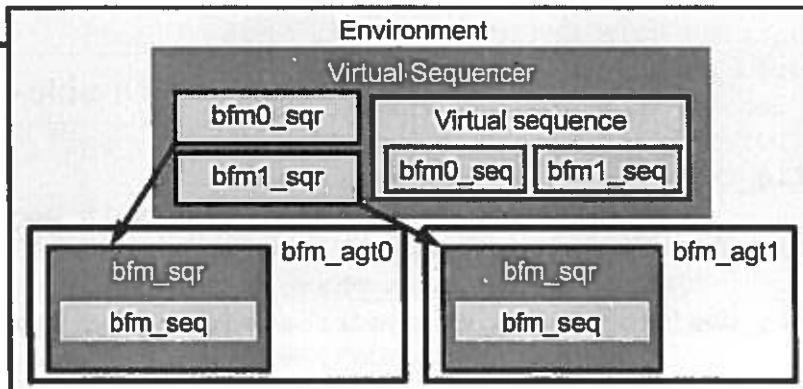
Set virtual sequencer to
execute virtual sequence

9-7

Connect Sequencer to Virtual Sequencer

- In connect_phase, set up the controlled sequencers for the virtual sequencer

```
// Continued from previous page
virtual function void connect_phase(uvm_phase phase);
    v_sqr.bfm0_sqr = bfm_agt0.bfm_sqr;
    v_sqr.bfm1_sqr = bfm_agt1.bfm_sqr
endfunction
endclass
```



9-8

In the connect phase, point the sequencer handles in the virtual sequencer to the actual location of the sequencers in the testbench.

Sequence Execution Management

What if sequence needs to start in middle of another sequence?

```
class virtual_sequence extends uvm_sequence; // other code not shown
virtual task body();
    fork
        `uvm_do_on(seq0, p_sequencer.bfm0_sqr);
    begin
        wait(...); // wait for some event before processing
        `uvm_do_on(seq1, p_sequencer.bfm1_sqr);
    end
    join
endtask
endclass
```

- Use **uvm_event** via **uvm_pool**

9-9

Synchronization Mechanism: uvm_event

■ Wait for one trigger to releases all waits

```
class uvm_event extends uvm_object; // Emulates Verilog event with more features
function new(string name="");
virtual function void trigger(uvm_object data=null); // data optional
virtual function void reset(bit wakeup=0); // Triggers uvm_event
virtual function bit is_on();
virtual function bit is_off(); // After trigger, uvm_event stays on until reset is called
virtual task wait_on(bit delta=0); // Query state of uvm_event
virtual task wait_off(bit delta=0); // Wait for state. Does not block if true.
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
virtual function time get_trigger_time();
virtual function int get_num_waiters();
endclass : uvm_event
```

9-10

Component Synchronization: uvm_barrier

- Wait for number of waiters to reach threshold to release all waits

```
class uvm_barrier extends uvm_object;  
function new (string name="", int threshold=0);  
virtual function void set_threshold (int threshold);
```

Sets threshold level (n number of waiters) to release all waits

```
virtual function int get_threshold ();  
virtual task wait_for();
```

Caller blocked until threshold of waiters has been reached. Increments waiter count.

```
virtual function int get_num_waiters ();  
virtual function void reset (bit wakeup=1);
```

Reset numbers of waiters to 0. If wakeup set, release all current waiters.

```
virtual function void set_auto_reset (bit value=1);
```

If value is 1, when threshold is hit, count resets to 0. If 0, count does not reset.

```
virtual function void cancel ();  
endclass
```

Cancels wait. Decrement waiter count.

9-11

Specialized Pools for Synchronization

- Simplify usage of `uvm_event` and `uvm_barrier` through two specialized resource pools available in UVM :

```
typedef uvm_object_string_pool #(uvm_barrier) uvm_barrier_pool;  
typedef uvm_object_string_pool #(uvm_event)    uvm_event_pool;
```

- Through the resource pools, events and barrier can be globally accessible
 - The most 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
```

get methods create keyed resource if it does not already exist

get methods create
keyed resource if it
does not already exist

9-12

uvm_event_pool Example (Trigger)

■ 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);
      detect(tr);
    end
  endtask: run_phase

  virtual task detect(reset_tr tr);
    @(sig.s.reset_n);
    if (sig.s.reset_n == 1'b0) begin
      tr.kind = reset_tr::ASSERT;
      reset_event.trigger(); —————— trigger "reset" uvm_event
    end else begin
      tr.kind = reset_tr::DEASSERT;
      reset_event.reset(); —————— clear "reset" uvm_event
    end
    analysis_port.write(tr);
  endtask: detect
endclass
```

Get "reset" uvm_event from uvm_event_pool

trigger "reset" uvm_event

clear "reset" uvm_event

9-13

uvm_event_pool Example (Wait for Trigger)

■ Globally accessible

```
class virtual_reset_sequence extends uvm_sequence; // other code left off
  `uvm_declare_p_sequencer(virtual_reset_sequencer)
  uvm_event reset_event = uvm_event_pool::get_global("reset");
  reset_sequence r_seq;  driver_reset_sequence d_seq;

  virtual task body();
    fork
      `uvm_do_on(r_seq, p_sequencer.r_seqr);
      foreach (p_sequencer(pkt_seqr[i]) begin
        int j = i;
        fork
          begin
            reset_event.wait_on(); // Wait for "reset" uvm_event to be on
            `uvm_do_on(d_seq, p_sequencer(pkt_seqr[j]));
          end
        join_none // Execute response to the reset event
      end
    join
  endtask
endclass
```

Annotations:

- Get "reset" uvm_event from uvm_event_pool
- Wait for "reset" uvm_event to be on
- Execute response to the reset event

9-14

Protecting Stimulus (Grab/Ungrab)

■ Sequence can reserve a sequencer for exclusive use

- Until explicitly released
- Other request for exclusive use or stimulus injection will be blocked

```
class simple_sequence extends uvm_sequence #(packet);
  // utils macro and constructor not shown
  virtual task body();
    grab();           Sequence requesting grab() reserves
    repeat(10) begin parent sequencer for exclusive use
      `uvm_info(get_name(), "In body()", UVM_HIGH)
      `uvm_do_with(req, {addr == 6; data == 6;})
    end
    ungrab();        ungrab() method releases grab for
  endtask          default/specified sequencer.
endclass
```

9-15

Unit Objectives Review

Having completed this unit, you should be able to:

- Control execution order of sequences within a chosen phase with Virtual Sequences and Virtual Sequencers
- Manage synchronization of sequence execution within a virtual sequence

9-16

Appendix

Resource Pool Sequence Library

Resource Pool

Resource Pool: uvm_pool

- **uvm_pool class can be used to create user db pools**

```
class uvm_pool #(type KEY=int, T=uvm_void) extends uvm_object;
  const static string type_name = "uvm_pool";
  typedef uvm_pool #(KEY,T) this_type;
  static protected this_type m_global_pool; Singleton global pool
  protected T pool[KEY];
  function new(string name=""); Local pool
  static function T get_global(KEY key);
```

Get type T item from pool via KEY.

If doesn't already exists, create and store a T item with default value in pool.

```
static function this_type get_global_pool();
virtual function T get(KEY key);
virtual function void add(KEY key, T item);
virtual function int num();
virtual function void delete(KEY key);
virtual function int exists(KEY key);
virtual function int first(ref KEY key);
virtual function int last(ref KEY key);
virtual function int next(ref KEY key);
virtual function int prev(ref KEY key);
endclass
```

Get singleton global pool

pool content
access methods

9-19

Sequence Library

Sequence Library

- When multiple sequences need to be executed by a sequencer, how does one manage these sequences?
 - e.g. Execute a directed sequence before other randomly chosen sequences
- Solution: Sequence Library
 - Helps to organize and manage the execution of multiple sequences
 - By default executes sequences in the library randomly
 - Allows user to implement user algorithm of picking which sequence out of the sequence library to execute

9-21

Building a Sequence Library Package (1/2)

To enhance reuseability, encapsulate sequence library classes in a package

```
package packet_seq_lib_pkg;
import uvm_pkg::*;
class packet extends uvm_sequence_item;
...
endclass
class packet_seq_lib_base extends uvm_sequence_library #(packet);
`uvm_object_utils(packet_seq_lib_base)
`uvm_sequence_library_utils(packet_seq_lib_base)
function new(string name = "packet_seq_lib");
    super.new(name);
    init_sequence_library();
endfunction
endclass
// continued on next slide
```

Similar to the idea behind test_base, create a base library class without any sequences.

Macro builds the infrastructure of the sequence library

Method populates library with registered sequences

9-22

```
// For packet
class packet extends uvm_sequence_item;
rand bit [3:0] sa, da;
rand bit [7:0] payload[$];
`uvm_object_utils_begin(packet)
`uvm_field_int(sa, UVM_ALL_ON)
`uvm_field_int(da, UVM_ALL_ON)
`uvm_field_queue_int(payload, UVM_ALL_ON)
`uvm_object_utils_end
constraint valid { payload.size inside {[1:20]}; }
function new(string name = "packet");
    super.new(name);
endfunction
endclass
```

Building a Sequence Library Package (2/2)

■ Add common sequences to package

- Leave sequence library base empty

```
// continued from previous slide
class packet_seq_base extends uvm_sequence #(packet);
    // utils macro and constructor not shown
    virtual task pre_start();
        if (get_parent_sequence() == null && starting_phase != null)
            starting_phase.raise_objection(this);
    endtask
    virtual task post_start();
        if (get_parent_sequence() == null && starting_phase != null)
            starting_phase.drop_objection(this);
    endtask
endclass
class scenario_seq extends packet_seq_base; ...
class noise_seq extends packet_seq_base; ...
class drop_seq extends packet_seq_base; ...
endpackage
```

9-23

Reference Sequence Library in Environment

- In the environment, make sequence library the default sequence for the sequencer

```
class router_env extends uvm_env;
    router_agent agent;
    // utils macro and constructor not shown

    function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        agent = router_agent::type_id::create("agent", this);
        uvm_config_db#(uvm_object_wrapper)::set(this, "agent.seqr.main_phase",
            "default_sequence", packet_seq_lib_base::get_type());
    endfunction
endclass
```

If the sequence library does not contain any sequence, nothing happens.

If the sequence library is populated with sequences, by default, 10 sequences will be randomly picked out of the library and executed.

9-24

```
// component_utils and constructor
`uvm_component_utils(router_env);
function new(string name, uvm_component parent);
    super.new(name, parent);
endfunction : new
```

Register and Execute Sequences

- Import library package in program block
- Register sequence(s) with sequence library in test
 - Use `add_typewide_sequence()` to register sequence in all instances of sequence library in the test

```
program automatic test;
import uvm_pkg::*; import packet_seq_lib_pkg::*;
// include other classes
class test_scenario extends test_base;
// component_utils and constructor not shown
virtual function void build_phase(uvm_phase phase);
super.build_phase(phase);
packet_seq_lib_base::add_typewide_sequence(scenario_seq::get_type());
uvm_config_db#(int)::set(this, "*.seqr", "item_count", 20);
endfunction
endclass
...
endprogram
```

Compile with vcs: (using UVM in VCS installation)
vcs -sverilog -ntb_opts uvm-1.1 `packet_seq_lib_pkg.sv` `test.sv`

Simulate with:
`simv +UVM_TESTNAME=test_scenario`

9-25

Customize Sequence Library Object

- Sequence library can be customized for each sequencer
 - Create sequence library object
 - Use `add_sequence()` to register sequence
 - Configure chosen sequencer to use sequence library object
 - Affects only the configured sequencer in test

```
class test_100_pkt extends test_base;
  // component_utils and constructor not shown
  packet_seq_lib_base p_lib;
  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    p_lib = packet_seq_lib_base::type_id::create("p_lib", this);
    p_lib.add_sequence(scenario_seq::get_type());
    uvm_config_db#(uvm_sequence_base)::set(this,
      "*.agent1.seqr.main_phase", "default_sequence", p_lib);
    uvm_config_db#(int)::set(this, "*.agent1.seqr", "item_count", 100);
  endfunction
endclass
```

9-26

Configuration Issues

- The following will compile but not recognized

```
p_lib = packet_seq_lib_base::type_id::create("p_lib", this);
uvm_config_db#(packet_seq_lib_base)::set(this,
    "*seqr.main_phase", "default_sequence", p_lib);
```

- When configuring sequence library object

- Type MUST be `uvm_sequence_base`

```
p_lib = packet_seq_lib_base::type_id::create("p_lib", this);
uvm_config_db#(uvm_sequence_base)::set(this,
    "*seqr.main_phase", "default_sequence", p_lib);
```

- The following will not compile

```
p_lib = packet_seq_lib_base::type_id::create("p_lib", this);
uvm_config_db#(uvm_object_wrapper)::set(this,
    "*seqr.main_phase", "default_sequence", p_lib);
```

- Configuration problems can be difficult to debug

- Always check type and path

9-27

Configure Sequence Library

- The default execution behavior can be modified via the UVM `uvm_sequence_library_cfg` class

```
class uvm_sequence_library_cfg extends uvm_object;
  `uvm_object_utils(uvm_sequence_library_cfg)
  uvm_sequence_lib_mode selection_mode;
  int unsigned min_random_count, max_random_count;
  function new(string name="";
              uvm_sequence_lib_mode mode=UVM_SEQ_LIB_RAND,
              int unsigned min=1, int unsigned max=10);
endclass
```

- Combination of `min_random_count` sets the number of sequences to execute
- `selection_mode` can be one of four modes:
 - `UVM_SEQ_LIB RAND` (Random sequence)
 - `UVM_SEQ_LIB RANDC` (Random cyclic sequence)
 - `UVM_SEQ_LIB ITEM` (Random item. No sequence)
 - `UVM_SEQ_LIB USER` (User defined sequence execution)

9-28

Configure Sequence Library Example

■ In test, set sequence library configuration values

```
program automatic test;
import uvm_pkg::*;
import seq_lib_pkg::*;
// include other classes
class test_20_packet extends test_base;
    uvm_sequence_library_cfg seq_lib_cfg;
    // utils macro and constructor not shown
    virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase);
        seq_lib_cfg = new("seq_lib_cfg", UVM_SEQ_LIB_ITEM, 20, 20);
        uvm_config_db #(uvm_sequence_library_cfg)::set(this,
            ".seqr.main_phase", "default_sequence.config", seq_lib_cfg);
    endfunction
endclass
initial
    run_test();
endprogram
```

If the selection mode is set to UVM_SEQ_LIB_ITEM, no sequence is executed. The sequence library will automatically produce the requested number of sequence items (packets).

9-29

```
// component_utils and constructor
`uvm_component_utils(test_20_packet);
function new(string name, uvm_component parent);
    super.new(name, parent);
endfunction : new
```

User Define Sequence Execution

- For UVM_SEQ_LIB_USER mode, user must implement select_sequence() method

```
class my_packet_seq_lib extends packet_seq_lib_base;
  // utils macros and constructor not shown
  `uvm_add_to_seq_lib(direct_sequence, my_packet_seq_lib)
  `uvm_add_to_seq_lib(random_sequence, my_packet_seq_lib)
  `uvm_add_to_seq_lib(burst_sequence, my_packet_seq_lib)
  int count = 0;
  function int unsigned select_sequence(int unsigned max);
    if (count == 0) begin count++; return 0; end
    return ($urandom_range(1, max));
  endfunction: select_sequence
```

Add sequence to library

```
class test_user_seq extends test_base; // other code not shown
  uvm_sequence_library_cfg seq_cfg = new("seq_cfg", UVM_SEQ_LIB_USER, 10);
  function void build_phase(uvm_phase phase); super.build_phase(phase);
    uvm_config_db#(uvm_object_wrapper)::set(this,"*.seqr.main_phase",
      "default_sequence", my_packet_seq_lib::get_type());
    uvm_config_db#(uvm_sequence_library_cfg)::set(this,"*.seqr.main_phase",
      "default_sequence.config", seq_cfg);
  endfunction
endclass
```

Configure for execution in UVM_SEQ_LIB_USER mode

Implement control method

9-30

```
// utils and constructor
`uvm_object_utils(my_packet_seq_lib)
`uvm_sequence_library_utils(my_packet_seq_lib)

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

Agenda: Day 3

**DAY
3**

9 Virtual Sequence/Sequencer

10 Component Phasing Revisited

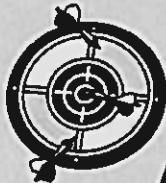


11 UVM Register Abstraction Layer (RAL)

12 Summary



Unit Objectives

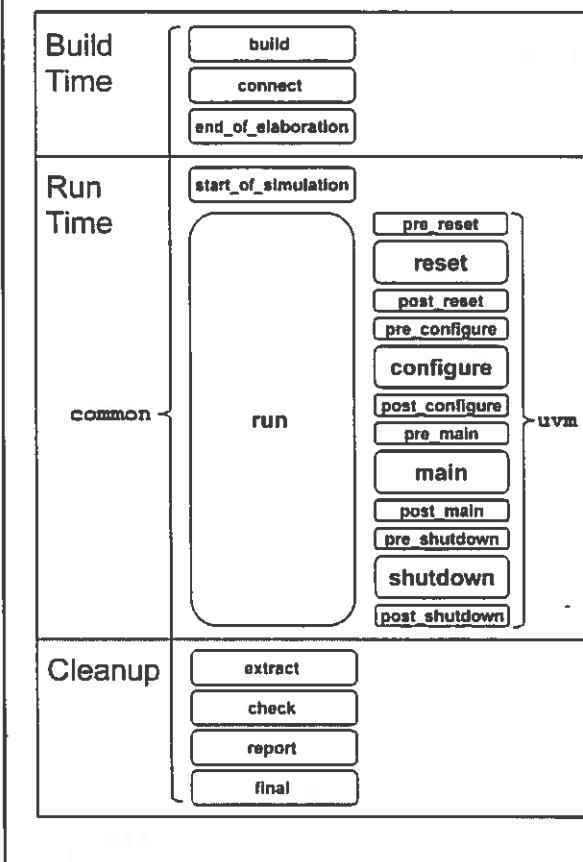


After completing this unit, you should be able to:

- Create synchronized and un-synchronized phase domains
- Control phase timeout
- Create user phases
- Create phase domains
- Implement phase callbacks

10-2

Phasing in UVM 1.0+



- **Synchronized phase execution**
- **Two predefined task "domain"**
 - **common** - Simple components (driver/monitor)
 - ◆ **run**
 - **uvm** - Complex components (test/environment/scoreboard)
 - ◆ **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 structure
connect	Establish cross-component connections
end_of_elaboration	Check for correctness of testbench structure
start_of_simulation	Set configuration for stimulus generation
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 more details please see UVM Reference Guide

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

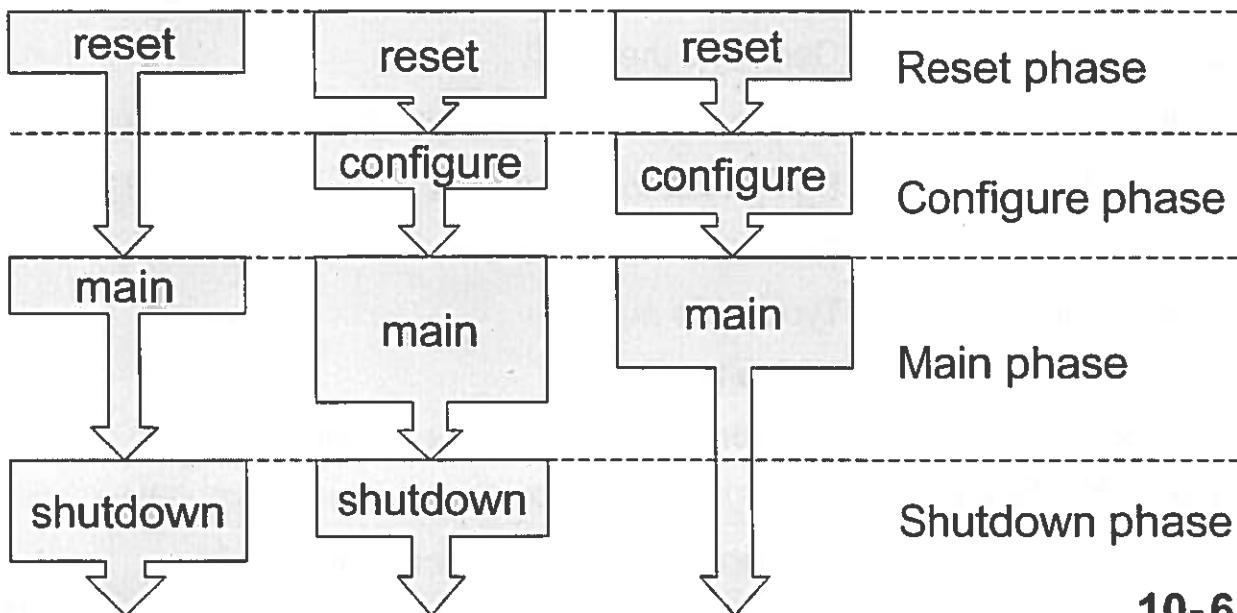
For more details please see UVM Reference Guide

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

component A component B component C



10-6

Phase Objection (1/5)

■ Objection typically raised at stimulus creation

- In sequence

```
class packet_sequence ...;
  // other code not shown
  virtual task pre_start();
    if (get_parent_sequence() == null && starting_phase != null)
      starting_phase.raise_objection(this);
  endtask
  virtual task post_start();
    if (get_parent_sequence() == null && starting_phase != null)
      starting_phase.drop_objection(this);
  endtaskendclass
```

Executes only if sequence is the parent sequence and starting_phase exists

```
class router_env extends uvm_env;
  virtual function void build_phase(uvm_phase phase);
    // other code not shown
    uvm_config_db #(uvm_object_wrapper)::set(this,"agent.seqr.main_phase",
        "default_sequence", packet_sequence::get_type());
  endfunction
endclass
```

Objection set for main phase

10-7

Phase Objection (2/5)

- No objection raised/dropped 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
```

Trace with
+UVM_OBJECTION_TRACE
run-time switch

```
class iMonitor extends ...;
    virtual task run_phase(uvm_phase phase);
        forever begin
            @ (negedge sigs.iMonClk.frame_n[port_id]);
            phase.raise_objection(this);
            get_packet(tr);
            phase.drop_objection(this);
            ...
        end
    endtask
endclass
```

10-8

Phase Objection (3/5)

- Known latency can be taken care of with drain time

```
class test_drain extends test_base;
  `uvm_component_utils(test_drain)
  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfunction
    Object in phase
  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
  Get objection handle
  Set drain time
```

10-9

Phase Objection (4/5)

- Unknown latency can be taken of in scoreboard

```
typedef class scoreboard;
class sb_callback extends uvm_callback;
    function new(string name = "sb_callback");
        super.new(name);
    endfunction
    virtual task end_of_test(scoreboard sb, uvm_phase phase); endtask
endclass

class scoreboard extends uvm_scoreboard; // other code not shown
    `uvm_register_cb(scoreboard, sb_callback)
    virtual task shutdown_phase(uvm_phase phase);
        phase.raise_objection(this);
        `uvm_do_callbacks(scoreboard, sb_callback, end_of_test(this, phase));
        phase.drop_objection(this);
    endtask
endclass
```

Can register n-number of end-of-test condition.
If no callback is registered, does not block.

10-10

Phase Objection (5/5)

- Callbacks can access DUT signal for end of test

```
class sb_eot_callback extends sb_callback; // other code not shown
    virtual router_io sigs;
    virtual task end_of_test(scoreboard sb, uvm_phase phase);
        uvm_config_db#(virtual router_io)::get(sb, "", "sigs", sigs);
        @(sigs.done); // condition of completion of dut activities
    endtask
endclass
class test_base extends uvm_base; // other code not shown
    sb_eot_callback eot_cb;
    virtual function void build_phase(uvm_phase phase);
        super.build_phase(phase); // other code not shown
        uvm_config_db#(virtual router_io)::set(this, "env.sb",
                                                "router_io", router_test_top.sigs);
    endfunction
    virtual function void connect_phase(uvm_phase phase);
        super.connect_phase(phase);
        eot_cb = new();
        uvm_callbacks #(scoreboard, sb_callback)::add(env.sb, eot_cb);
    endfunction
endclass
```

10-11

Phase Timeout

If objections are not dropped, timeout will kill task phases

- **+UVM_TIMEOUT**

- Run-Time switch
- Timeout specification for each task phase
- Defaults to 9,200 sec
 - ◆ Causes error and warning messages if reached

- **uvm_top.set_timeout(.timeout(1ms), .overridable(1))**

- Overrides +UVM_TIMEOUT
- Typically embedded in test code
 - ◆ Typically in test task phases
 - If **.overridable** is set to 1, can be set to different value for different phases
- Task phases before the `set_timeout()` method call defaults to what was set by +UVM_TIMEOUT

10-12

Advanced Features

■ Phase Domains

- UVM has one default phase domain
- Phases within same domain are synchronized
- No inter-domain synchronization by default
- User can set full and partial synchronization

■ User Defined Phases

- Can be mixed with predefined phases

■ Phase Jumping

- Forwards and Backwards

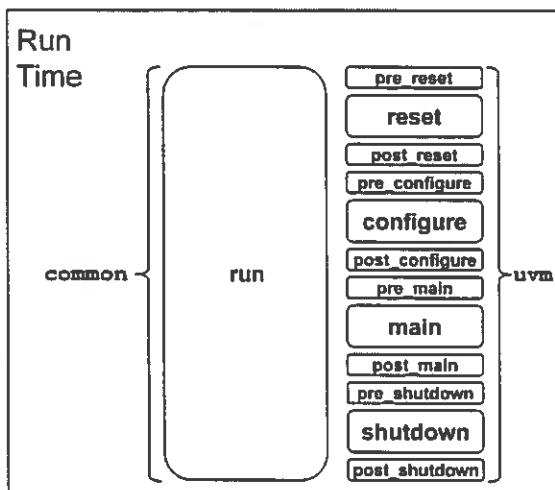
**Only recommended for (Environment) Implementers
not (Component) Developers**

10-13

Phase Domains (1/2)

- Run Time task phases are organized as domains
 - There are two Run Time task domains: **common** and **uvm**
 - ◆ Executing concurrently as two synchronized threads

```
virtual task main_phase(uvm_phase phase);  
  $display("Phase in %s domain", phase.get_domain_name());
```



10-14

Phase Domains (2/2)

- User can create customized domains

```
class top_env extends uvm_env;
    sub_env env0, env1;
    uvm_domain domain0=new("domain0"), domain1=new("domain1");

    virtual function void connect_phase(uvm_phase phase);
        env0.set_domain(domain0);
        env1.set_domain(domain1);

        domain1.sync(.target(this.get_domain()));
        // domain0.sync(.target(this.get_domain()),
        //             .phase(uvm_main_phase::get()));
        // domain0.sync(.target(this.get_domain()),
        //             .phase(uvm_post_main_phase::get()),
        //             .with_phase(uvm_shutdown_phase::get()));

        ...
    endfunction
endclass
```

Created two user domains

env0's phases are independent of all other domains
env1's phases are synchronized with **top_env**

env0's specified phase is sync'ed with **top_env**

10-15

Note: sync() method only synchronize the starting time for the specified phase. The starting time for the following phase are not synchronized.

User Defined Phase

- User can create customized phases

User phase name

Phase name prefix

```
'uvm_user_task_phase(new_cfg, driver, my_)
```

Component where phase will execute

Macro creates a new phase class called **my_new_cfg_phase**
driver must implement **new_cfg_phase(uvm_phase phase)**

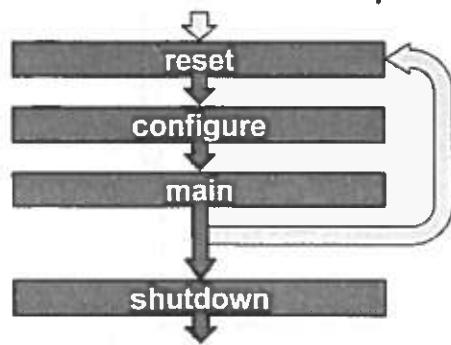
```
class environment extends uvm_env; // utils and constructor
virtual function void build_phase(uvm_phase phase);
    uvm_domain env_domain = this.get_domain();
    super.build_phase(phase);
    drv = driver::type_id::create("drv", this);
    env_domain.add(my_new_cfg_phase::get(),
                    .after_phase(uvm_configure_phase::get()));
endfunction
endclass
```

Add phase into domain

10-16

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 implement `reset_phase()` to return control signals to de-asserted state
 - All residual properties must be deleted
 - All residual processes must be killed



```
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
```

10-17

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
 - Clears objection for phase when called

```
class driver extends uvm_driver #(...);
    event update;
    covergroup d_cov @(update); ... endgroup
    virtual task run_phase(...);
        forever begin
            seq_item_port.get_next_item(req);
            process(req); -> update;
            seq_item_port.item_done();
        end
    endtask
endclass

virtual task main_phase(...);
    forever begin
        @(update);
        if ($get_coverage() == 100.0)
            phase.jump(uvm_post_main_phase::get());
    end
endtask
```

10-18

Phase Callbacks (1/2)

■ phase_started

- Called when phase starts to execute

```
virtual function void phase_started(uvm_phase phase);
    if (phase.is_before(uvm_reset_phase::get())) begin
        ...
    end else begin
        if (phase.is(uvm_reset_phase::get())) begin
            ...
        end else begin
            if (phase.is_after(uvm_reset_phase::get())) begin
                ...
            end
        end
    end
endfunction : phase_started
```

10-19

Phase Callbacks (2/2)

- **phase_ready_to_end**

- Called when all objections for phase are dropped

- **phase_ended**

- Called before phase terminates

```
virtual function void phase_ready_to_end(uvm_phase phase);  
  
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  
    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

Having completed this unit, you should be able to:

- Create synchronized and un-synchronized phase domains
- Control phase timeout
- Create user phases
- Create phase domains
- Implement phase callbacks

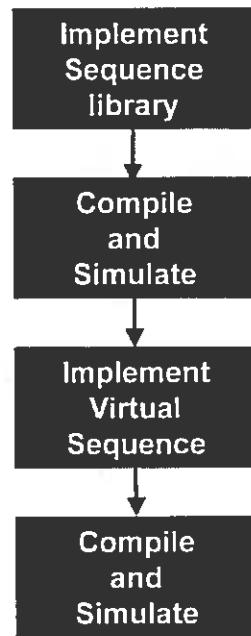
10-22

Lab 6 Introduction

Implement sequence library and virtual sequence



45 min



10-23

Appendix

uvm_phase Class Key Methods

Component Phasing Guideline

Jump Code Example

uvm_phase Class Key Methods

uvm_phase Class Key Methods

- The following are key methods of the uvm_phase class

```
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_state get_state(); // possible states: UVM_EQ, UVM_NE, UVM_LT, UVM_LTE, UVM_GT, UVM_GTE
  task wait_for_state(uvm_phase_state state, uvm_wait_op op=UVM_EQ); // UVM_EQ, UVM_NE, UVM_LT, UVM_LTE, UVM_GT,
  UVM_GTE
  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);
  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();
endclass
```

10-26

Component Phasing Guideline

Driver Guideline

- Emulates launch and capture registers
- Build time phases
 - build_phase() // retrieve configuration
- Run time phases
 - start_of_simulation_phase() // print configuration
 - run_phase() // get and process item
- Cleanup phases
- Callbacks
 - phase_ended() // for jump back

10-28

Monitor Guideline

- Emulates capture register
- Build time phases
 - build_phase() // retrieve configuration
- Run time phases
 - start_of_simulation_phase() // print configuration
 - run_phase() // report observed transaction via analysis port
- Cleanup phases
- Callbacks
 - phase_ended() // for jump back

10-29

Agent Guideline

- **A container class for interface-base components**
 - No behavioral code
- **Build time phases**
 - `build_phase()` // construct sub-components
 // retrieve and set sub-component configuration
 - `connect_phase()` // connect sub-components
- **Run time phases**
 - `start_of_simulation_phase()` // report configuration
- **Cleanup phases**

10-30

Scoreboard Guideline

- **Tracks correctness of operation**
- **Build time phases**
 - build_phase() // construct sub-components
 // retrieve configuration
 - connect_phase() // connect analysis ports
- **Run time phases**
 - start_of_simulation_phase() // print configuration
 - shutdown_phase() // implement end of test callbacks
- **Cleanup phases**
 - report_phase() // report self-check results
- **Callbacks**
 - phase_ended() // for jump back

10-31

Environment Phase Guideline

- A container class for all DUT verification components
 - No behavioral code
- Build time phases
 - build_phase() // construct sub-components
 // retrieve and set sub-component configuration
 - connect_phase() // connect sub-components
 - end_of_elaboration_phase() // check structure
- Run time phases
 - start_of_simulation_phase() // display configuration
- Cleanup phases
 - report_phase() // report summary

10-32

Test Phase Guideline

- **No restriction/limitation of phases**
- **Build time phases**
 - build_phase() // construct and configure environment
 - connect_phase() // make changes to environment connections
- **Run time phases**
 - start_of_simulation_phase() // display configuration
 - run_phase() // set dynamically changing configurations
 - reset_phase() // execute reset sequence
 - configure_phase() // execute configure sequence
 - main_phase() // execute stimulus sequence
 - shutdown_phase() // additional end of test condition
- **Cleanup phases**
 - final_phase() // report final summary for test

10-33

Jump Code Example

Driver Code for Jump

```
class driver extends uvm_driver #(packet);
  uvm_process m_proc[$];

  function void phase_ended(uvm_phase phase);
    uvm_phase jump_phase = phase.get_jump_target();
    if (jump_phase != null) begin
      `uvm_info("JUMP", {"to", jump_phase.get_name()}, UVM_MEDIUM);
      foreach (m_proc[i]) begin
        m_proc[i].kill();
      end
      m_proc.delete();
      if (req != null) begin
        seq_item_port.item_done();
        req = null;
      end
    end
  endfunction: phase_ended
// continue on next page
```

10-35

Driver Code for Jump

```
// continue from previous page
virtual task run_phase(uvm_phase phase);
    forever begin
        fork begin
            uvm_process p = new(process::self());
            m_proc.push_back(p);
            send_packet();
            m_proc.pop_front();
        end join
    end
endtask
virtual task send_packet();
    seq_item_port.get_next_item(req);
    send(req);
    seq_item_port.item_done();
    req = null;
endtask: send_packet
endclass
```

10-36

Monitor Code for Jump

```
class monitor extends uvm_monitor;
  uvm_process m_proc[$];
  function void phase_ended(uvm_phase phase);
    uvm_phase jump_phase = phase.get_jump_target();
    if (jump_phase != null) begin
      `uvm_info("JUMP", {"to", jump_phase.get_name()}, UVM_MEDIUM);
      foreach (m_proc[i]) begin
        m_proc[i].kill();
      end
      m_proc.delete();
    end
  endfunction: phase_ended
  virtual task run_phase(uvm_phase phase);
    forever begin
      fork begin
        uvm_process p = new(process::self());
        m_proc.push_back(p);
        get_packet();
        m_proc.pop_front();
      end join
    endtask
  endclass
```

10-37

Scoreboard Jump Code

```
class scoreboard #(type T = packet) extends uvm_scoreboard;
    T expected_list[$];
    function void phase_ended(uvm_phase phase);
        uvm_phase jump_phase = phase.get_jump_target();
        if (jump_phase != null) begin
            `uvm_info("JUMP", phase.get_name(), UVM_MEDIUM);
            expected_list.delete();
        end
    endfunction: phase_ended
    virtual function void write_before(T tr);
        expected_list.push_back(tr);
    endfunction
endclass
```

10-38

Agenda: Day 3

**DAY
3**

9 Virtual Sequence/Sequencer

10 Component Phasing Revisited



11 UVM Register Abstraction Layer (RAL)

12 Summary



Synopsys 40-I-054-SSG-004

© 2013 Synopsys, Inc. All Rights Reserved

11-1

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

3

- Every DUT has them
- First to be verified
 - Reset value
 - Bit(s) behavior
- High maintenance
 - Modify tests
 - Modify firmware model

Registers

3.1 MODER (Mode Register)

Bit #	Access	Description
31-17		Reserved
16	RW	RECCSMLL – 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 maximum frame length is equal to MINFL)
14	RW	HUGEN – Huge Packet 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

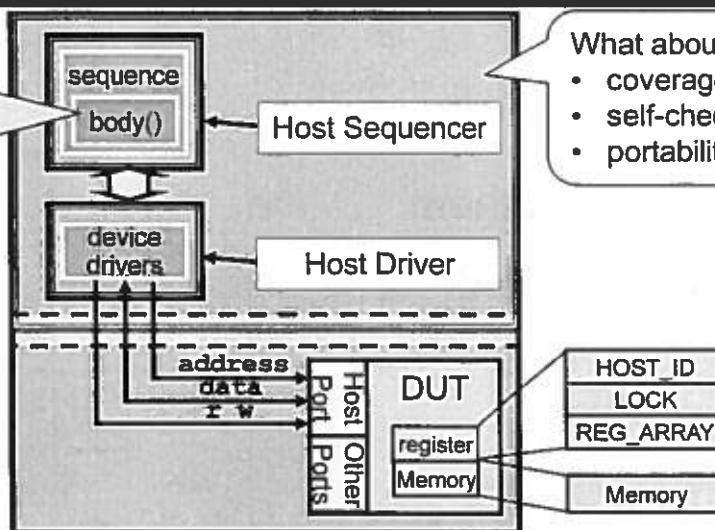
The register and memory space of a design brings a lot of overhead to verification. Any change to the register/memory space in the specification has an avalanche effect on the DUT and TB. Every test and component affected must be changed. This can be time consuming as well as a maintenance nightmare (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?

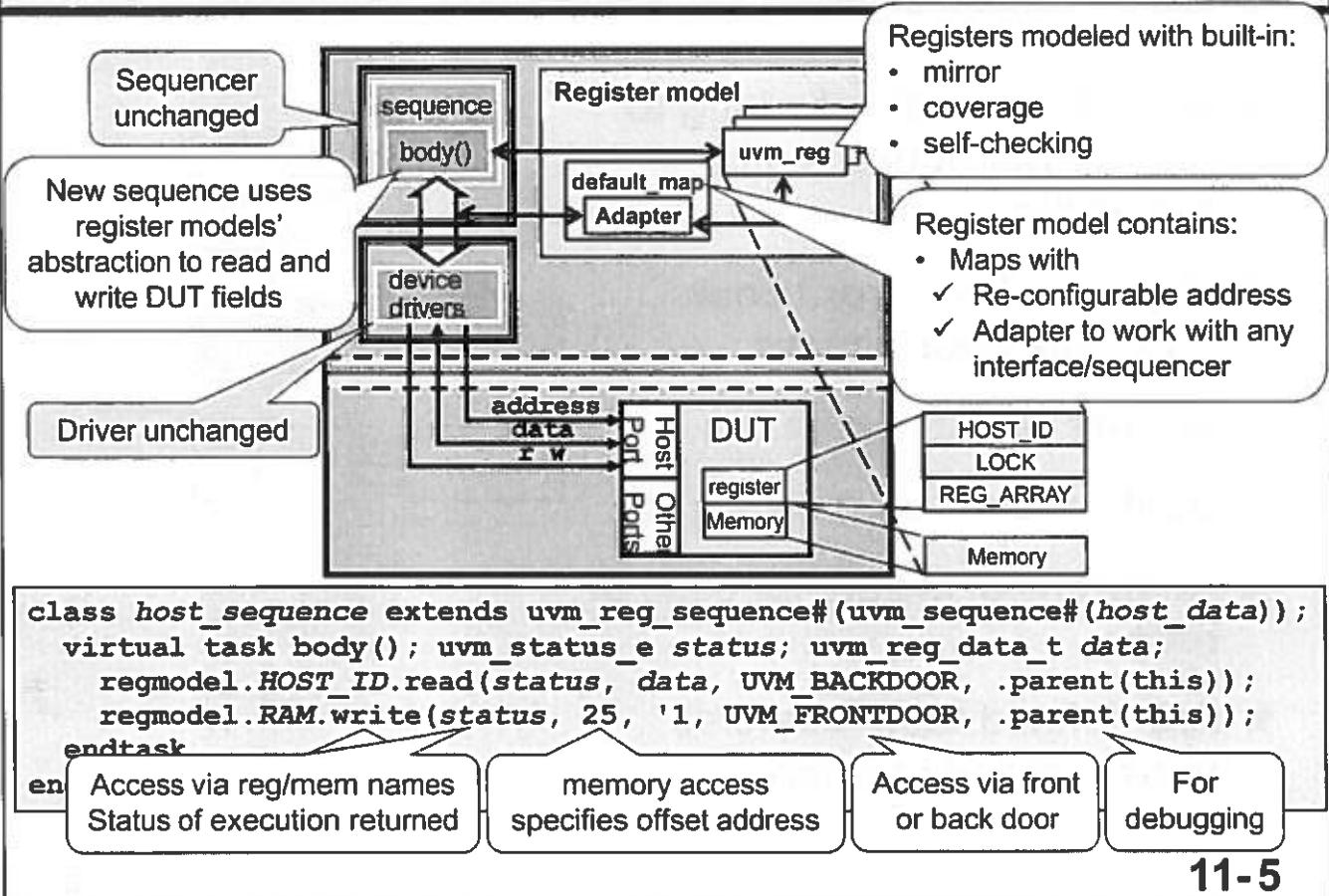


```
class host_sequence extends uvm_sequence #(host_data);    ...;
  virtual task body();
    `uvm_do_with(req, {addr=='h100; data=='1; kind==host_data::WRITE;});
    `uvm_do_with(req, {addr=='h1025; kind==host_data::READ;});  ....;
  endtask
endclass
```

Address hardcoded!
Field name unknown!
Status of execution unknown!
Front door access only!

11-4

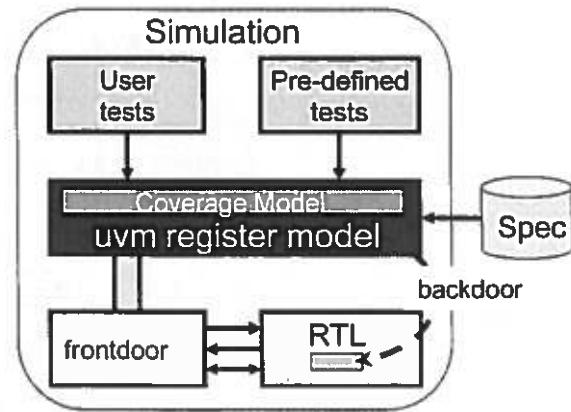
Testbench with UVM Register Abstraction



11-5

UVM Register Abstraction

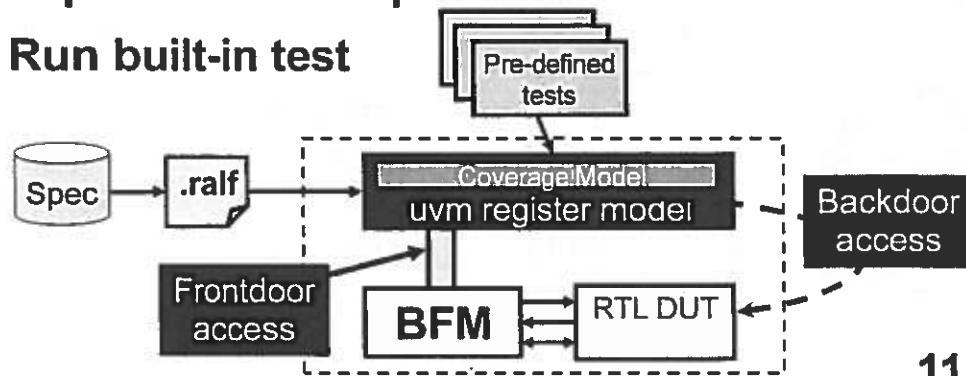
- Abstracts reading/writing to configuration fields and memories
- Supports both front door and back door access
- Mirrors register data
- Built-in functional coverage
- Hierarchical model for ease of reuse
- Pre-defined tests exercise registers and memories



11-6

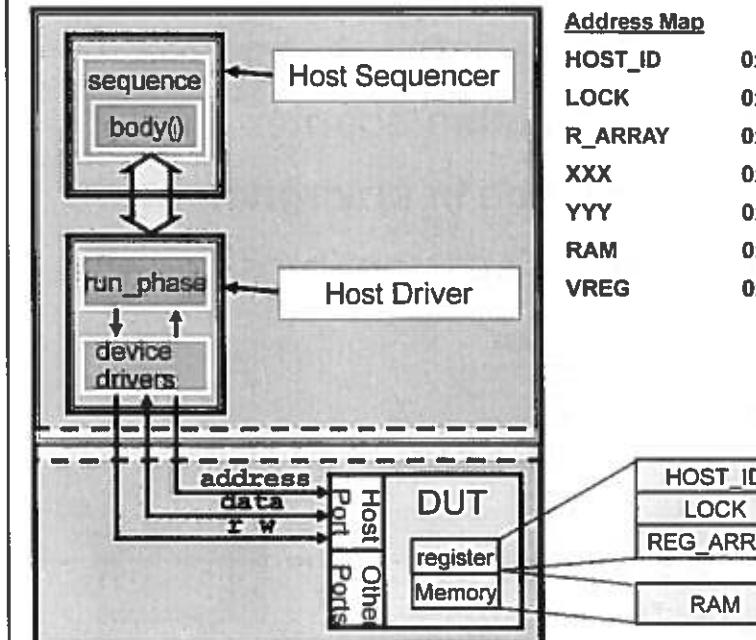
Implement UVM Register Abstraction

- 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 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	0xff

R_ARRAY[256] Registers

Field	H_REG
Bits	15-0
Mode	rw
Reset	0x00

Register XXX

Register XXX

Register YYY

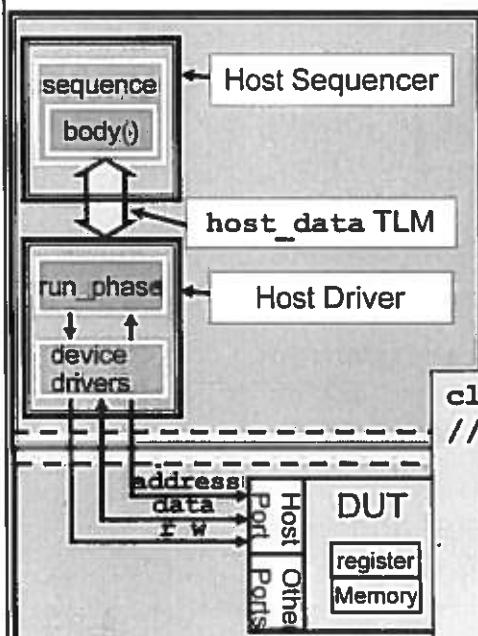
RAM (4K)	
Bits	15-0
Mode	rw

VREG[16]

Bits	15-0
-------------	------

11-8

Step 1: Create Host Data & Driver



```
class host_data extends uvm_sequence_item;
// constructor and utils macro not shown
typedef enum int unsigned {
    READ,
    WRITE
} kind_e;
typedef ... (...) status_e;
rand kind_e kind;
status_e status;
rand bit[15:0] addr, data;
endclass: host_data
```

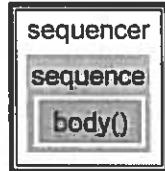
Transaction specifies operation, address, data and status

```
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: host_driver
```

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==host_data::READ;});
    `uvm_do_with(req, {addr=='h100; data='1; kind==host_data::WRITE;});
    `uvm_do_with(req, {addr=='h1025; kind==host_data::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 seqr;
endclass

class host_env extends uvm_env; // other support code not shown
    host_agent h_agent;
    function void build_phase(uvm_phase phase);
        // super.build_phase and construction of components not shown
        uvm_config_db#(uvm_object_wrapper)::set(this,"h_agent.configure_phase",
            "default_sequence", host_bfm_sequence::get_type());
    endfunction
endclass

class test_base extends uvm_test; // other support code not shown
    host_env h_env; 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_agent", "sigs",
            router_test_top.host);
    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 {
        bits 8;
        access ro;
        reset 'h03;
    }
    field CHIP {
        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;
    }
}

```

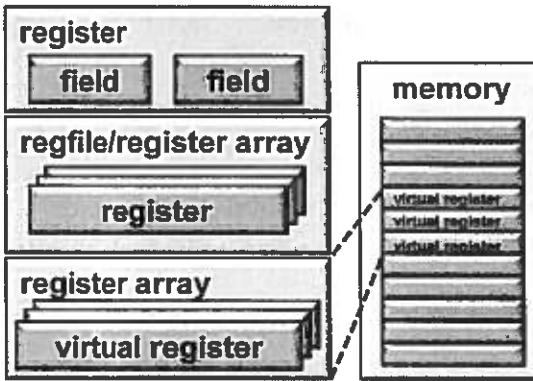
```

regfile REG_FILE {
    register XXX (xxx) @'h0 {...}
    register YYY (yyy) @'h1 {...}
}

memory RAM {
    size 4k;
    bits 16;
    access rw;
}

```

RTL register name



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	0xff

R_ARRAY[256] Registers	
Field	H_REG
Bits	15-0
Mode	rw
Reset	0x00

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

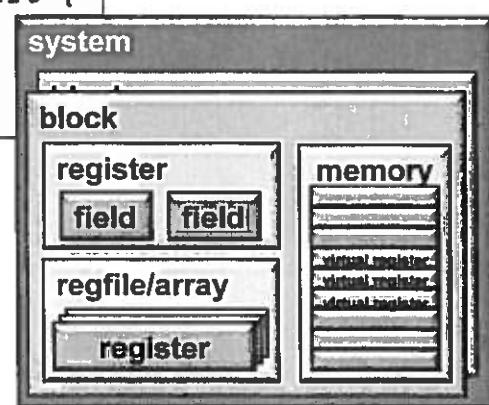
.ralf block emulates RTL block module level
Specifies all content of block

```
block host_regmodel { Register name in block Address
    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 @ 'h0FF0 {
        field VREG { bits 16; access rw; }
    }
}
```

Address Map	
HOST_ID	0x0000
LOCK	0x0100
R_ARRAY	0x1000-0x10FF
XXX	0x3000
YYY	0x3001
RAM	0x4000-0x4FFF
VREG	0x4FF0-0x4FFF

Module instance name in DUT

```
system dut_regmodel {
    bytes 2;
    block host_regmodel      (host)     @ 'h0000;
    block other_regmodel     (other)    @ 'h8000;
}
```



.ralf system emulates upper layer module
which instantiates the block module

11-13

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

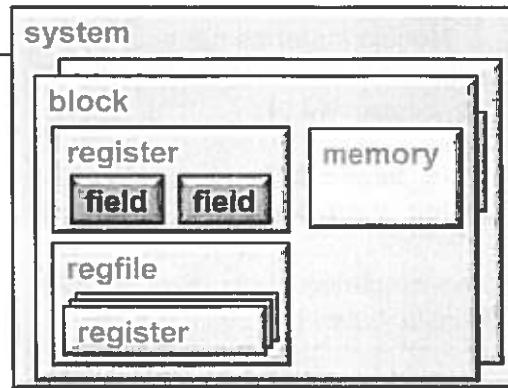
UVM Register Abstraction File (.ralf) Syntax

■ Field

- Contains Bits, Access, Reset, Constraints and Coverage

```
field field_name {
    bits n;
    access rw|ro|wo|w1|w0c|w1c|rcl|...;
    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_ralgen_ug.pdf` in VCS installation directory: `$VCS_HOME/doc/UserGuide/pdf`.

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_ralgen_ug.pdf` in VCS installation directory: `$VCS_HOME/doc/UserGuide/pdf`.

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]

11-16

For detailed RALF file syntax and description, please consult uvm_ralgen_ug.pdf in VCS installation directory: \$VCS_HOME/doc/UserGuide/pdf.

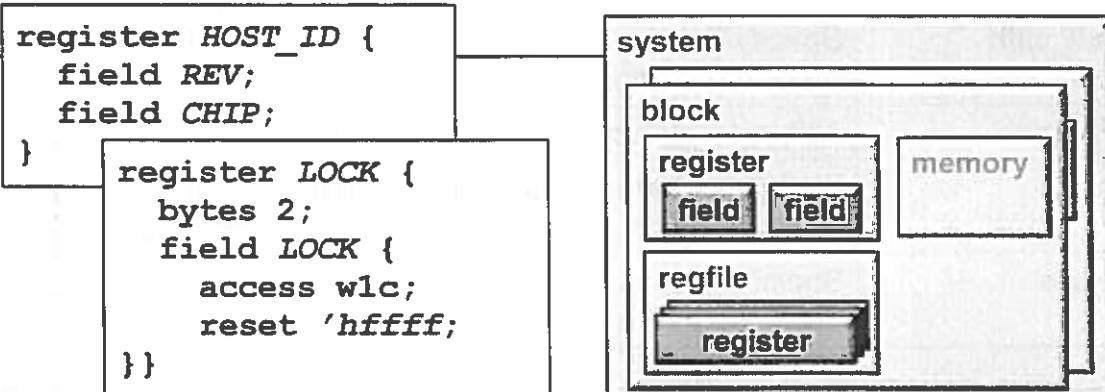
UVM Register Abstraction File (.ralf) Syntax

■ 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;
    field CHIP;
}
register LOCK {
    bytes 2;
    field LOCK {
        access wlc;
        reset 'ffff;
    }
}
```



11-17

For detailed RALF file syntax and description, please consult uvm_ralgen_ug.pdf in VCS installation directory: \$VCS_HOME/doc/UserGuide/pdf.

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_ralgen_ug.pdf` in VCS installation directory: `$VCS_HOME/doc/UserGuide/pdf`.

UVM Register Abstraction File (.ralf) Syntax

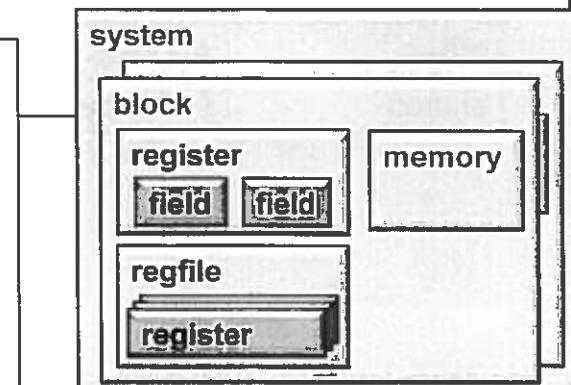
■ Register File

- 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 'h00;
        }
        register YYY (yyy) @'h1 {
            ...
        }
    }
}
```



11-19

For detailed RALF file syntax and description, please consult uvm_ralgen_ug.pdf in VCS installation directory: \$VCS_HOME/doc/UserGuide/pdf.

UVM Register Abstraction: Register File

RAL register file can have any of the following specifications

[n]	Array of registers
hdl_path	Specify the module instance containing the register (backdoor access)
offset	Address offset within the register file
shared	All blocks instancing this register share the space

11-20

For detailed RALF file syntax and description, please consult uvm_ralgen_ug.pdf in VCS installation directory: \$VCS_HOME/doc/UserGuide/pdf.

UVM Register Abstraction File (.ralf) Syntax

■ Memory

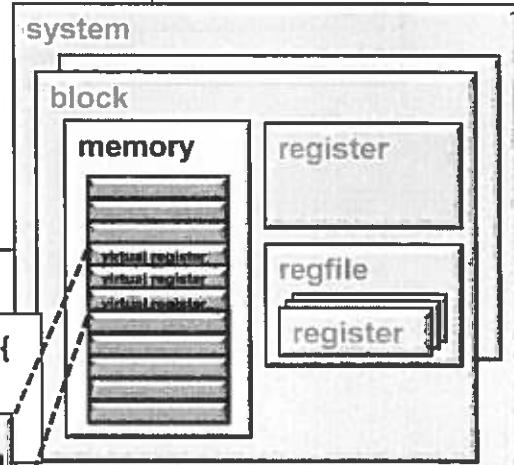
- Leaf level definition

```
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; }
}
```



11-21

For detailed RALF file syntax and description, please consult uvm_ralgen_ug.pdf in VCS installation directory: \$VCS_HOME/doc/UserGuide/pdf.

UVM Register Abstraction: Memory

RAL memories can have any of the following specifications

size	Number of consecutive addresses
bits	Number of bits in each address location
access	Allowed access type

11-22

For detailed RALF file syntax and description, please consult `uvm_ralgen_ug.pdf` in VCS installation directory: `$VCS_HOME/doc/UserGuide/pdf`.

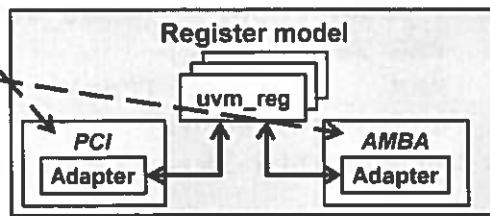
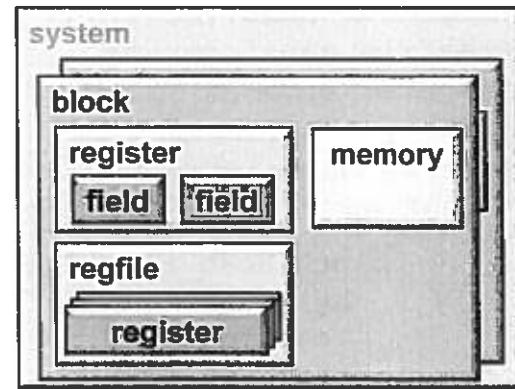
UVM Register Abstraction File (.ralf) Syntax

■ 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>  
    }  
}
```



11-23

For detailed RALF file syntax and description, please consult `uvm_ralgen_ug.pdf` in VCS installation directory: `$VCS_HOME/doc/UserGuide/pdf`.

UVM Register Abstraction: Block

- Contains register or memory elements of module

```
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           Signal name in module
    register LOCK               (lock)
    register R_ARRAY[256]        (host_reg[%d]) @'h1000;
    regfile REG_FILE           @'h0000;
    memory RAM                 (ram)          @'h0100;
    virtual register VREG[16]   RAM @'h3000;
    field VREG { bits 16; access rw; }
}
```

Must add index

Signal name in module

Register array

11-24

For detailed RALF file syntax and description, please consult uvm_ralgen_ug.pdf in VCS installation directory: \$VCS_HOME/doc/UserGuide/pdf.

UVM Register Abstraction: Block

RAL block can have any of the following specifications

bytes	Number of bytes concurrently addressed
endian	Specifies how wide registers are mapped

11-25

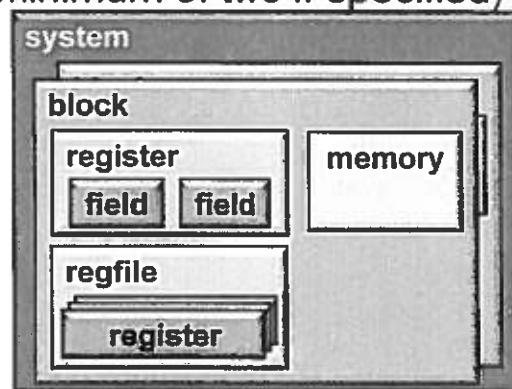
For detailed RALF file syntax and description, please consult `uvm_ralgen_ug.pdf` in VCS installation directory: `$VCS_HOME/doc/UserGuide/pdf`.

UVM Register Abstraction File (.ralf) Syntax

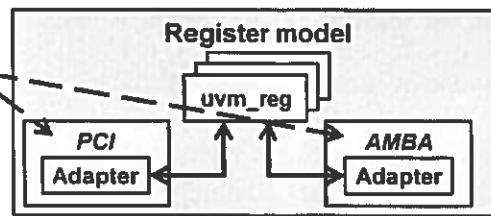
■ Top Level or subsystem

- 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>  
    }  
}
```



11-26

For detailed RALF file syntax and description, please consult `uvm_ralgen_ug.pdf` in VCS installation directory: `$VCS_HOME/doc/UserGuide/pdf`.

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;
}
```

11-27

For detailed RALF file syntax and description, please consult uvm_ralgen_ug.pdf in VCS installation directory: \$VCS_HOME/doc/UserGuide/pdf.

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 PORT_LOCK {
    field LOCK {...}
}
register REG_ARRAY {
    field HOST_REG {...}
}
regfile REG_FILE {...}
memory RAM {...}
block host_regmodel {...}
system dut_regmodel {...}
```

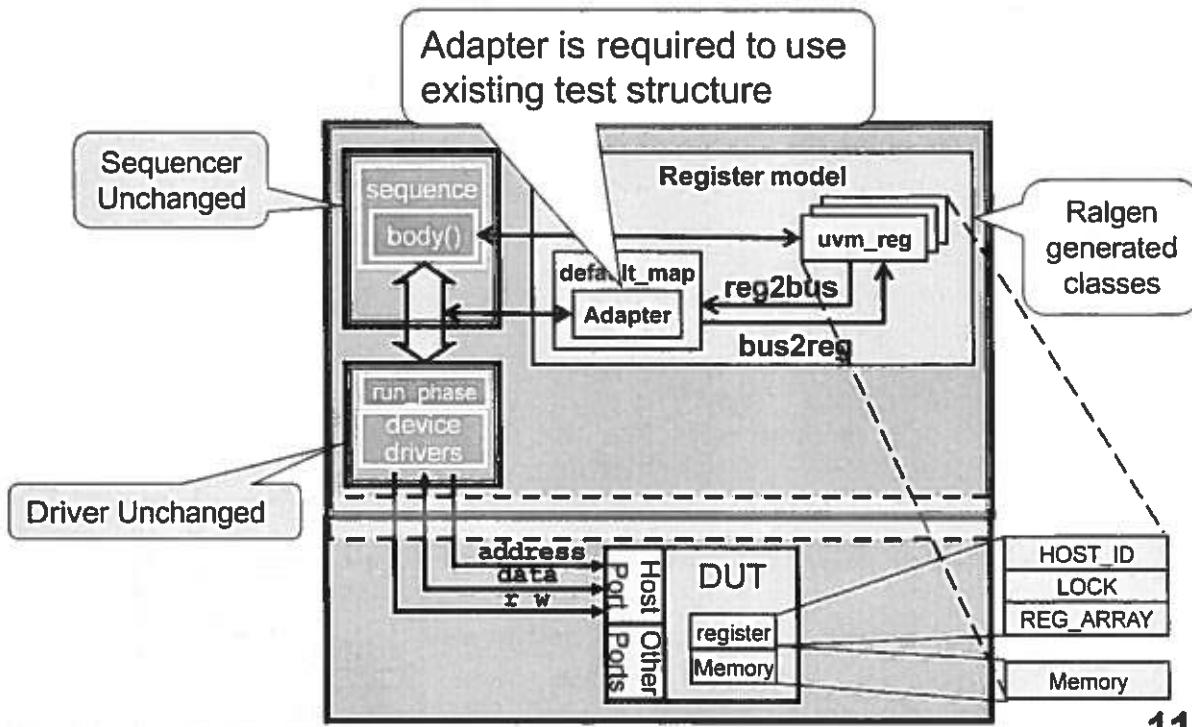
```
// ral_dut_regmodel.sv
class ral_reg_HOST_ID extends uvm_reg;
    uvm_reg_field REV_ID;
    uvm_reg_field CHIP_ID;
    ...
endclass : ral_reg_HOST_ID
class ral_reg_PORT_LOCK extends uvm_reg;
class ral_reg_REG_ARRAY extends uvm_reg;
class ral_regfile_REG_FILE extends uvm_regfile;
class ral_mem_RAM extends uvm_mem;
class ral_block_host_regmodel extends uvm_reg_block;
    rand ral_reg_HOST_ID      HOST_ID;
    rand ral_reg_PORT_LOCK   PORT_LOCK;
    rand ral_reg_REG_ARRAY   REG_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

11-28

Step 4: Create UVM Register Adapter

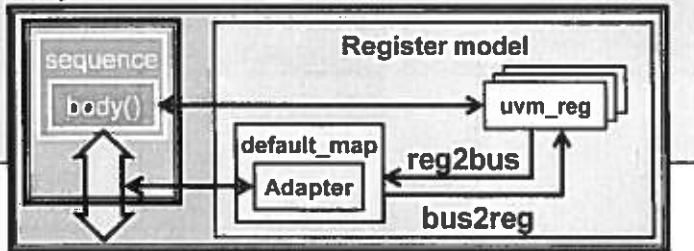
- Environment needs adapters to convert UVM register data to bus transactions for each agent/sequencer



11-29

Sequencer Adapter Class (One per Interface)

```
class reg_adapter extends uvm_reg_adapter;
    virtual function uvm_sequence_item reg2bus(const ref uvm_reg_bus_op rw);
        host_data tr = host_data::type_id::create("tr");
        tr.kind = (rw.kind inside {UVM_READ, UVM_BURST_READ}) ?
                    host_data::READ : host_data::WRITE;
        tr.addr = rw.addr;  tr.data = rw.data; return tr;
    endfunction
    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.kind = (tr.kind == host_data::READ) ? UVM_READ : UVM_WRITE;
        rw.addr = tr.addr;  rw.data = tr.data;
        case (tr.status)
            host_data::IS_OK:  rw.status = UVM_IS_OK;
            host_data::NOT_OK: rw.status = UVM_NOT_OK;
            host_data::HAS_X:  rw.status = UVM_HAS_X;
            default: `uvm_fatal(...)
        endcase
    endfunction
endclass
```



11-30

UVM Register Abstraction Sequence

```
class host_sequence_base extends uvm_sequence#(host_data);
  // other code not shown
  virtual task pre_start(); //objection raised here
  virtual task post_start(); //objection dropped here

class host_bfm_sequence extends host_sequence_base;
  // other code not shown
  virtual task body();
    `uvm_do_with(req, {addr=='h000; kind==host_data::READ;});
    `uvm_do_with(req, {addr=='h4009;data=='1;kind==host_data::WRITE;});
  endtask
endclass
```

Becomes

```
class host_ral_sequence extends uvm_reg_sequence #(host_sequence_base);
  ral_block_host_regmodel regmodel; // other code not shown
  virtual task pre_start(); super.pre_start(); // to raise objection
  uvm_config_db#(ral_block_host_regmodel)::get(get_sequencer(), "", "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
```

Abstracted and self-documenting code

11-31

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 always done in the pre_start() and post_start() method of the host_sequence_base class.

Step 5: Instantiating UVM Register Model

```
class host_env extends uvm_env; // Other code not shown
  host_agent          h_agent;
  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
      string hdl_path;
      regmodel = ral_block_host_regmodel::type_id::create("regmodel", this);
      regmodel.build(); Create UVM register hierarchy Not build_phase()!
      regmodel.lock_model();
    end
    Lock register hierarchy and create address map
  end
  if (!uvm_resource_db#(string)::read_by_name("host_regmodel",
                                              "hdl_path", hdl_path, this))
    regmodel.set_hdl_path_root(hdl_path); // DPI-based backdoor access
  else `uvm_warning("host_regmodel", "HOST XMR path is not set!");
  uvm_config_db#(ral_block_host_regmodel)::set(this, "h_agent.seqr",
                                                "regmodel", regmodel, this);
  Set regmodel for sequence to pick up
end
endfunction: build_phase // Continued on next slide
```

11-32

The backdoor access as shown in the slides is a backdoor access through DPI not direct Verilog XMR (cross-module-reference). The benefit of DPI backdoor access is that the code is independent of compile-methodology. For some compilers, if sections of SystemVerilog code were to be compiled independently then linked together at elaboration or running, the Verilog XMR will create a compile error. With DPI, compile error will not happen.

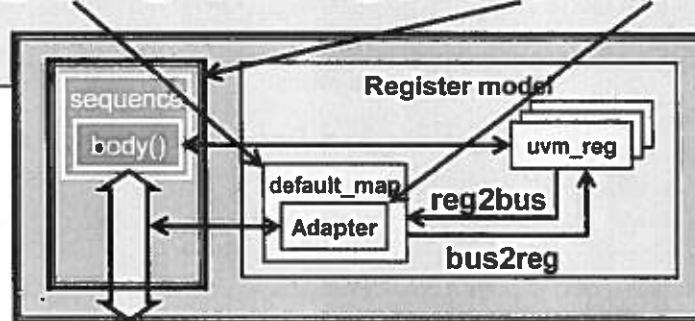
The drawback of DPI backdoor access is that it is much slower than direct Verilog XMR access.

XMR based backdoors are faster. Any typos in the XMR paths are identified at elaboration time. Thus XMR path errors are easier to debug. Whereas for DPI, typos would only be seen in runtime. A caution - XMR based backdoors is a SNPS implementation with ralgen (enabled through -b switch), and requires the path from harness to DUT to be provided as a +define at compile time. Please take a look at Appendix slide 11-68 and 11-70 for additional information.

Tie Sequencer Adapter to Register Map

- Register each sequencer with the associated adapter in the UVM register model

```
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_agent.seqr, adapter);
endfunction
endclass
```



- A map within a register model represents an interface

- Registers with only one interface uses default_map
- Registers with multiple interfaces use map names specified by user (domain name in ralgen)

11-33

Run RAL Sequence Implicitly or Explicitly

```
class test_ral_base extends test_base; // Support code not shown
    virtual function void build_phase(uvm_phase phase); // Other code
        uvm_resource_db #(string)::set("host_regmodel", "hdl_path",
                                      "host_test_top.dut", this); // dpi backdoor
    endfunction
endclass
```

Fill in path to DUT for dpi backdoor access

```
class test_ral_implicit extends test_ral_base; // Support code not shown
    virtual function void start_of_simulation_phase(uvm_phase phase);
        super.start_of_simulation_phase(phase);
        uvm_config_db#(uvm_object_wrapper)::set(this,"*.seqr.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_agent.seqr); ->
        phase.drop_objection(this);
    endtask
endclass
```

Or, execute RAL sequence explicitly

11-34

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 implement, it is very difficult to change the behavior. You cannot easily disable it. You cannot easily replace the execution with something else.

The general recommendation is to go the implicit route for better reuse.

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 set auto predict in test or implement `uvm_reg_predictor`

Sequence Name	Description
<code>uvm_reg_hw_reset_seq</code>	Test the hard reset values of register
<code>uvm_reg_bit_bash_seq</code>	Bit bash all bits of registers
<code>uvm_reg_access_seq</code>	Verify accessibility of all registers
<code>uvm_mem_walk_seq</code>	Verify memory with walking ones algorithm
<code>uvm_mem_access_seq</code>	Verify access by using front and back door
<code>uvm_reg_mem_builtin_seq</code>	Run all reg and memory tests
<code>uvm_reg_mem_hdl_paths_seq</code>	Verify hdl_path for reg and memory
<code>uvm_reg_mem_shared_access_seq</code>	Verify accessibility of shared reg and memory

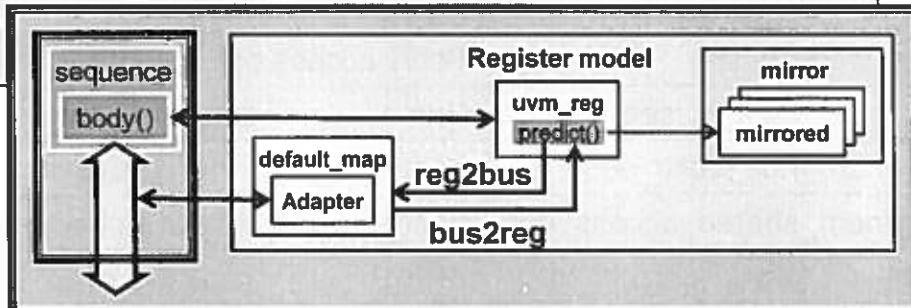
11-35

Implicit Mirror Predictor

■ Mirror updated when registers are written and read

- Advantage:
 - ◆ Simple: Read, write method calls automatically updates mirror. No other user code required.
- Drawbacks:
 - ◆ Timing of mirror update may not be cycle accurate
 - ◆ Changes internal to DUT is not reflected in mirror
 - ◆ Not a good choice if mirror value is needed in user tests

```
class router_env extends uvm_env; ...;
    virtual function void connect_phase(uvm_phase phase); ...;
        regmodel.default_map.set_auto_predict(1);
    endfunction
endclass
```



11-36

Run RAL Test Sequences with Auto Predict

- Call `set_auto_predict(1)` to enable auto predict

```
class test_ral extends test_base; // support code left off
  string seq_name="uvm_reg_bit_bash_seq";
  uvm_reg_sequence selftest_seq;
  virtual_reset_sequence rst_seq;
  virtual function void build_phase(...); super.build_phase(...);
    uvm_config_db#(uvm_object_wrapper)::set(this, "*",
      "default_sequence", null);
  endfunction
  virtual task run_phase(uvm_phase phase);
    rst_seq = virtual_reset_sequence::type_id::create("rst_seq", this);
    phase.raise_objection(this, "Starting tests");
    v_reset_seq.start(env.v_reset_seqr); Run reset Create test
    clp.get_arg_value("+seq=", seq_name);
    $cast(selftest_seq, factory.create_object_by_name(seq_name));
    env.regmodel.default_map.set_auto_predict(1); Enable auto predict
    if not done in env
    selftest_seq.model = env.regmodel;
    selftest_seq.start(env.h_agent.seqr); Run test
    phase.drop_objection(this, "Done with tests");
  endtask
endclass
Can select other sequence at run-time
```

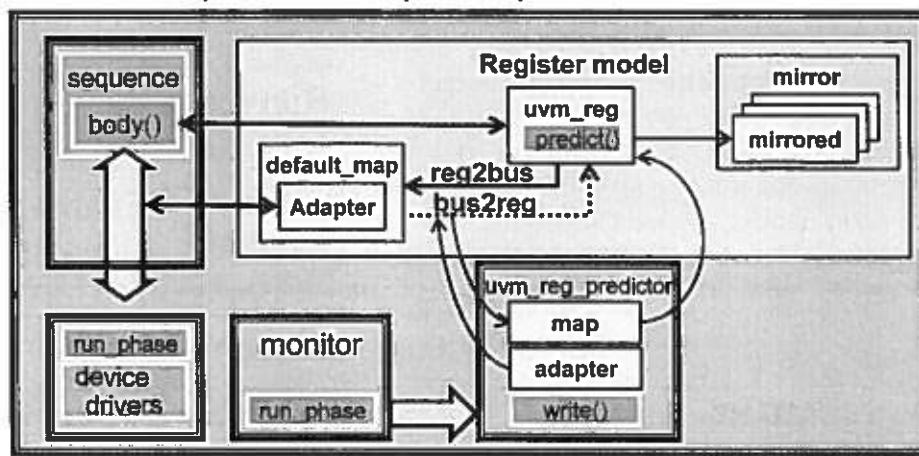
```
simv +UVM_TESTNAME=test_ral +seq=uvm_reg_hw_reset_seq
```

11-37

Explicit Mirror Predictor

■ Mirror updates when monitor observe register changes

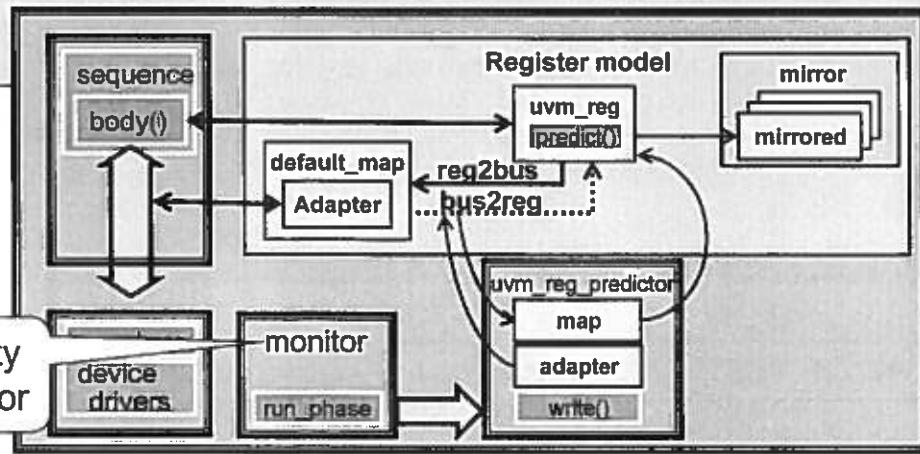
- Advantage:
 - ◆ Mirror is updated based on observation of bus protocol
 - ◆ Changes internal to DUT can be reflected in mirror
 - ◆ Correct choice if mirror value is needed in user tests
- Drawbacks:
 - ◆ More complex to set up. Requires a monitor.



11-38

Implementing 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_agent.analysis_port.connect(hreg_predict.bus_in);
  endfunction
endclass
```



11-39

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.

RAL Test Sequence with Explicit Predict

- Test structure is almost identical to auto predict test
 - Difference is auto predict is turned off

```
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");
    v_reset_seq = virtual_reset_sequence::type_id::create("v_reset_seq",
                                                          this);
    v_reset_seq.start(env.v_reset_seqr);
    clp.get_arg_value("+seq=", seq_name);
    $cast(selftest_seq, factory.create_object_by_name(seq_name));
    env.regmodel.default_map.set_auto_predict(0);
    selftest_seq.model = env.regmodel;
    selftest_seq.start(env.h_agent.seqr);

    phase.drop_objection(this, "Done with register tests");
  endtask
endclass
```

Disable auto predict
if not done in env

Select sequence at run-time

```
simv +UVM_TESTNAME=test_ral +seq=uvm_reg_hw_reset_seq
```

11-40

Unit Objectives Review

Having completed 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

11-41

Appendix

UVM Register Modes

UVM Memory Modes

ralgen Options

UVM Register Class Tree

UVM Register/Memory Class Members

UVM Register Callbacks

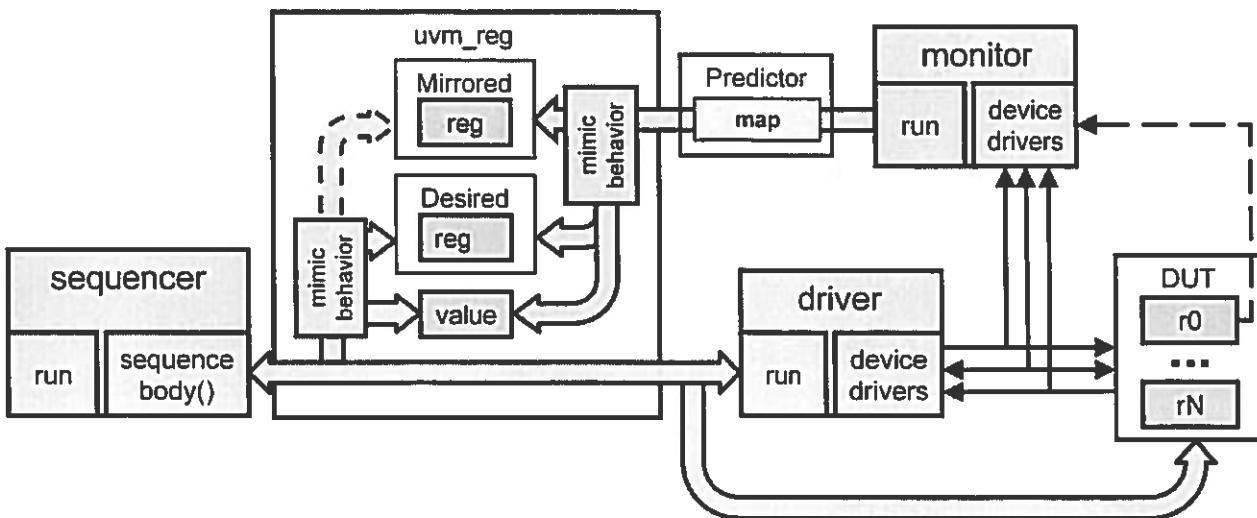
Changing Address Offsets of a Domain

UVM Register Modes

UVM Register Modes

- Frontdoor read/write
- Backdoor read/write/peek/poke
- Mirror set/get/update

Memory is not mirrored



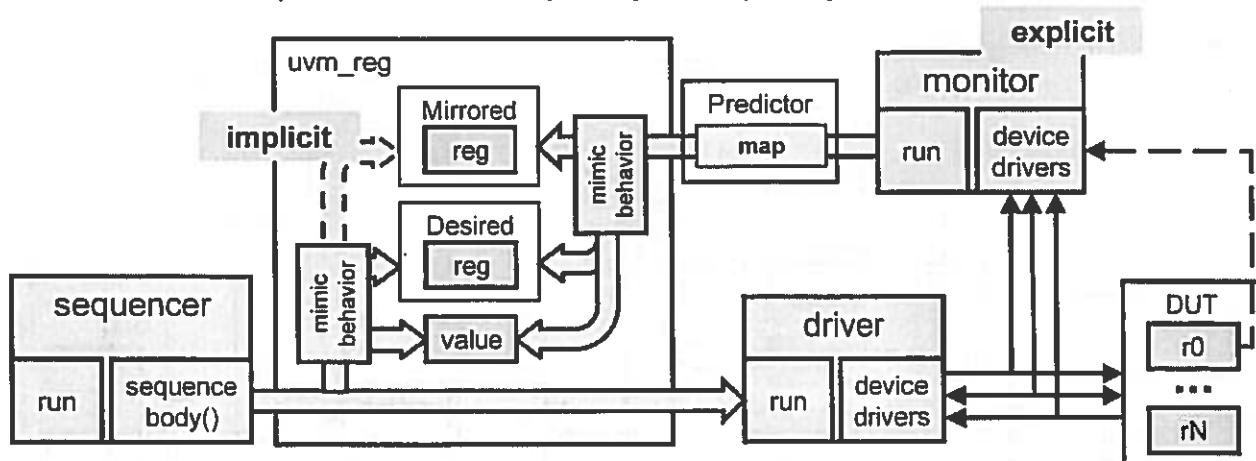
11-44

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 can be different from the actual content of the DUT register. Think of it as a scratch pad for data manipulation before the content is used to populate the DUT register. See **set()** and **update()** methods in the coming slides.

Register Frontdoor Write

- `model.r0.write(status, value, [UVM_FRONTDOOR], .parent(this));`
- `write_reg(model.r0, status, value, [UVM_FRONTDOOR]);`
 - Sequence sets uvm_reg with value
 - uvm_reg content is translated into bus transaction
 - Driver gets bus transaction and writes DUT register
 - Mirror is updated either implicitly or explicitly



11-45

The `write_reg()` method is a convenience method provided by the `uvm_reg_sequence` base class.

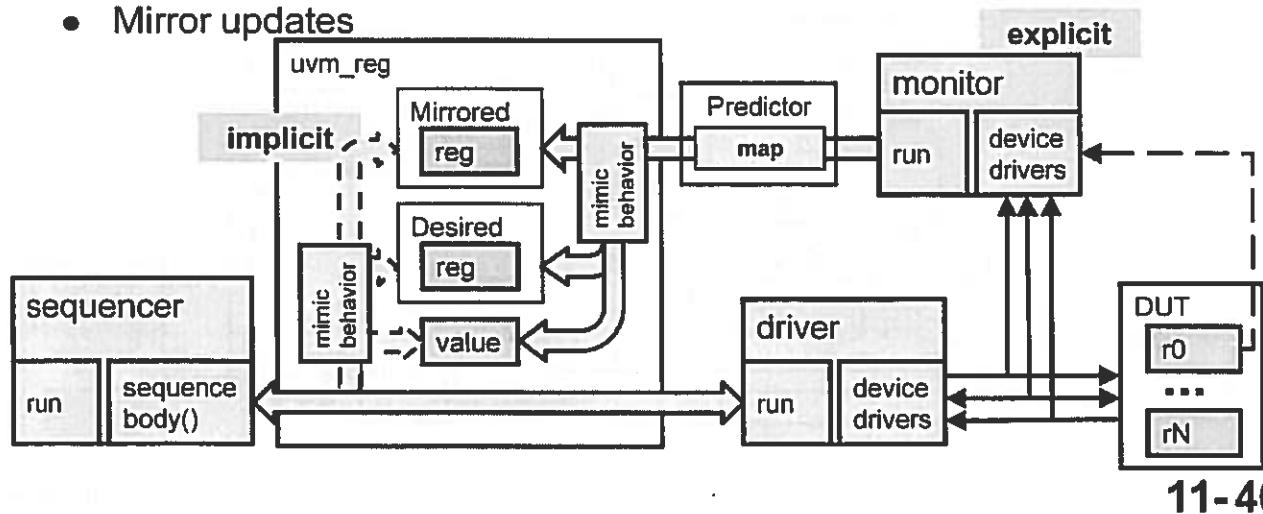
The advantage of using the convenience method is that it is a little more self-documenting and avoids needing to provide the parent handle.

The disadvantage of using the convenience method is that the method implementation is inconsistent. Some `uvm_reg` and `uvm_mem` methods have equivalent convenience methods while others don't. So, if you use the convenience method, it will be very likely that you will have a mixture of convenience methods and non-convenience methods in your sequence. A poor idea.

Until the UVM source code is updated to provide equivalent convenience methods for `uvm_reg` and `uvm_mem` methods, the non-convenience methods may be a better choice.

Register Frontdoor Read

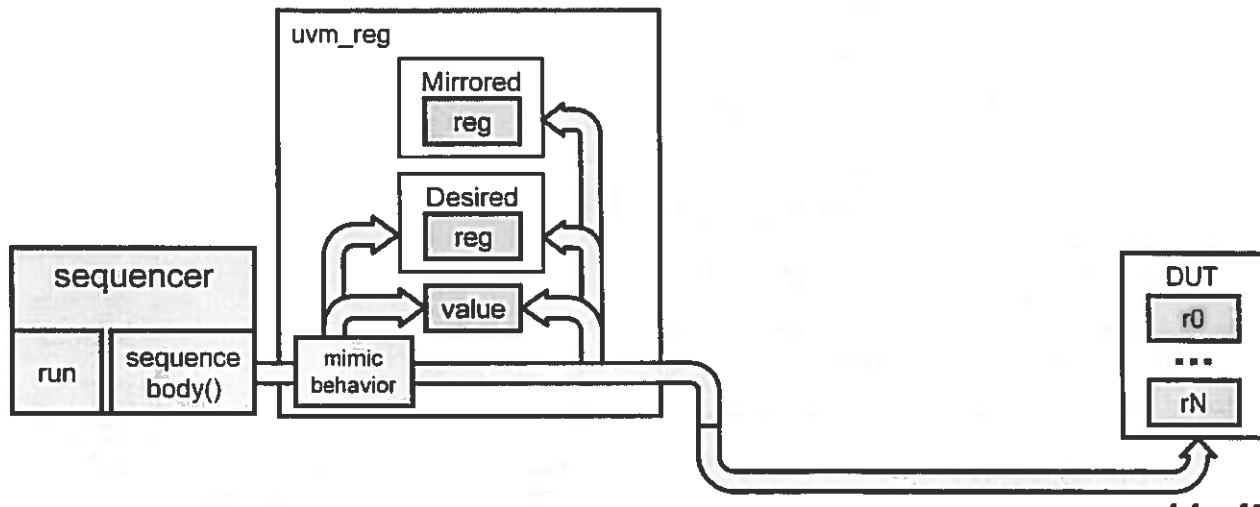
- `model.r0.read(status, value, [UVM_FRONTDOOR], .parent(this));`
- `read_reg(model.r0, status, value, [UVM_FRONTDOOR]);`
 - Sequence executes uvm_reg READ
 - uvm_reg READ is translated into bus transaction
 - Driver gets bus transaction and read DUT register
 - Read value is translated into uvm_reg data and returned to sequence
 - Mirror updates



The `read_reg()` method is a convenience method provided by the `uvm_reg_sequence` base class.

Register Backdoor Write

- `model.r0.write(status, value, UVM_BACKDOOR, .parent(this));`
- `write_reg(model.r0, status, value, UVM_BACKDOOR);`
 - Sequence write to uvm_reg with value mimicking register access policy (wc clears register)
 - uvm_reg uses DPI/XMR to set DUT register with value
 - Mirror is updated implicitly

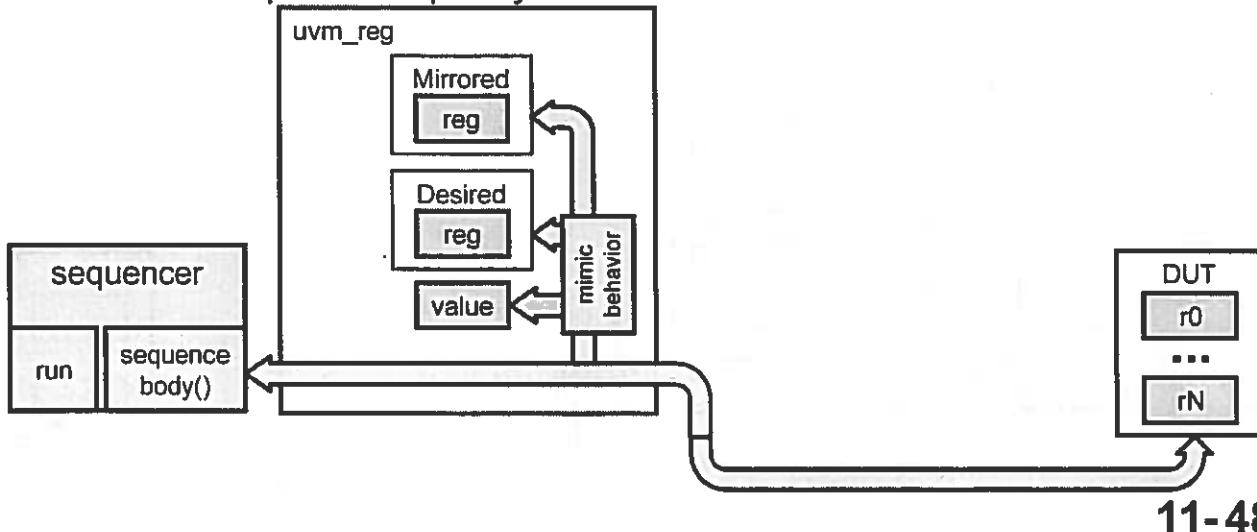


11-47

The `write_reg()` method is a convenience method provided by the `uvm_reg_sequence` base class.

Register Backdoor Read

- `model.r0.read(status, value, UVM_BACKDOOR, .parent(this));`
- `read_reg(model.r0, status, value, UVM_BACKDOOR);`
 - Sequence executes `uvm_reg` READ
 - `uvm_reg` uses DPI/XMR to get DUT register value
 - ◆ Modifies DUT register according to register policy – rc clears register
 - Mirror is updated implicitly

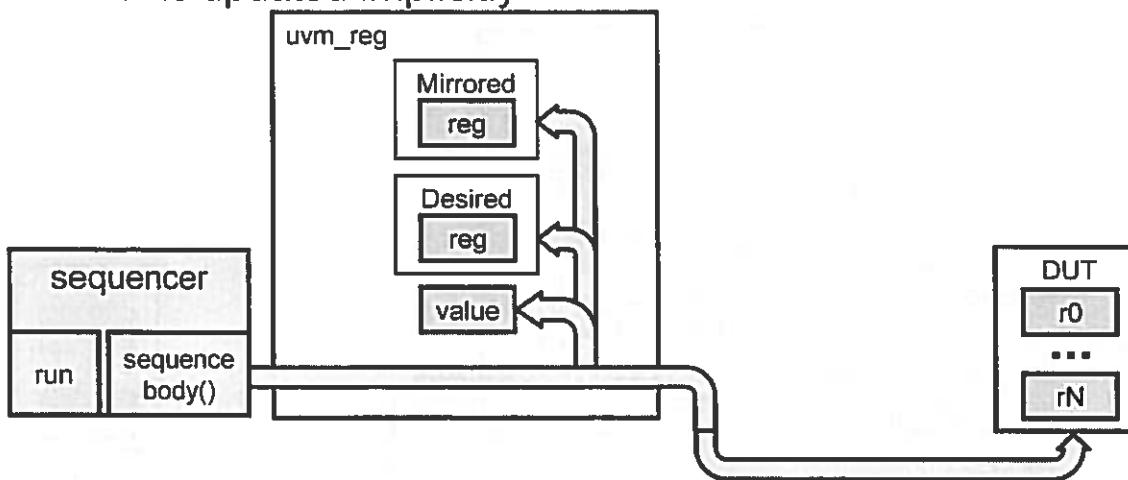


11-48

The `read_reg()` method is a convenience method provided by the `uvm_reg_sequence` base class.

Register Backdoor Poke

- `model.r0.poke(status, value, .parent(this));`
- `poke_reg(model.r0, status, value);`
 - Sequence write to uvm_reg with value as is
 - ◆ Does not mimicking register access policy (wc clears register)
 - uvm_reg uses DPI/XMR to set DUT register with value
 - Mirror is updated implicitly

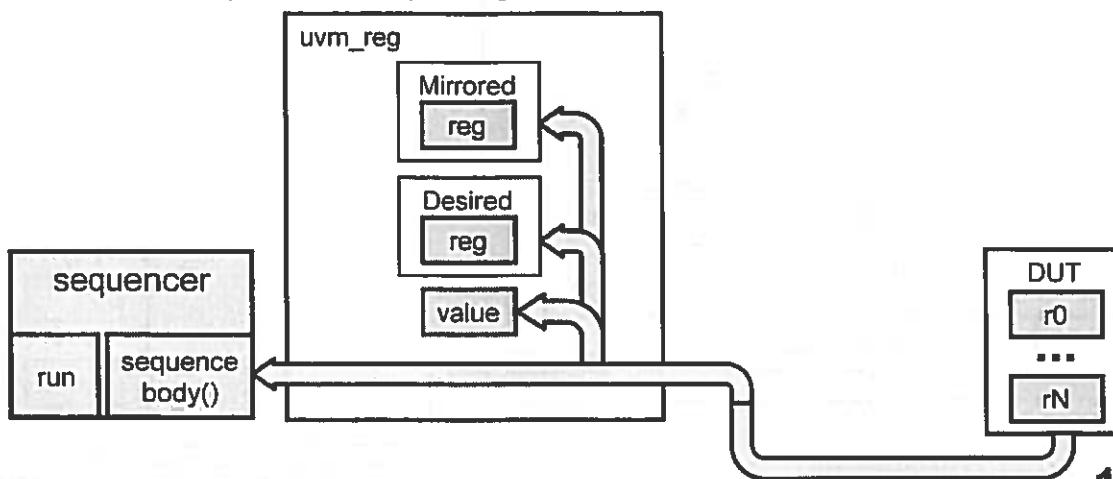


11-49

The `poke_reg()` method is a convenience method provided by the `uvm_reg_sequence` base class.

Register Backdoor Peek

- `model.r0.peek(status, value, .parent(this));`
- `peek_reg(model.r0, status, value);`
 - Sequence executes `uvm_reg` PEEK
 - `uvm_reg` uses DPI/XMR to get DUT register value as is
 - ◆ Does not mimic behavior (rc, etc.)
 - Mirror is updated implicitly

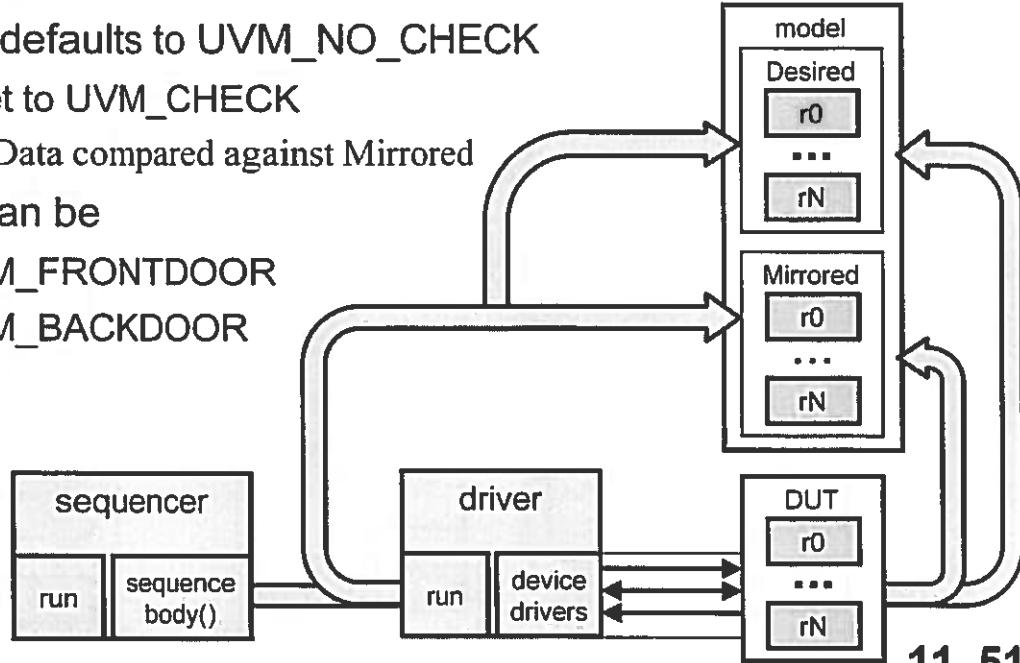


11-50

The `peek_reg()` method is a convenience method provided by the `uvm_reg_sequence` base class.

Mirrored & Desired Property Update

- `model.r0.mirror(status, [check], [path], .parent(this));`
- `mirror_reg(model.r0, status, [check], [path]);`
 - 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`

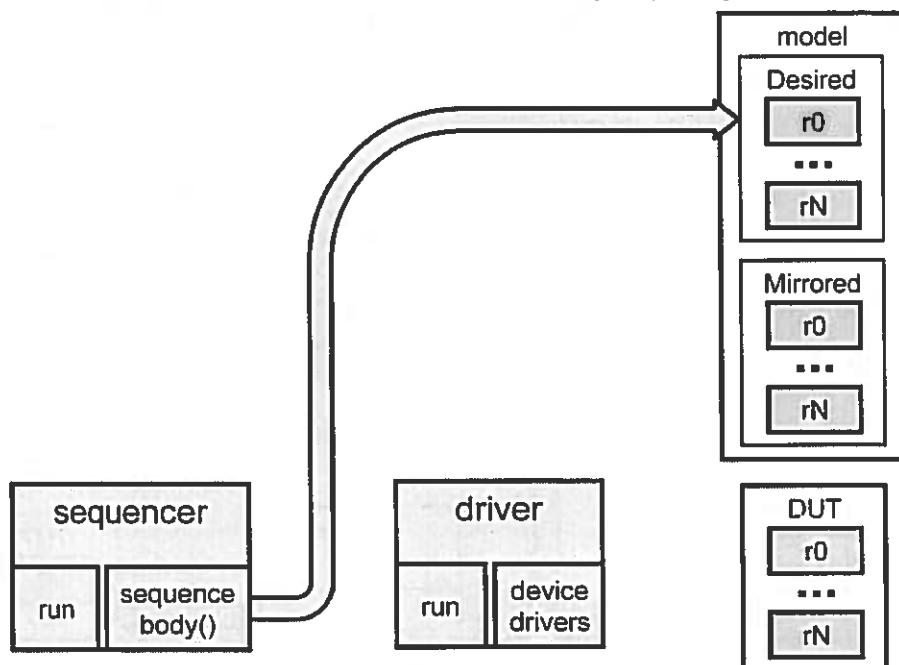


11-51

The `mirror_reg()` method is a convenience method provided by the `uvm_reg_sequence` base class.

UVM Register Desired Property Write

- ***model.r0.set(value);***
 - Set method set value in desired property

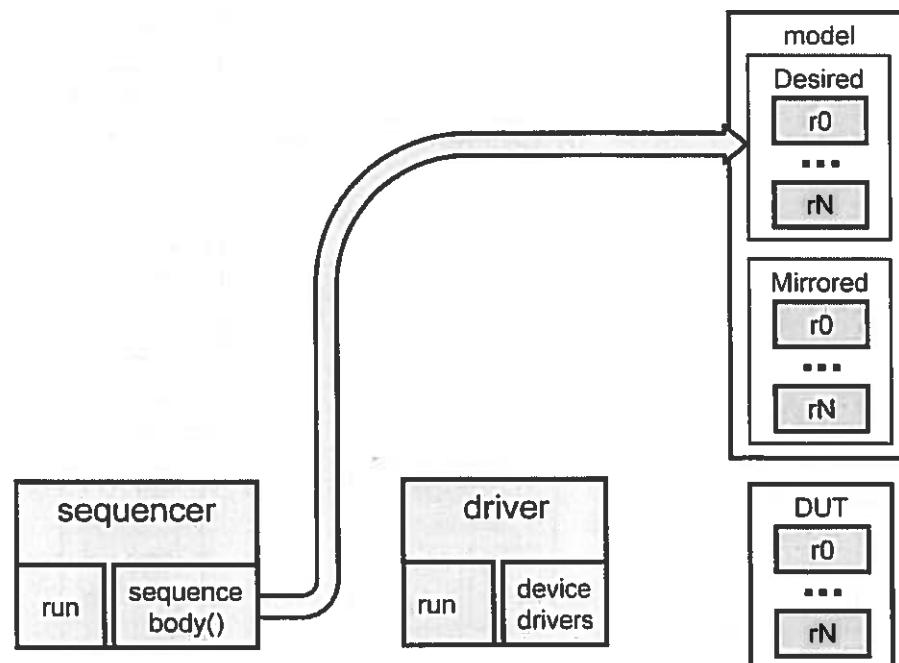


11-52

No convenience method.

Randomize UVM Register Desired Property

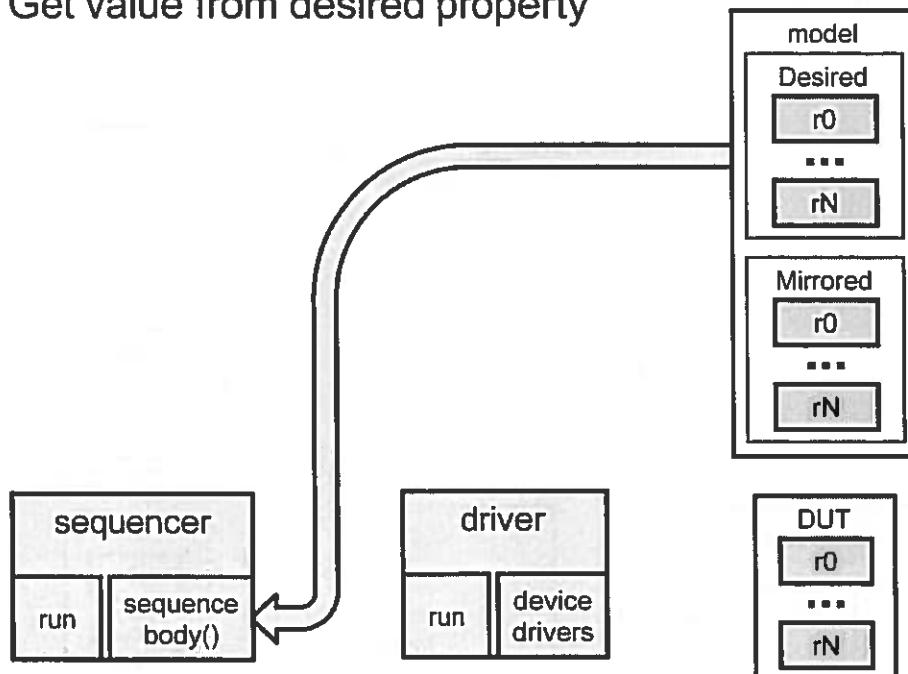
- `model.randomize();`
 - Populate desired property with random value



11-53

UVM Register Desired Property Read

- ***value = model.r0.get();***
 - Get value from desired property

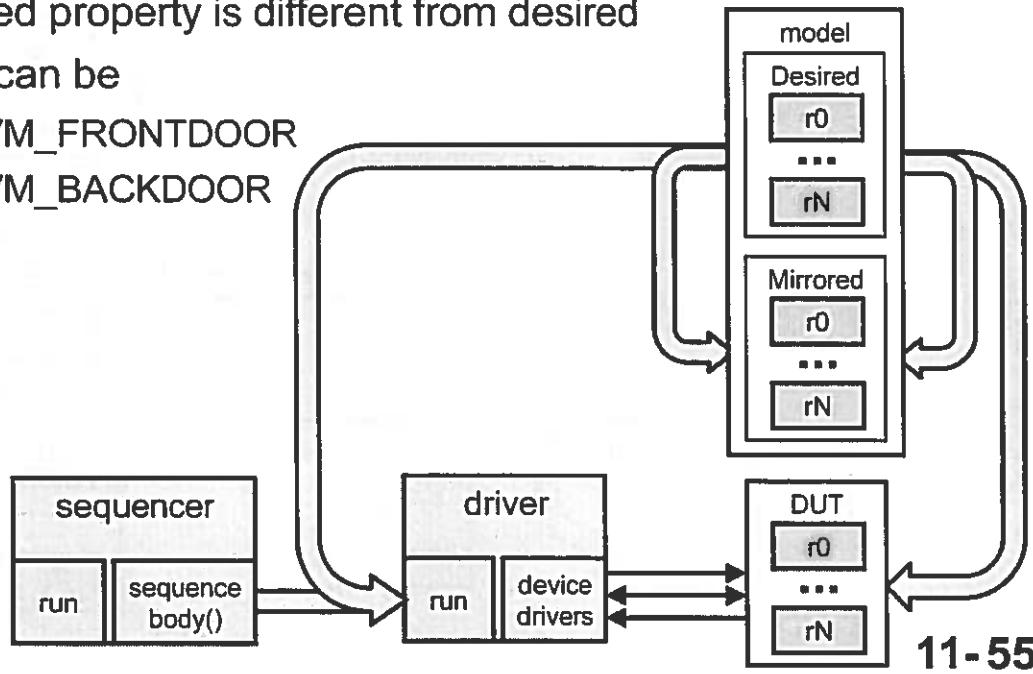


11-54

No convenience method.

Mirrored & DUT Value Update

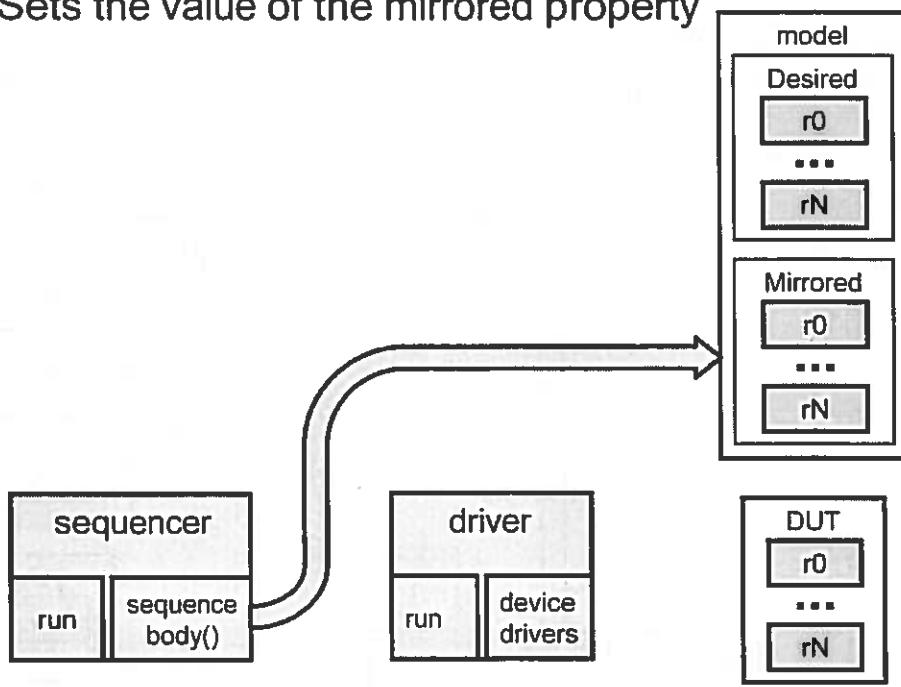
- `model.update(status, [path], .parent(this));`
- `update_reg(model, status, [path]);`
 - Update DUT and mirrored property with desired property if mirrored property is different from desired
 - `path` can be
 - ◆ UVM_FRONTDOOR
 - ◆ UVM_BACKDOOR



The `update_reg()` method is a convenience method provided by the `uvm_reg_sequence` base class.

Writing to uvm_reg Mirrored Property

- ***model.r0.predict(value);***
 - Sets the value of the mirrored property

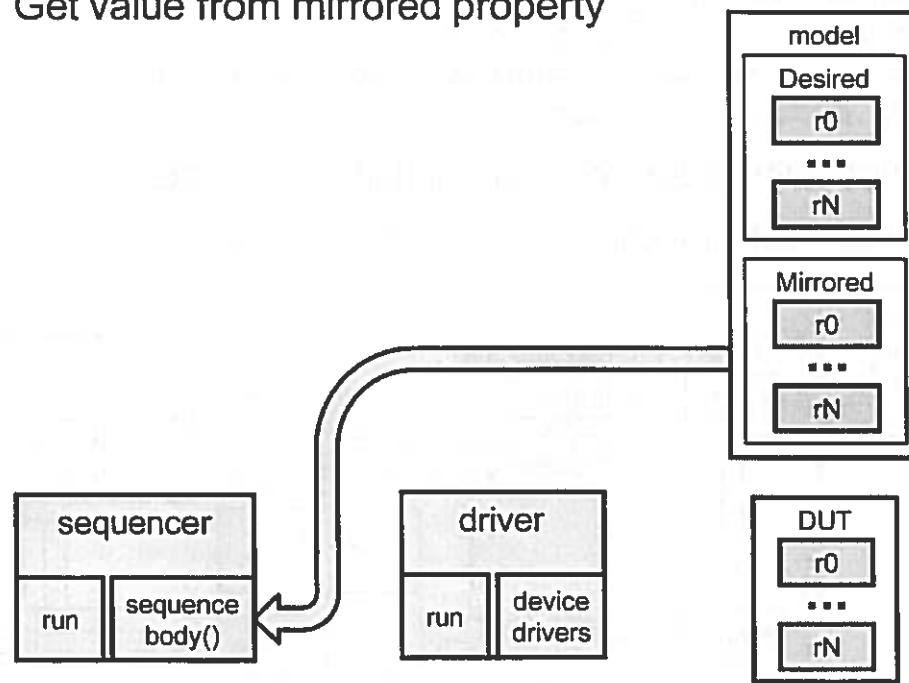


11-56

No convenience method.

Reading uvm_reg Mirrored Property

- ***value = model.r0.get_mirrored_value();***
 - Get value from mirrored property



11-57

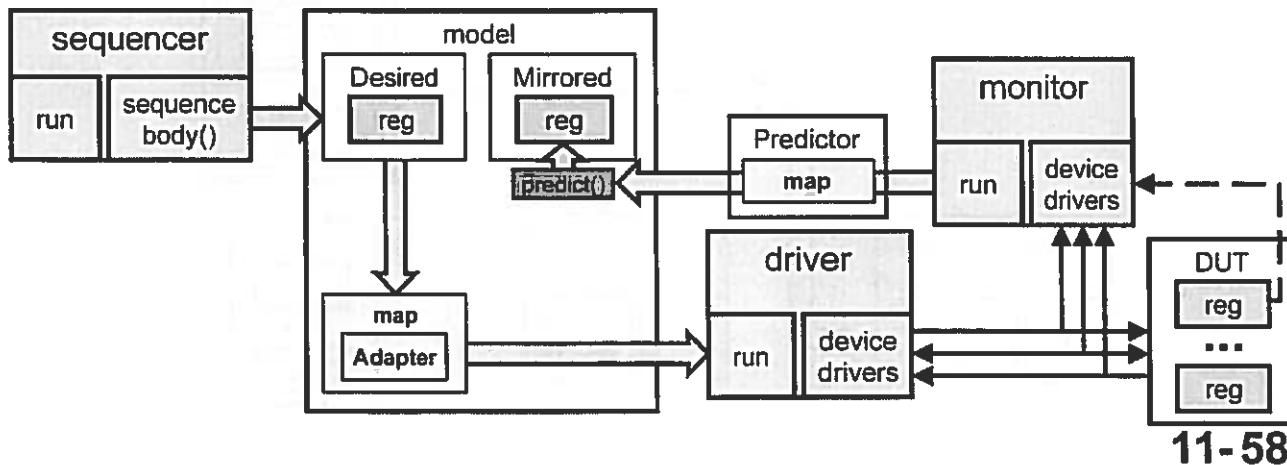
No convenience method.

Typical use of uvm_reg predict() method (1/2)

Example for explicit predictor implementation:

```
typedef uvm_reg_predictor #(host_data) hreg_predictor;
hreg_predictor hreg_predict;
// other code left off see slides on predictor
h_agent.analysis_port.connect(hreg_predict.bus_in);
regmodel.default_map.set_auto_predict(0);
```

- Monitor pass observed transaction to predictor
- Predictor calls predict() method to update mirrored property



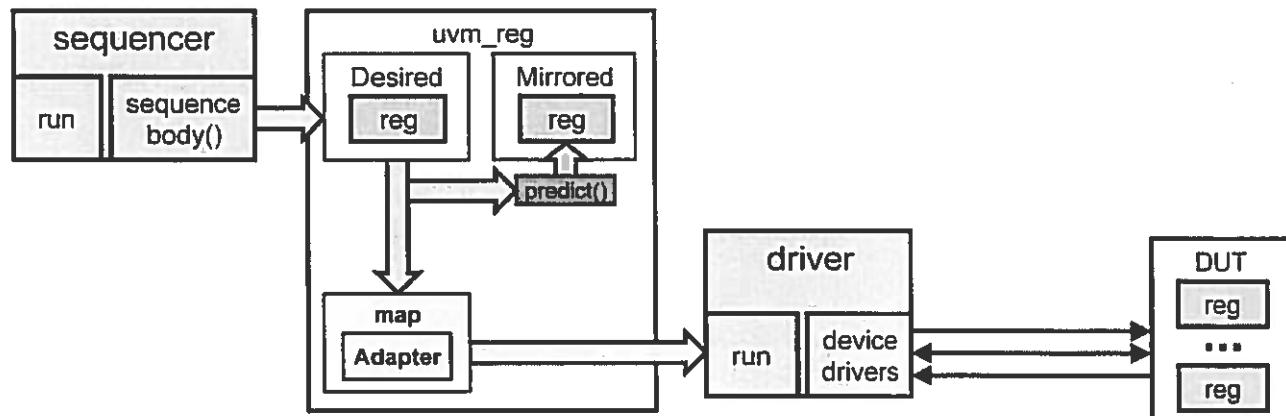
11-58

Typical use of uvm_reg predict() method (2/2)

Example for implicit predictor implementation:

```
regmodel.default_map.set_auto_predict(1);
```

- uvm_reg calls predict() method to update mirrored property directly on register activity rather observed physical changes

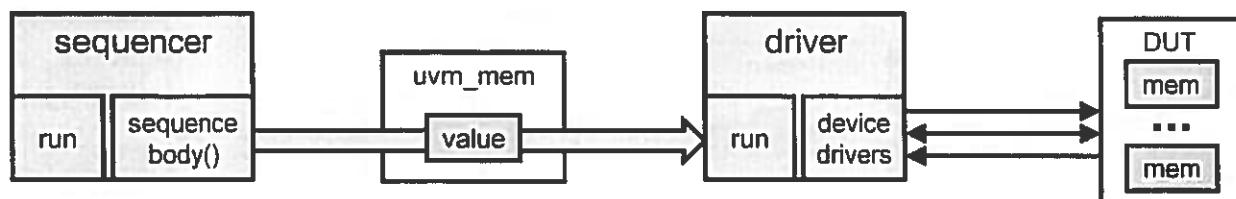


11-59

UVM Memory Modes

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
- `write_mem(model.mem, status, offset, value, [UVM_FRONTDOOR]);`
 - 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



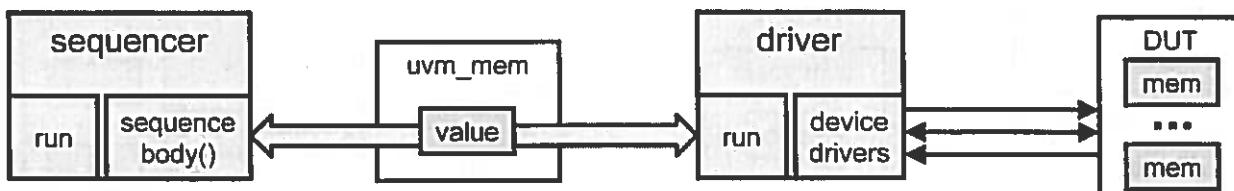
11-61

The `write_mem()` method is a convenience method provided by the `uvm_reg_sequence` base class.

No convenience method for `burst_write()`.

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
- `read_mem(model.mem, status, offset, value, [UVM_FRONTDOOR]);`
 - 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



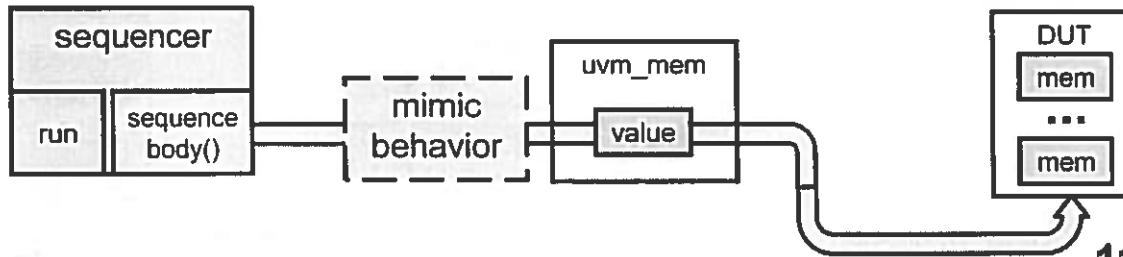
11-62

The `read_mem()` method is a convenience method provided by the `uvm_reg_sequence` base class.

No convenience method for `burst_read()`.

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
- `write_mem(model.mem, status, offset, value, UVM_BACKDOOR);`
 - 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



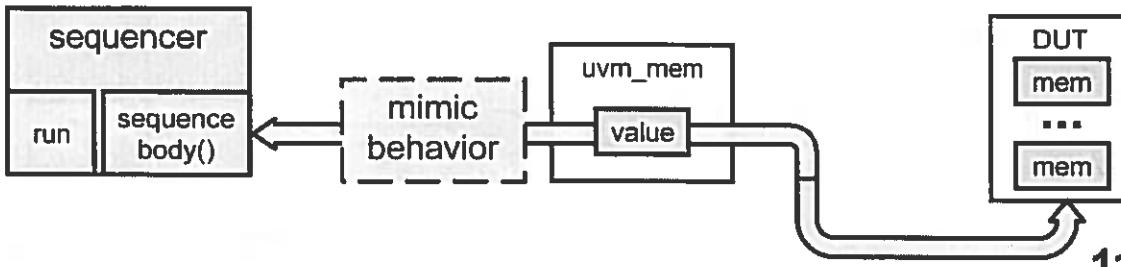
11-63

The `write_mem()` method is a convenience method provided by the `uvm_reg_sequence` base class.

No convenience method for `burst_write()`.

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
- `read_mem(model.mem, status, offset, value, UVM_BACKDOOR);`
 - 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

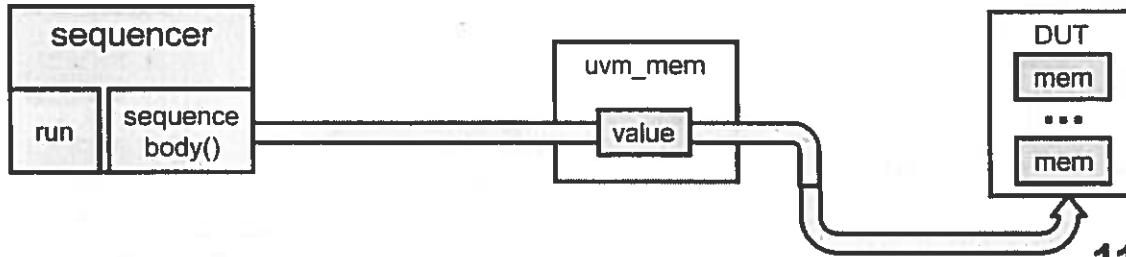


The `read_mem()` method is a convenience method provided by the `uvm_reg_sequence` base class.

No convenience method for `burst_read()`.

Memory Backdoor Poke

- `model.mem.poke(status, offset, value, UVM_BACKDOOR, .parent(this));`
- `poke_mem(model.mem, status, offset, value, UVM_BACKDOOR);`
 - Sequence writes to `uvm_mem` with value
 - ◆ Does not mimicking memory access policy (ro memory WILL be modified)
 - `uvm_mem` uses DPI/XMR to set DUT memory with value
 - Memory is not mirrored

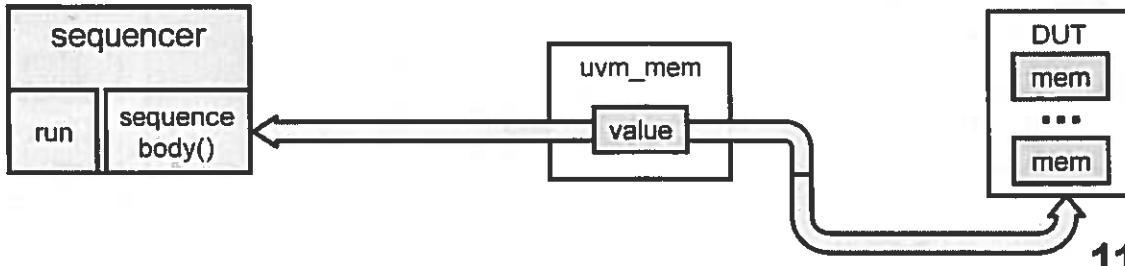


11-65

The `poke_mem()` method is a convenience method provided by the `uvm_reg_sequence` base class.

Memory Backdoor Peek

- `model.mem.peek(status, offset, value, UVM_BACKDOOR, .parent(this));`
- `peek_mem(model.mem, status, offset, value, UVM_BACKDOOR);`
 - Sequence executes `uvm_mem PEEK`
 - `uvm_mem` uses DPI/XMR to get DUT memory value
 - Memory is not mirrored



11-66

The `peek_mem()` method is a convenience method provided by the `uvm_reg_sequence` base class.

ralgen Options

ralgen Options

```
ralgen [options] -uvm -t top_model file.ralf
```

■ Common options:

- **-B**
 - ◆ Generate byte-base addressing
- **-b**
 - ◆ Generate XMR-based (non-DPI) back-door access code
- **-c a**
 - ◆ Generate the “Address Map” functional coverage model
- **-c b**
 - ◆ Generate the “Register Bits” functional coverage model
- **-c f**
 - ◆ Generate the “Field Values” functional coverage model

■ See *uvm_ralgen_ug.pdf* for other options

- Or, execute: ralgen -h

11-68

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
```

11-69

Before VCS2011.12-2, ralgen only supported data-wide word granularity.

Therefore, to accomplish “byte” granularity, one would have to shift their RALF addresses 2 bits to the right (addr = 4 in the DUT is addr = 1 in RALF, addr = 8 in the DUT is addr = 2 in the DUT, etc). The bits could be shifted left within reg2bus.

However, as of VCS2011.12-2, one can pass a “-B” option to generate a model that has “byte address granularity”.

Data-wide word granularity is still the default when running ralgen.

ralgen XMR

```
// host.ralf
register HOST_ID {...}
block host_regmodel {
    bytes 2;
    register HOST_ID (host_id) @'h0000;
}

// ral_dut_regmodel.sv
class ral_reg_host_regmodel_HOST_ID_bkdr extends uvm_reg_backdoor;
    // function new not shown
    virtual task read(uvm_reg_item rw);
        do_pre_read(rw);
        rw.value[0] = `HOST_REGMODEL_TOP_PATH.host_id;
        rw.status = UVM_IS_OK;
        do_post_read(rw);
    endtask
    virtual task write(uvm_reg_item rw);
        rw.status = UVM_NOT_OK;
    endtask
endclass

vcs +define+HOST_REGMODEL_TOP_PATH=router_test_top.dut ...
```

UVM
RAL
Classes

XMR path from harness to DUT must be specified

11-70

ralgen Functional Coverage

```
// host.ralf
register HOST_ID {
    field REV_ID {...}      ralgen -c b -uvm -t dut_regmodel host.ralf
    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};

program automatic test; // other code not shown
initial begin
    uvm_reg::include_coverage("*", UVM_CVR_ALL);
    run_test();
end
endprogram
class test_coverage extends base; // other code not shown
virtual function void end_of_elaboration_phase (uvm_phase phase);
    env.regmodel.set_coverage(UVM_CVR_ALL);
endfunction
endclass
```

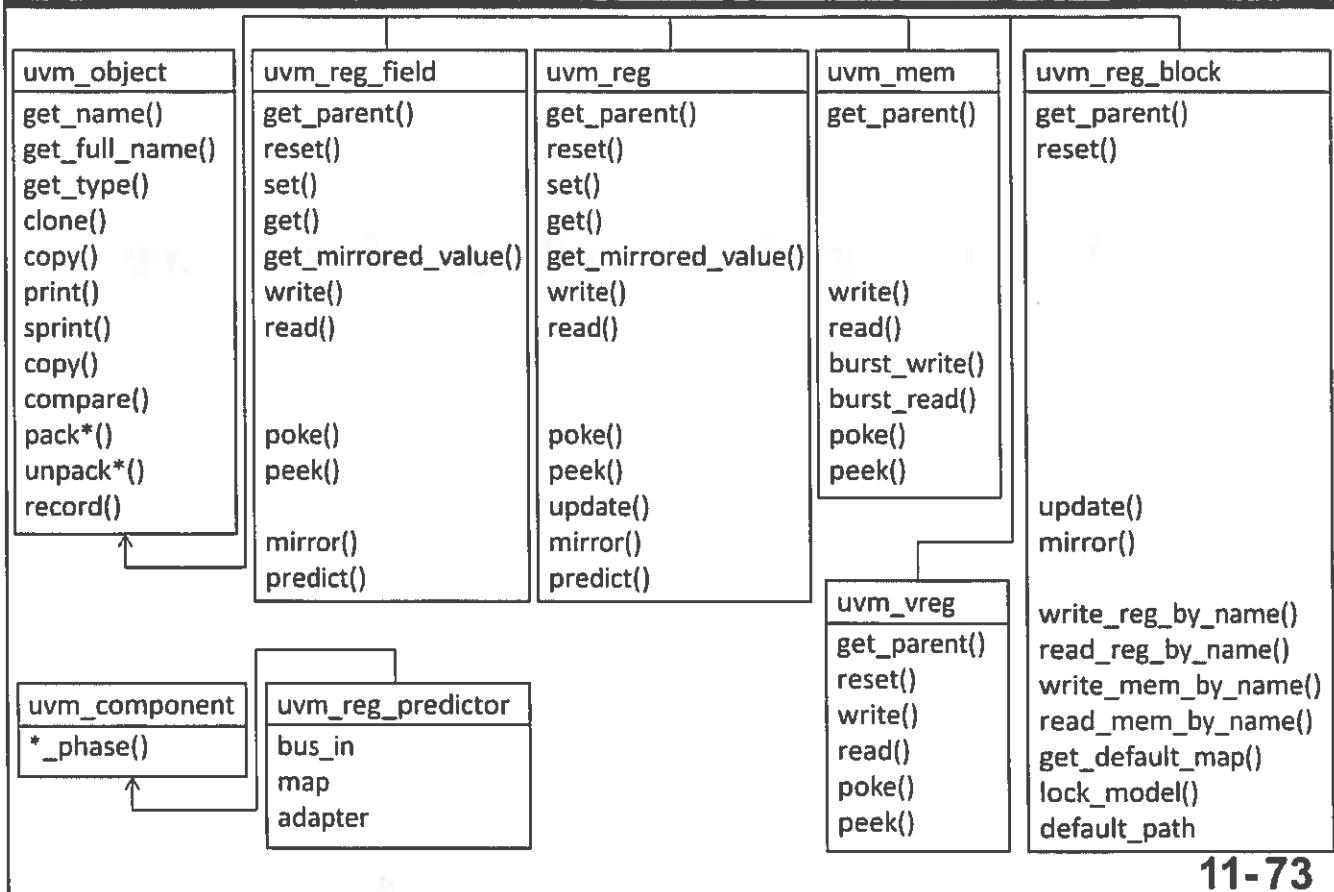
User must enable coverage with include_coverage() and set_coverage()

UVM
RAL
Classes

11-71

UVM Register Class Tree

UVM Register Base/Library Class Hierarchy



11-73

UVM Register/Memory Class Members

UVM Register Class Key Properties

```
'ifndef UVM_REG_ADDR_WIDTH
`define UVM_REG_ADDR_WIDTH 64
`endif
`ifndef UVM_REG_DATA_WIDTH
`define UVM_REG_DATA_WIDTH 64
`endif
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 ;

class uvm_reg_field extends uvm_object;
  `uvm_object_utils(uvm_reg_field)
  rand uvm_reg_data_t value; // Public property for
                            // coverage and randomization
                            // mirrored in m_desired after randomize()
  local uvm_reg_data_t m_mirrored; // Shadows what is in the DUT
  local uvm_reg_data_t m_desired; // User field
endclass: uvm_reg_field

virtual class uvm_reg extends uvm_object;
  protected uvm_reg_field m_fields[$]; // Fields in LSB to MSB order
  protected bit m_update_in_progress;
  function new (string name="", int unsigned n_bits, int has_coverage);
endclass: uvm_reg
```

11-75

UVM Memory Class Key Properties

```
class uvm_mem extends uvm_object;
  uvm_mem_mam mam;
  virtual function string get_rights(...);
  virtual function string get_access(...);
  function longint unsigned get_size();
  virtual function void get_virtual_registers(...);
  virtual function uvm_reg_addr_t get_offset(...);
  virtual function uvm_reg_addr_t get_address(...);
  virtual task write(...);
  virtual task read(...);
  virtual task burst_write(...);
  virtual task burst_read(...);
  virtual task poke(...);
  virtual task peek(...);
  virtual task do_write (uvm_reg_item rw);
  virtual task do_read (uvm_reg_item rw);
  virtual protected task backdoor_read(uvm_reg_item rw);
  virtual task backdoor_write(uvm_reg_item rw);
  `uvm_register_cb(uvm_mem, 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_mem
```

Caution: Not complete.
Only key members are shown.

11-76

uvm_reg_bus_op Definition

```
typedef struct {
    // Variable: kind - Can be: UVM_READ, UVM_WRITE, UVM_BURST_READ, UVM_BURST_WRITE
    uvm_access_e kind;
    uvm_reg_addr_t addr;
    uvm_reg_data_t data;
    // Variable: n_bits - The number of bits of <uvm_reg_item::value>
    // being transferred by this transaction.
    int n_bits;
    // Variable: byte_en - Enables for the byte lanes on the bus.
    // Meaningful only when the bus supports byte enables and the
    // operation originates from a field write/read.
    uvm_reg_byte_en_t byte_en;
    // Variable: status - Result can be: UVM_IS_OK, UVM_HAS_X, UVM_NOT_OK.
    uvm_status_e status;
} uvm_reg_bus_op;
```

11-77

UVM Register Callbacks

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

11-79

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

11-80

Changing Address Offsets of a Domain

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
```

11-82

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.

Agenda: Day 3

**DAY
3**

9 Virtual Sequence/Sequencer

10 Component Phasing Revisited



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
- Command line debug

12-2

UVM Methodology Guiding Principles

- **Top-down implementation methodology**

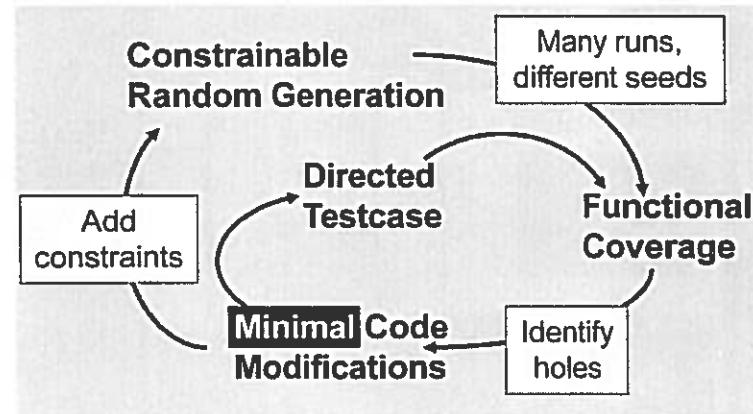
- Emphasizes “*Coverage Driven Verification*”

- **Maximize design quality**

- More testcases
 - More checks
 - Less code

- **Approaches**

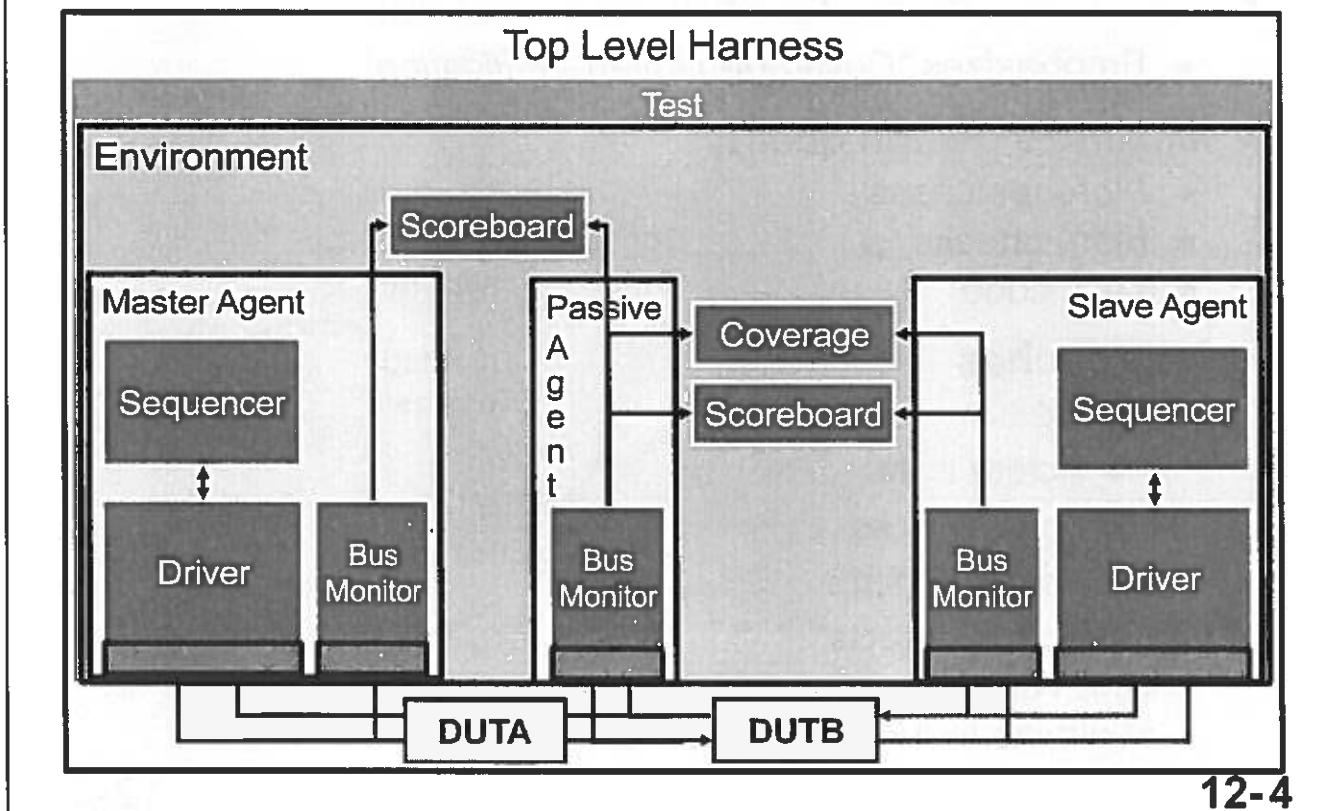
- Reuse
 - ◆ Across tests
 - ◆ Across blocks
 - ◆ Across systems
 - ◆ Across projects
 - One verification environment, many tests
 - Minimize test-specific code



12-3

Scalable Architecture

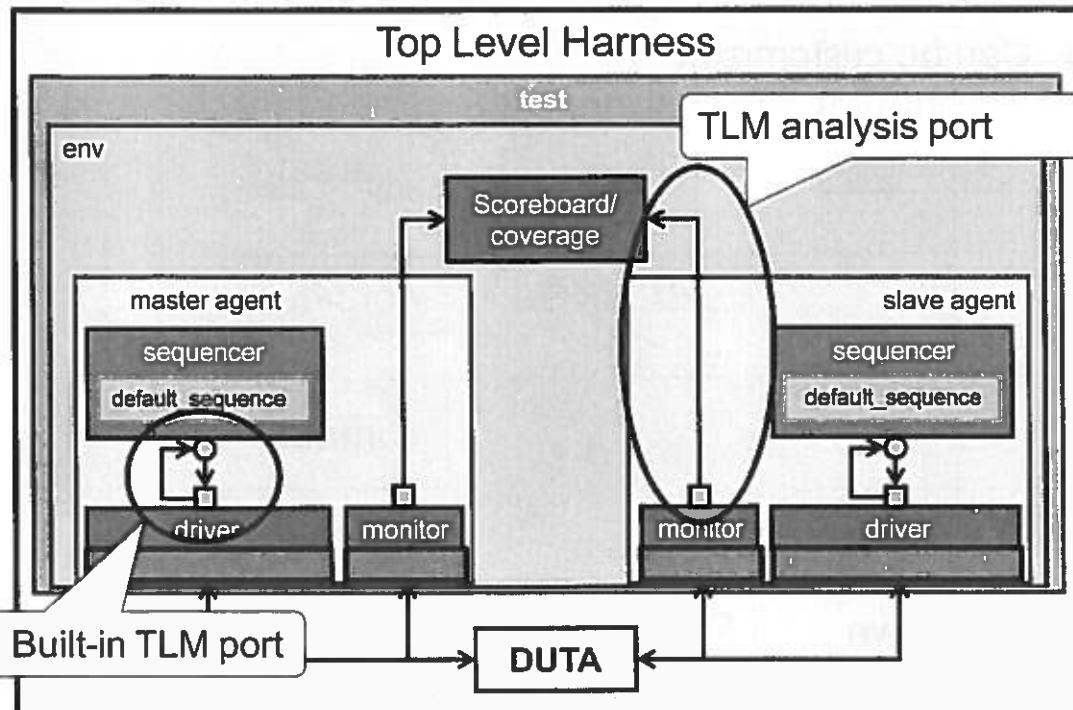
- Interface based agent enables block to system reuse



12-4

Standardized Component Communication

■ TLM, TLM 1.0, TLM 2.0



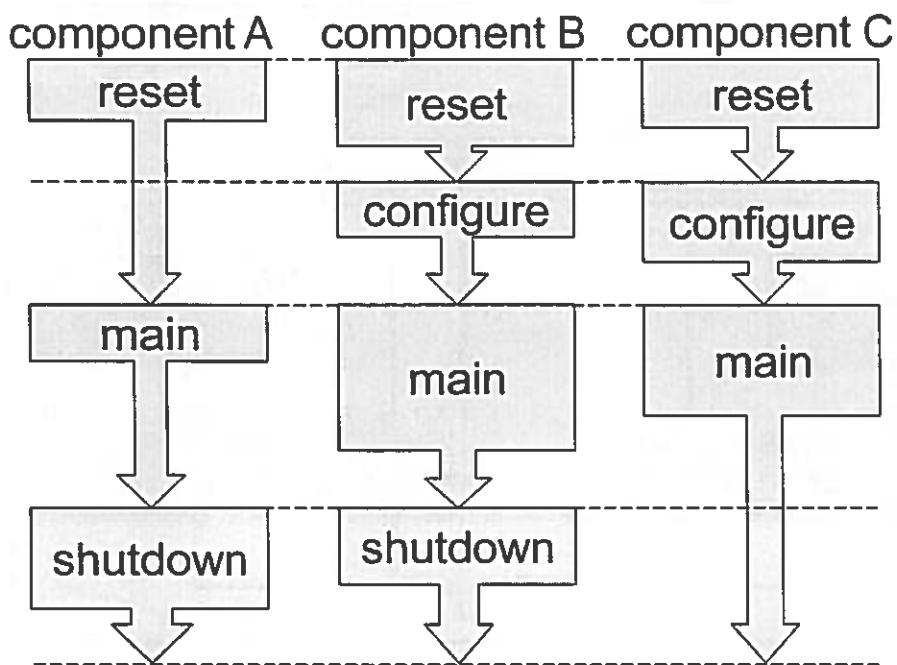
12-5

In future UVM implementation, it is likely that TLM 2.0 will be built into the base classes.

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#(S) sigs;
  function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    if (!uvm_config_db#(virtual router_tb_if)(this, "", "sigs", sigs))
      `uvm_fatal("CFGERR", "Driver DUT interface not set");
  endfunction
endclass
```

- `uvm_config_db#(_type)::set(...)`

```
class test_base extends ...; // simplified code
  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    uvm_config_db#(virtual router_tb_if)(this, "env.*", "sigs",
                                         router_test_top.sigs);
  endfunction
endclass
```

12-7

Flexible Components Configuration (2/2)

- Sequence execution phase can be specified

```
class reset_sequence extends uvm_sequence #(reset_tr);
    virtual task body();
        `uvm_do(req);
    endtask
    task pre_start(); // raise objection
    task post_start(); // drop objection
endclass
```

- **uvm_config_db#(_type) ::set(...)** executes sequence in chosen phase

```
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, "agnt.seqr.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);
    super.build_phase(phase);
    set_inst_override_by_type("env.comp", component::get_type(),
      new_comp::get_type());
  endfunction
endclass
```

Simulate with:
simv +UVM_TESTNAME=test_new

Use parent-child relationship to do search and replace for components

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

create() used to build component

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

Modify operation

```
graph TD
  simv[simv] --> uvm_top[uvm_top]
  uvm_top --> run_test[run_test()]
  uvm_top --> uvm_test_top[uvm_test_top]
  uvm_test_top --> env_top[env]
  env_top --> env[env]
  env --> new_comp[new_comp]
  new_comp --> comp[comp]
```

12-9

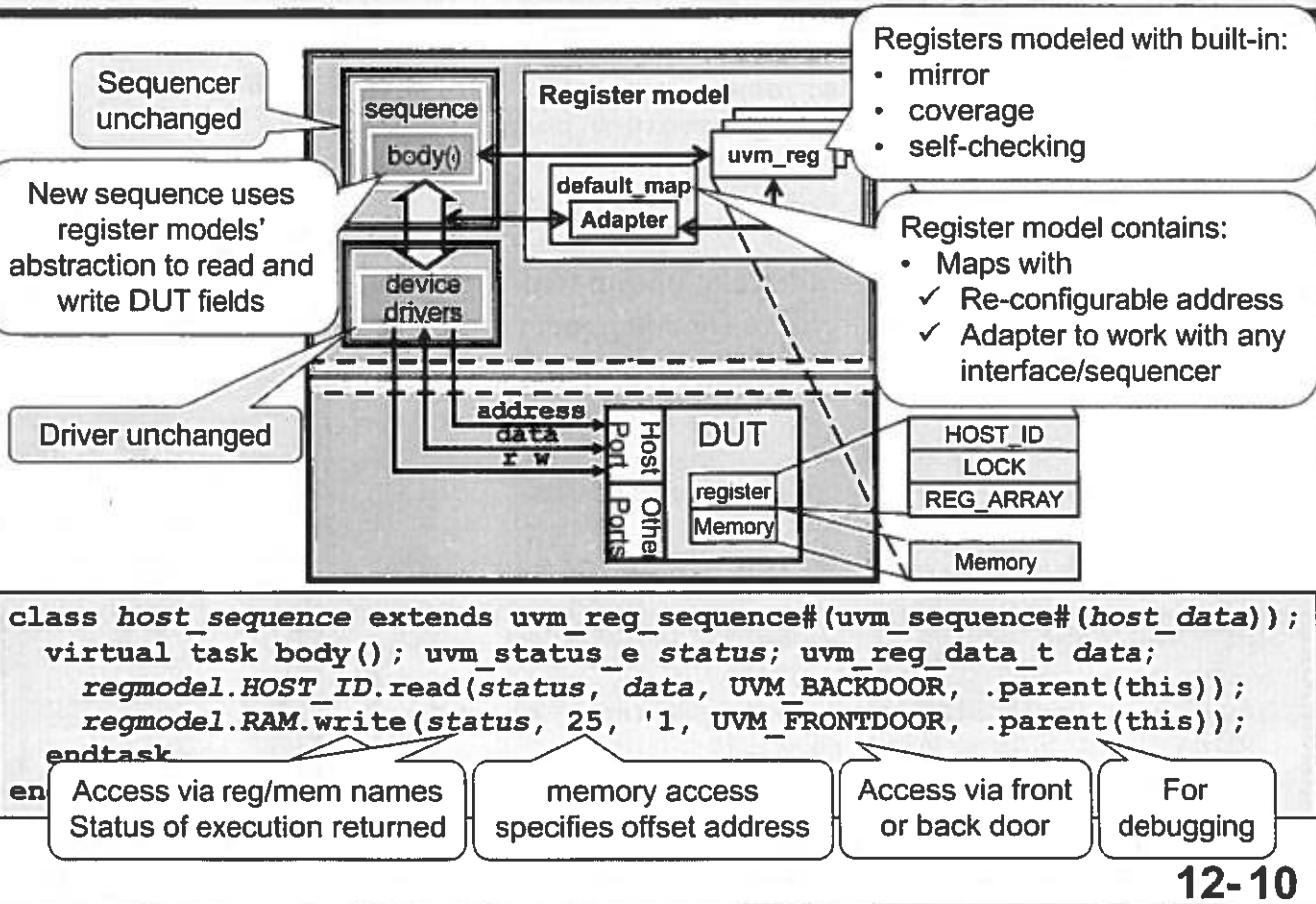
An important concept in UVM is that your testbench classes such as the environment, drivers, etc., can be written once at the start of the project, and modified very rarely if ever. This means that existing tests won't break when someone checks in a new version of a component. You can still change the 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 drivers, but it can also allow you to replace an existing component with an extended one.

In this case the *component* class has been replaced by *new_comp*. At the top level, the test tells the UVM factory to override the *component* class with *new_comp*.

Now when the environment creates a *component*, it actually gets *new_comp*.

Standardized Register Abstraction



UVM Command Line Options

- +UVM_TESTNAME
- +UVM_VERBOSITY
- +UVM_TIMEOUT
- +UVM_MAX_QUIT_COUNT
- +UVM_PHASE_TRACE
- +UVM_OBJECTION_TRACE
- +UVM_CB_TRACE_ON
- +UVM_CONFIG_DB_TRACE
- +UVM_RESOURCE_DB_TRACE
- +uvm_set_verbosity
- +uvm_set_action
- +uvm_set_severity
- +uvm_set_inst_override
- +uvm_set_type_override

12-11

Getting Help

- **UVM class reference guide:**
 - \$VCS_HOME/doc/UserGuide/pdf/UVM_Class_Reference_Manual_1.1.pdf
- **Code examples:**
 - \$VCS_HOME/doc/examples/uvm
- **Solvnet:**
 - www.solvnet.synopsys.com
- **VCS support:**
 - vcs_support@synopsys.com
- **SNUG (Synopsys Users Group):**
 - www.snug-universal.org
- **Accellera web site:**
 - <http://www.accellera.org/activities/vip>
 - Check for source code

12-12

Lab 7 Introduction

Implement RAL



60 min

Implement
register sequence
without RAL

Compile and
Simulate

Create RAL
representation

Create register
sequence with RAL

Compile and
Simulate

12-13

That's all Folks!



12-14



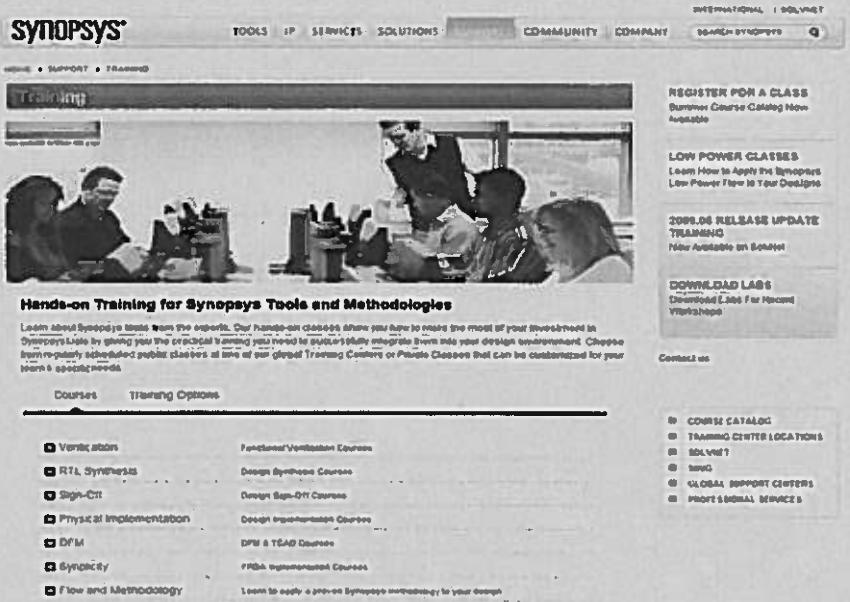
Customer Support

© 2013 Synopsys, Inc. All Rights Reserved

20130102

Synopsys Support Resources

- 1. Build Your Expertise:
Hands-on Training
for Synopsys Tools and
Methodologies**
Synopsys.com/Support/Training
 - ◆ Workshop schedule
and registration
 - ◆ Download materials
(*SolvNet id* required)
- 2. Empower Yourself:
solvnet.synopsys.com**
 - ◆ Online technical
information and access to
support resources
 - ◆ Documentation & Media
- 3. Access Synopsys Experts:
Support Center**



The screenshot shows the Synopsys Training website. At the top, there's a navigation bar with links for TOOLS, IP, SERVICES, SOLUTIONS, COMMUNITY, and COMPANY. A search bar is also present. On the left, there's a sidebar with links for REGISTER FOR A CLASS, LOW POWER CLASSES, 2010.0 RELEASE UPDATE TRAINING, DOWNLOAD LABS, COURSE CATALOG, TRAINING CENTER LOCATIONS, SOLVNET, MMG, GLOBAL SUPPORT CENTERS, and PROFIT SIGNAL SERVICES. The main content area features a banner for "Hands-on Training for Synopsys Tools and Methodologies". Below the banner, there's a section titled "Learn about Synopsys tools from the experts. Our hands-on classes allow you have to make the most of your investment in Synopsys tools by giving you the practical training you need to successfully integrate them into your design environment. Choose from regularly scheduled public classes at one of our global Training Centers or Private Classes that can be customized for your team & specific needs." There are two tabs: "Courses" and "Training Options". Under "Courses", there are several categories with sub-links: Verifications, Functional Verification Courses; RTL Synthesis, Design Synthesis Courses; Sign-off, Design Sign-off Courses; Physical Implementation, Design Implementation Courses; DFM, DFM & TSMC Courses; Synopsys, PIBA Implementation Courses; and Flow and Methodology, Learn to apply a pre-on Synopsys methodology to your design.

CS-2

SolvNet Online Support Offers

- Immediate access to the latest technical information
- Product- and Release-Specific Update Training
- Thousands of expert-authored articles, Q&As, scripts and tool tips
- Open a support case
- Release information
- Online documentation
- License keys
- Electronic software downloads
- Synopsys announcements (latest tool, event and product information)

The screenshot shows the SolvNet search interface. At the top, there's a navigation bar with links for SOLVNET HOME, SYNOPSYS.COM, FEEDBACK, SITE MAP, HELP, and SIGN OUT. Below that is a sub-navigation bar with links for HOME, SOLVNET SEARCH, DOCUMENTATION, SUPPORT, DOWNLOADS, TRAINING, METHODOLOGY, MY PROFILE, and SEARCH SOLVNET. A search bar is present with a magnifying glass icon. To the right, there are several promotional boxes: one for MIPS Analog IP Support, another for a free "VMM for Low Power" book, a discover Zroute link, reference methodology scripts, and a SHUG 2009 registration link. The main content area is titled "New / Updated Articles" and shows a list of recent articles. One article is highlighted: "DRC Violation not reported for an uncontrollable clock during capture cycle" (08-25-2009). Other visible article titles include "Maximizing IC Validator Performance When Running on Small Cells" (08-25-2009), "Using Scaling and Link_paths_per_instance Variable in Low-Power UPF Timing and Power Analysis in PrimeTime PX" (08-25-2009), "Finding the Percentage of Flip Flops That Are Not Clock Gated" (08-24-2009), "How is the Capacitance in a Driver Load Extrapolated RC-Off Warning Calculated?" (08-24-2009), and "Script to retrieve delay between two pins on a timing path" (08-24-2009). At the bottom of the article list, there's a link to "More New Articles".

CS-3

SolvNet Registration is Easy

1. Go to solvnet.synopsys.com/
ProcessRegistration
2. Pick a username and password.
3. You will need your “Site ID” on
the following page.
4. Authorization typically takes
just a few minutes.



The image shows two screenshots of the Synopsys New User Registration process. The top screenshot shows the initial registration page with fields for Corporate Email, Username (minimum of 4 characters, lowercase only), First Name, and Password (must be at least 8 characters long, include at least 1 number, 1 symbol, and 1 uppercase letter). The bottom screenshot shows the continuation of the registration process, where the user is prompted to enter their Active Site ID and can add another site. Both screenshots include the Synopsys logo and navigation links for sign-in, help, and privacy policy.

New User Registration

Your Corporate Email _____

Username _____
(minimum of 4 characters; a-z lowercase only) [0-9]

First Name _____

YOUR password must meet the following criteria:
• Must be at least 8 characters long
• Must include at least 1 number
• Must include at least 1 symbol character (non-letter or number, such as %, *, #)
• Must include at least 1 uppercase letter

Synopsys Sign In

New User Registration

Important: Please Read Before Registering.

To access Synopsys protected applications, you **must** provide an **Active Site ID** in the field below.

Synopsys Site ID Add Another Site

© 2009 Synopsys Inc. All Rights Reserved

CONTACT US | PRIVACY POLICY

CS-4

Support Center: AE-based Support

- **Industry seasoned Application Engineers:**
 - 50% of the support staff has >5 years applied experience
 - Many tool specialist AEs with >12 years industry experience
 - Access to internal support resources
- **Great wealth of applied knowledge:**
 - Service >2000 issues per month
- **Remote access and debug via ViewConnect**

The screenshot shows the Synopsys Global Support Centers page. At the top, there's a navigation bar with links for TOOLS, IP, SERVICES, and SOLUTIONS. Below that is a main menu with links for HOME, SUPPORT, and GLOBAL SUPPORT CENTERS. A large banner features a woman in a lab coat and text about expert support. To the right, a box says "Contact us: Open a support case". Other sections include "OPEN A SUPPORT CASE", "RELEASE NOTIFICATIONS", "2008.12 RELEASE UPDATE TRAINING", and "MIPS ANALOG IP SUPPORT". At the bottom, there are links for "GLOBAL TRAINING CATALOG", "SOLVNET", and "SERVICES".

www.synopsys.com/support

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):**
 - <http://www.synopsys.com/Community/SNUG/>

CS-6

Summary: Getting Support

- Customer Education Services
- SolvNet
- Support Center
- SNUG

CS-7

This page was intentionally left blank.