Vertical & Horizontal Reuse of UVM Environments

Mark Litterick
What is Verification Reuse?

“Code reuse, is the use of existing software, or software knowledge, to build new software” Wikipedia

• Verification Reuse:
  – achieving verification goals through effective code reuse
• Specifically:
  – reuse verification components between environments
  – reuse verification components between hierarchies
  – reuse verification environments between projects
  – also reuse of base-classes, patterns & methodology
• Several different paradigms:
  – referred to as horizontal and vertical reuse
Horizontal Reuse

- Reusing verification code *without changing role*
  - similar responsibilities, structure & use-case
- Reuse *stimulus, checks, coverage & messages*
  - same stimulus API presented to user or test writer
- **Horizontal reuse** usually means *predictable use-case*
  - typically achieved by configuration, adaption and flexibility
  - code is designed for more than one known use-case
Horizontal Reuse examples

• Reuse **verification component** in multiple envs

• Reuse **verification environment** in multiple projects
Vertical Reuse

• Reusing verification code with a different role
  – different responsibilities, structure & use-case

• Reuse stimulus, checks, coverage & messages, but...
  – stimulus nested or layered inside high-level scenario
    (with different user API)
  – or stimulus not reused at all
    (but checks, coverage & messages are)

• Vertical reuse usually means unknown use-cases
  – typically achieved by encapsulation and extensibility
  – code is designed to allow the user to do other things
Vertical Reuse Examples

- **Reuse passive** block-level in **top-level** environment

  ![Diagram showing passive block-level reuse in top-level environment](image)

  **DIFFERENT STRUCTURE, NO STIMULUS**
  (STIMULUS COMES FROM ANOTHER UVC)

- **Reuse active** component in **top-level context**

  ![Diagram showing active component reuse in top-level context](image)

  **DIFFERENT ROLE, STIMULUS IS PART OF**
  HIGH-LEVEL SCENARIO *(NOT SO RANDOM)*
Reuse in UVM

- **UVM enables reuse**, but does *not guarantee* it
  - SystemVerilog **object-oriented** program paradigm (OOP)
  - UVM provides reusable **base-classes & methodology**
  - UVM reuses standard **software patterns** for utilities:
    factory, configuration database, event pools, phases, ...
- **Reuse affects all aspects** of environments, including:
  - architecture
  - configuration
  - stimulus
  - checks
  - coverage
  - modeling
Vertical & Horizontal Reuse of UVM Environments

Demystifying the UVM Configuration Database

Configuration Object Encapsulation & Appropriate config_db Usage

Parameterized Classes, Interfaces & Registers

Behind the Scenes of the UVM Factory

Effective Stimulus & Sequence Hierarchies

Adaptive Protocol Checks - Config Aware Assertions

Self-Tuning Functional Coverage - Strategy & Implementation

Advanced UVM Register Modeling & Performance

Tutorial Content

DVCon EU 2014

DVCon EU 2015
Additional UVM Concerns For

VERTICAL REUSE
Top-Level Verification Requirements

• **Top-Level** has different **concerns** to block-level
  – functional **correctness** & **performance** of full **application**
  – interaction of modules, sub-systems & **shared resources**
  – operation with all **clock**, **power** & **test** domains
  – **connectivity** of all blocks & sub-systems

• Cannot **achieve closure** on all these by looking only at external pins in a complex top-level SoC
  – require **checks**, **coverage** & **debug messages** at all levels

**NOT ENOUGH TO PROVE OPERATION OF THE PERFECT CHIP**
(ALSO NEED TO **ISOLATE & DEBUG FAILURES** EFFECTIVELY)
Vertical Reuse

• Three observations:
  – vertical reuse is hard to prepare for due to change of role and unknown use-cases
  – top-level environment often must be developed in parallel with block-level
  – vertical reuse is effort for implementer and only adds value for the re-user

• Reality check:
  – don’t try to second guess everything the user might need
  – get structure right, validate operation & allow for adaption

REUSE GUIDELINES | RETROFITTING REUSE
Active/Passive Operation

• Active components provide or affect the stimulus driven to the DUT, influencing the stimulus flow

• Passive components do not provide or affect the stimulus in any way, they just observe

• Active/Passive mode affects:
  – run-time architecture
  – functional capability
  – error tolerance
  – debug capability
Active Block-Level

- **Stimulus** is provided by sequencers and drivers
  - *proactive master* generates request stimulus
  - *reactive slave* generates response stimulus
- **Checks** performed by interfaces, monitors & scoreboards
- **Coverage** is collected by monitors & scoreboards
- **Messages** are generated by all components
Passive Block-Level

- **No stimulus** is performed by passive components – sequencers and drivers are not even present
- **Checks** are still performed by interfaces, monitors & scoreboards
- **Coverage** is still collected by the monitors & scoreboards
- **Messages** are generated by the remaining components
Passive Reuse in Top-Level

ALL CHECKS, COVERAGE & MESSAGES REUSED TO ENSURE:
• EFFECTIVE VALIDATION OF BLOCK IN TOP-LEVEL CONTEXT
• COMPREHENSIVE TOP-LEVEL OPERATION AND DEBUG
Active/Passive Misuse

- low-level UVC OK but env ignores active/passive
- drivers post sequence items to scoreboard
- drivers doing functional timeout checks
- coverage defined in active components
- configuration updates from sequence or driver
- important messages from drivers
- error injection traffic reported as a warning
- passive components control end-of-test schedule
- uncontrollable end-of-test scoreboard checks
- ...

© Verilab & Accellera
Active/Passive Guidelines

- Complete env must consider active/passive
- Do **not** connect scoreboards to active components
- Perform functional checks in passive comps
- Collect functional coverage in passive comps
- Generate important messages in passive comps
- Update configuration only from passive comps
- Promote warnings to errors in passive mode
- Do **not** control end-of-test from passive comps
- Allow **disable** for scoreboard end-of-test check

**(OBVIOUSLY) ACTIVE COMPONENT IS NOT PRESENT IN PASSIVE MODE**

**(NOT SO OBVIOUSLY) ACTIVE STIMULUS IS ALSO **NOT** PRESENT**

Update pseudo-static config:
- **Not** sequence, but monitor
- **Not** reg write, but predict

**INTERNAL BUS MAY STILL BE ACTIVE WHEN TOP SCENARIO COMPLETES**

**TOP-LEVEL SCENARIO DECIDES WHEN TO END THE TEST**

*INTERNAL BUS MAY STILL BE ACTIVE WHEN TOP SCENARIO COMPLETES*

*INTERNAL BUS MAY STILL BE ACTIVE WHEN TOP SCENARIO COMPLETES*
Problems of Scale

• **Top-level** environment is already **complex**
• Integrating *many block-level* environments introduces additional requirements:
  – build and integration **encapsulation**
  – configuration containment and **consistency**
  – namespace isolation and **cohabitation**

**BLOCK-LEVEL SUPPLIERS NEED TO MAKE THINGS AS EASY AS POSSIBLE TO INTEGRATE AND REUSE**
Problems of Scale

- Environment pulled together only in base-test
- Multiple dispersed config objects and fields
- Multiple agent interfaces required for top-level
- Macro definitions have global scope
- Enumeration literals and types without prefix
- Inflexible inappropriate low-level coverage
- Incorrect message verbosity resulting in clutter
- ...

© Verilab & Accellera
**Scale Guidelines**

- **encapsulate** all sub-components in `env` not test
- use **hierarchical configuration** objects for `env`
- combine multiple I/Fs into **hierarchical interfaces**
- **avoid namespace collisions** by using scope prefix
- encapsulate **coverage** in **separate** class for overload
- apply more rigorous **message verbosity** rules
- ...

**TEST IS NOT REUSED**
**BUT ENVIRONMENT IS**

**MESSAGE CLUTTER UNACCEPTABLE**
**WITH A LOT OF REUSED BLOCK ENVs**

**USER ONLY SETS TOP-CONFIG**
**WHICH PROPAGATES DOWN**

**USER ONLY HAS ONE INTERFACE TO**
**INSTANTIATE & ADD TO CONFIG_DB**
Un-Encapsulated Reuse

MANY INCONSISTENT CONFIG OBJECTS & FIELDS TO MAINTAIN

MANY SMALL INTERFACES TO INSTANTIATE & SET IN CONFIG_DB

MANY COMPONENTS TO INSTANTIATE & CONNECT

UN-ENCAPSULATED REUSE IS ERROR-PRONE, HARD TO MAINTAIN & TOO MUCH EFFORT
Encapsulated Reuse

PROPER ENCAPSULATION MAKES VERTICAL REUSE OF MANY BLOCK-LEVEL ENVS A FEASIBLE PROPOSITION
What is the Cost of Reuse?

- Additional **effort** in **verification components**
  - architectural concepts are provided by methodology
  - shortcuts & quick fixes save time in initial project...
  - ...but in the long term all bad coding style costs money
  - effort here **benefits** both **supplier** and **reuser**
- Additional **effort** in **verification environments**
  - packaging environment, config, interfaces all costs effort
  - effort here **only** really **benefits reuser**
- Total **cost** is relatively **low** but **not zero**

**REUSE DOES NOT COME FOR FREE – YOU MUST DO SOMETHING**!
Reality Check

• Observation: since reuse does not come for free...
  • ... it will not be correctly implemented in first instance!
• Specifically:
  • first project implementing block-level has other priorities
  • top-level project needs to start in parallel with block-level
  • second project using the block should factor in the costs
  • trade-off reuse costs versus design from scratch
  • trade-off reuse costs verses quality & debug improvements
  • architectural fixes for reuse should be retrofitted to source (since working regression provides the best cross-check)
What is Retrofitting?

“Retrofitting refers to the addition of new technology or features to older systems” Wikipedia

• Retrofitting Reuse:
  • adding reuse capability to an existing component or environment that does not yet support reuse

• Specifically:
  • fixing a verification component to comply with guidelines
  • re-architecting a block-level environment for vertical reuse
  • ...attempting reuse of an environment for the first time!
Strategy for Retrofitting

- First, determine **what** you have
  - probably supplied documentation will not be good enough
  - print & review the topology, config and factory settings
- Next, determine **scope** of rework
  - active/passive build control or major scoreboard reconnect?
- Implement **changes** in block-level environment
  - run **passing** block-level **regressions** throughout
  - validate **passive** operation at block-level with **shadow** instance
- **Use** in the top-level environment
  - ensure top-level continues to run with **no bad side-effects**
  - ensure block-level checks, coverage and messages work

**VALIDATE PASSIVE MODE USING BLOCK-LEVEL REGRESSIONS**
Passive Shadow

**TWO INSTANCES OF THE SAME ENVIRONMENT**
ONE IN **ACTIVE** MODE, ONE IN **PASSIVE** MODE

**SHADOW PASSIVE ENVIRONMENT**

**NORMAL ACTIVE ENVIRONMENT**

**BLOCK-LEVEL BASE TEST**

**PROVE FUNCTIONALITY USING A PASSIVE SHADOW ENV**
FOR **UP-FRONT** OR **RETROFITTED** PASSIVE OPERATION

© Verilab & Accellera
References and Further Reading

• “Advanced UVM Tutorial”
  Verilab, DVCon Europe 2014, www.verilab.com/resources

• “Advanced UVM Register Modeling – There’s More Than One Way To Skin A Reg”

• “Pragmatic Verification Reuse in a Vertical World”
  M.Litterick, DVCon 2013, www.verilab.com/resources
Questions