

UVM Audit Tutorial

Assessing UVM Testbenches to Expose Coding Errors & Improve Quality

Presenter: Mark Litterick

Contributors: Jonathan Bromley, Jason Sprott, Tamás Simon



Outline

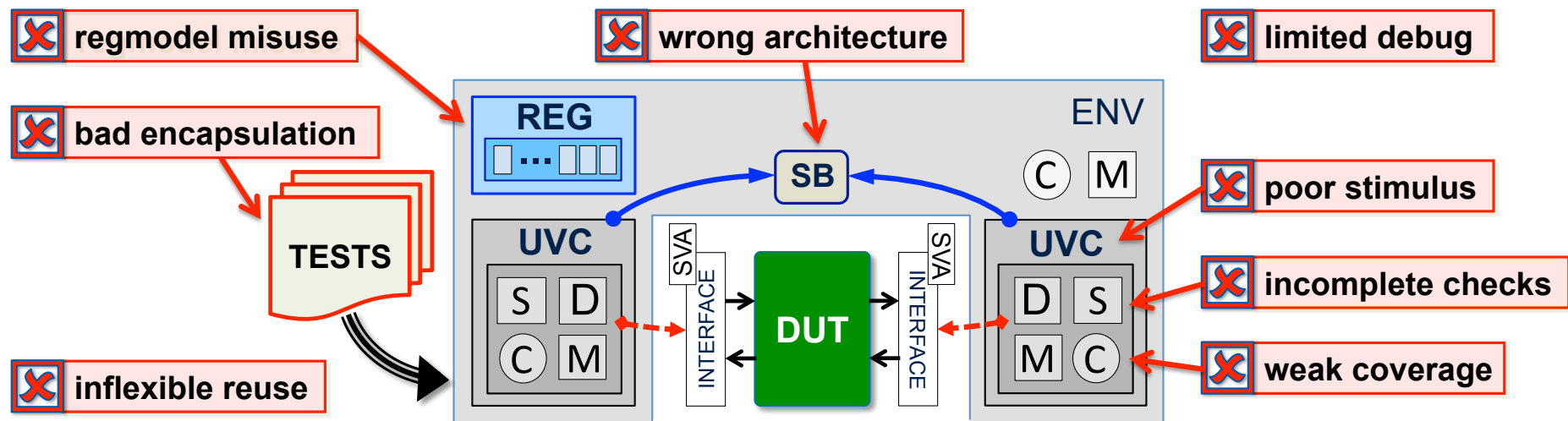
- **Introduction & background**
- **Development process** - overview
- **Code analysis** – detailed code & architecture
- **Digging deeper** – experimenting with code-base
- **Missing code** – looking for what is not there
- **Reporting findings** - overview
- **Conclusion & references**

What is an audit and why bother

INTRODUCTION

Background

- Many verification environments **claim** to follow **UVM best practice**...
 - but don't stand up to scrutiny: **increasing** project **effort**, **time**, **cost** and **risk**




Observations based on:

- real **projects**, different **clients**, diverse **applications**

What is an audit?

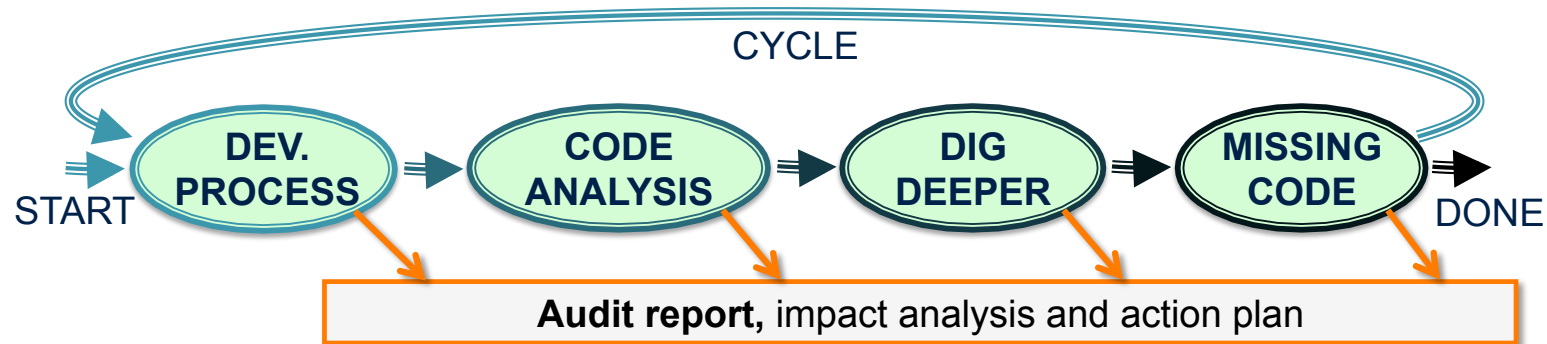
An **audit** is a systematic and independent examination of [...] to ascertain how far the [...] present a true and fair view of the concern. (*Wikipedia*)

- In the context of the *Universal Verification Methodology* (UVM):
 - examination of **existing code-base** and verification **methodology**
 - ascertain if appropriate, **best-in-class, UVM-like solutions** are being used
- Verilab consultants involved in several types of audit, including:
 - **formal audit** – typically at key methodology milestones
 -  – **ad-hoc audit** – typically performed when joining a project
- This tutorial provides **strategy** and **guidelines** for **auditing UVM** projects
 - that you can apply to ongoing, legacy and future projects

Why bother?

- UVM allows enough **flexibility** to write really **bad testbenches**
 - need to apply **verification, H/W & S/W expertise** to get excellent results
- Primary benefits from an audit include:
 - improved **code quality, testbench effectiveness & project efficiency**
- Who benefits from an audit:
 - **mature team**: supports ongoing **quality improvements**
 - **mixed team & externals**: allows for **consistent** code & **predictable** projects
 - ☒ – **individual**: know what you are **getting into**, informed **effort & risk** assessment
- Keep a **positive** attitude...
 - **knowing** the testbench **limitations** is *always* a **good** thing!

Audit Stages



- determine if **development process** includes key items
- **analysis** of code-base looking for typical problems
- **dig deeper** to validate if claimed behavior will scale
- assess if expected code artifacts are **missing**

Depends on audit **context & recipients**

- formal **report** (action plan)
- ...
- **no write-up** (private notes)

Get a handle on framework within which code was developed

DEVELOPMENT PROCESS

Reviewing Development Process

- Comprehensive **audit** of testbench **development process**
 - **essential** for **formal** methodology **audit** (detailed analysis)
 - **beneficial** for **ad-hoc** project **audit** (pragmatic overview)
- Looking for **evidence** of:
 - **coding & style guidelines**
 - **code review** culture
 - code **generation** & **template** library (register-model & verification components)
 - revision control & consistent **simulation** and **regression tool** usage


Full analysis of **development process**
is **outside** the scope of **UVM audit** ☹


What To Look For


- Coding & style guidelines
 - do they **exist**, are they reasonable & are they being **applied**?
 - are they **automated** into tools (**linting**, scripts or checklists)?
- Code reviews
 - are code reviews being **done** at all?
 - using client-server based code-review **methodology**?
- Code generation & template libraries
 - do generators produce **good regmodel** and **UVC frameworks**?
 - is the application-specific **content** also high-quality **UVM** code?


 consistency

 repeatability

 efficiency

 peer training

 resourcing

 no guarantee
of UVM content

Reviewing existing code-base to identify problems

CODE ANALYSIS

Reviewing Code-Base

- Comprehensive audit of existing code-base
 - identify areas of concern that can cause problems
 - looking for evidence of non-UVM like patterns
- For each audit item:
 - **Problem** - statement and clarification if why it is incorrect
 - **Indicator** - of conceptual or fundamental issue
 - **Solution** - what should have been done instead, or could be done now
 - **Effort** – required to repair or live with the problem
 - **Tip** – where possible provide tip of how to find evidence

Important note: this is *not* a UVM course, and we are not trying to *justify* UVM

- code examples just show patterns we are looking for, not individual fixes

Using Tasks Instead of Sequences

- **Limits** controllability & **effectiveness**
- Ubiquitous use of tasks indicates **lack of understanding of CRV**
- Sequences with constrained random control knobs much more powerful
- Lot of effort to repair and retrain
- Review sequence libraries & tests

```
task write_bus(addr, data);  
  `uvm_do_with(item, {  
    direction == WRITE;  
    address == addr;  
    wdata == data;  
  })
```

```
// randomize params...  
task set_config(...params...);  
  // randomize local vars...  
  write_bus(a1,d1);  
  write_bus(a2,d2);
```

```
class config_seq extends base_seq;  
  // rand control knobs and constraints...  
  `uvm_do_with(write_seq, addr==a1; data==d1;)  
  `uvm_do_with(write_seq, addr>a1; data inside {[100:200]});
```

Using *\$random* and *\$urandom*

- **Less powerful** and less **stable** than built-in UVM randomization
- Strong indicator of **bad** sequence based **stimulus** and **CRV** know-how
- UVM has mechanisms to maximize random stability & provides capability for complex constraints
- Lot of effort to repair and retrain
- *grep* for *\$random* & *\$urandom*

```
class example ... // or task
    bit mode = $random;
    bit [2:0] cfg;
    if (mode==0)
        cfg = $urandom_range(0,3);
    else
        cfg = $urandom_range(4,7);
```

```
class example_seq...
    rand mode_t mode;
    rand int cfg;
    constraint cfg_c {
        cfg inside {[0:7]};
        (mode==LO)-> cfg inside {[0:3]};
        (mode==HI)-> cfg dist {4:=1,[5:7]:/1};
    }
```

Duplicating Register Model Code

- **Bad for maintenance**, extremely bad for **derivative** designs (register changes => chaos)
- Indicates **lack of understanding** of **uvm_reg** model usage
- Proper coding is *immune* to field position changes in reg, if it moves to another register we *now* get compile error
- Straightforward to repair
- *grep* for “reg*.read”, *grep* for explicit data slices

```
regm.regx.read(status, data);  
if (data[3:0] > 0) // field a  
    ...  
flag = data[7:7]; // field b  
    ...
```

```
regm.regx.mirror(status);  
if (regm.regx.flda.get_mirrored_value() > 0)  
    ...  
flag = regm.regx.fldb.get_mirrored_value();  
    ...
```

Active (Only) Register Modeling

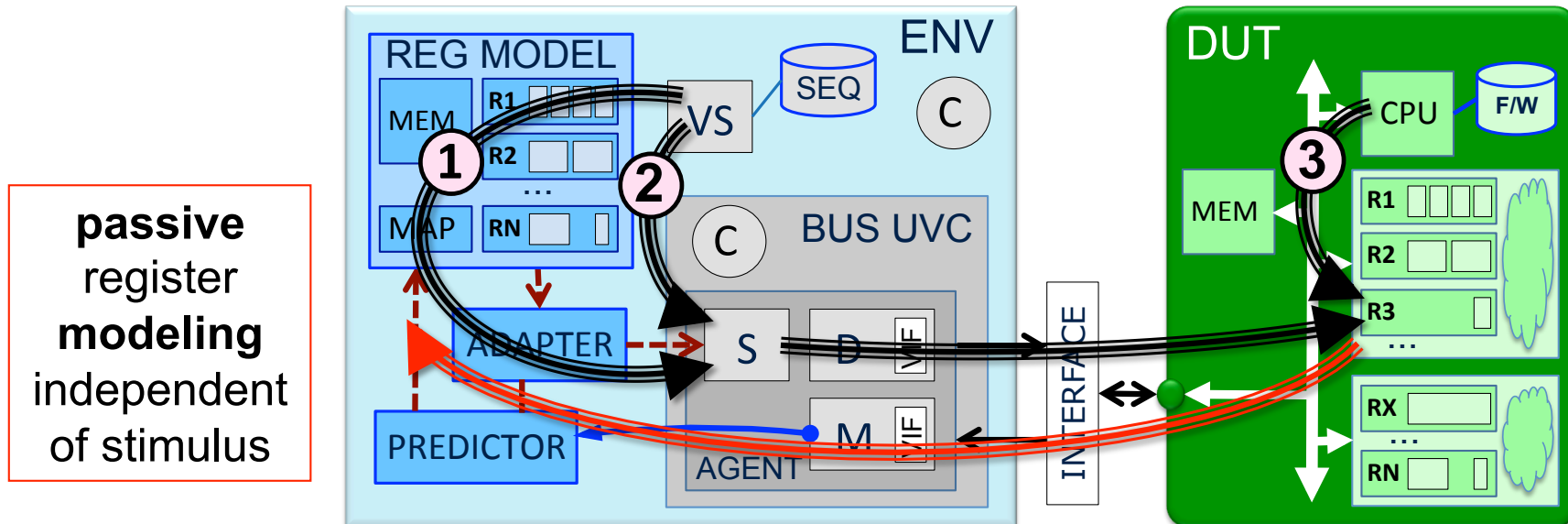
- Using active methods to model registers **limits reuse**
- Indicates **lack of expertise** with **regmodel** concepts
- Passive modeling more flexible powerful, required sys level
- Medium effort to repair
- *grep* for pre/post_read/write
- [4] ***Advanced UVM Register Modeling***

```
class my_field_t extends uvm_reg_field;  
  virtual task post_write(uvm_reg_item rw);  
  ...
```

```
class my_field_cb extends uvm_reg_cbs;  
  virtual task post_write(uvm_reg_item rw);  
  ...
```

```
class my_field_cb extends uvm_reg_cbs;  
  virtual function void post_predict(...);  
  ...
```


✓ Active & Passive Register Model Operation



- Model must tolerate active & passive operations:
 - active** model read/write generates items via adapter
 - passive** behavior when a sequence does not use model
 - passive** behavior when embedded CPU updates register

Ubiquitous Regmodel Handles

- Ubiquitous handles to regmodel are **project specific & fragile** code
- Interface protocol independent of project register implementation
- Indicates **lack of awareness** of alternatives
- Isolate functional behavior from register encoding and DUT-specific details
- Lot of effort to repair, but can be done
- [2] *Configuring a Date with a Model*

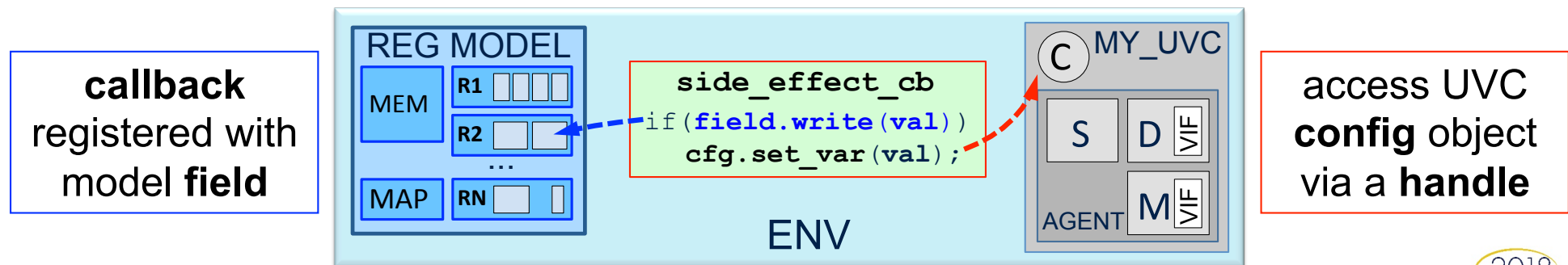
```
class my_bus_monitor ...;
  my_project_regmodel regm;
  ...
  if (regm.regx.fldc == 5)
    crc = calc_crc_modex(data);
```

```
class my_bus_monitor ...;
  // no regmodel handle allowed
  ...
  if (cfg.mode == MODEX)
    crc = calc_crc_modex(data);
```

```
class fldc_cb extends uvm_reg_cbs;
  ...
  function void post_predict(...);
    if (value==5) cfg.mode = MODEX;
```

✓ Update Configuration Using Callbacks

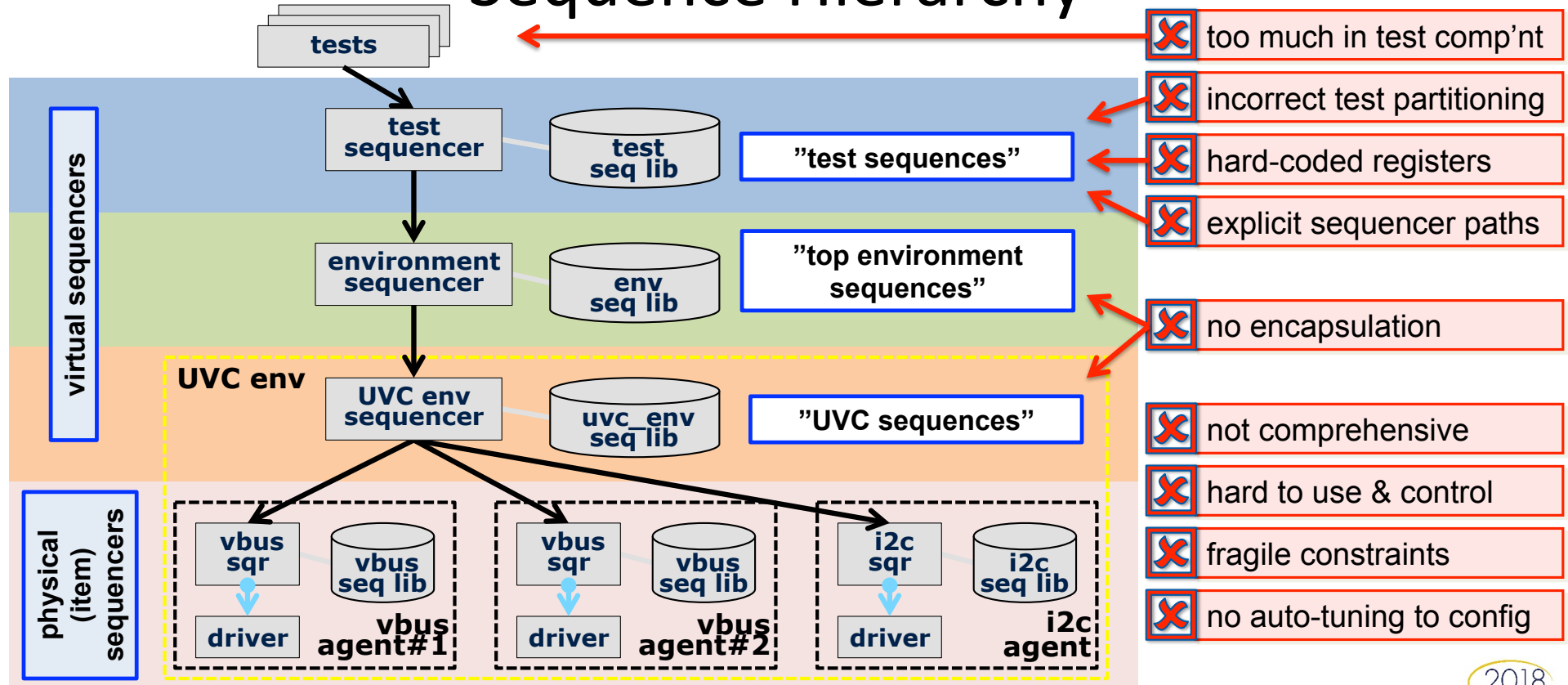
- Randomize or modify registers & reconfigure DUT...
- How do we update **UVC configuration** if it has no regmodel?
 - update from **register sequences** ☒ not passive
 - **snoop** on DUT bus transactions ☒ not backdoor
 - implement **post_predict** callback ☒ passive & backdoor



Poor Sequence Hierarchy & Encapsulation

- Bad sequence architecture **compromises reuse** and **effectiveness**
- Indicates **limited understanding** of **constrained-random stimulus**
- Correct encapsulation of resources (register model accesses, sequencer hierarchy and associated configuration) enables test reuse etc.
- Huge effort to repair, but new sequences can be retrofitted in parallel
- Requires expert knowledge to assess quality
 - code review, pattern analysis, layer examination, ...
- [1] *Use the Sequence, Luke*

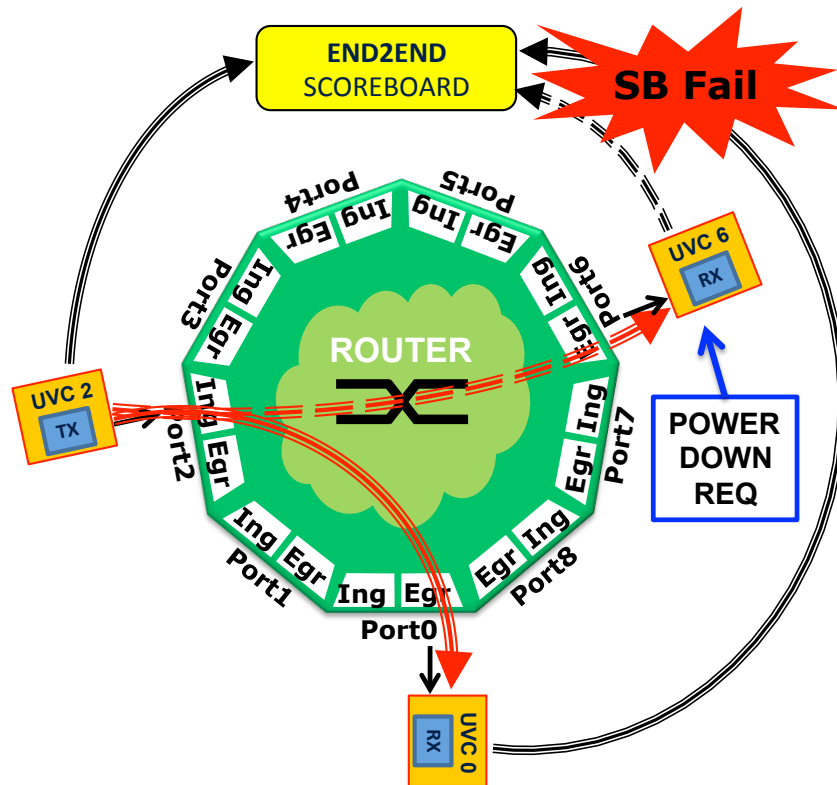
Sequence Hierarchy



Inappropriate Scoreboard Architecture

- Bad scoreboard architecture **compromises reuse** and **effectiveness**
- Indicates **limited understanding** of alternative **scoreboard concepts**
- Check specified transaction relationships at correct level of abstraction
 - *avoid* checking **unspecified** DUT-specific **implementation**
 - *avoid* **cycle-accurate** implementation-specific **modeling**
 - *avoid* white-box **probing** of internal DUT signals
- Huge effort to repair, but can be retrofitted in parallel with original
- Requires expert knowledge to assess quality
 - code review, pattern analysis, concept understanding, ...

NoC Router Example



Not Good Enough

- packets take time through router
- power-down requests **anytime**
- target can decide to power-down **just before a packet arrives**

```
// Basic Re-Routing
if (target POK) // power-ok
    expect packet at target
else // power-down
    expect packet at port0
```

X do *not* model or probe internal timing or impl'n

```
// add target POK to src & dst transactions
// apply fuzzy logic for expected result
case ({src_tr.pok, dst_tr.pok})
    00 : must go to port0
    11 : must go to target
    01,10 : may go to port0 or target
+ packet must not go to both,
+ packet must not get fragmented
```

Obsession With Seeds

- Symptom: regression files with many **explicit “magic” seeds**
 - seeds have limited lifetime during CRV development
 - we don’t know why seeds were considered special
 - original scenario is probably not stimulated but appears to pass
- Indicates a serious **lack of understanding of random stability & CRV**
- Assuming seed originally exposed an interesting scenario...
 - functional coverage & checks should have been implemented
 - constraints maybe needed modified to make it more likely
- Potential very high effort to recover, if coverage and checks inadequate
 - easy to fix in regressions (remove seeds) but impact is very hard to assess
 - training requirement for team to understand the issues here

Minor Things, Major Time-Wasters

- **Commented-out code** (should it be?)
 - use of block comments strongly discouraged since hinders grep detection
- **Badly encapsulated code** with much **repetition** and huge files
 - could seriously affect reuse and ramp-up time, as well as being error prone
- **Inappropriate use** of ***assert*** for randomize or assert(0)
 - stimulus and messages could be affected if assertions disabled
- **Bad coverage encapsulation** inside monitor or scoreboard components
 - *covergroups* should be inside dedicated container class for safe overrides
- **Inappropriate use** of ***config_db*** for dynamic operations
 - use *config_db* for static configuration, otherwise use configuration objects

Execute and experiment with the code-base

DIGGING DEEPER

Due Diligence

- Additional analysis is often required for ***due diligence***, for example:
 - where a formal audit is requested to assess code quality
 - where effort estimates based on legacy codebase are not clear
- Recommend **digging deeper** into code-base to assess UVM quality
- Requires a **working code-base** and regression environment
- In addition to a deeper analysis of the actual code by **inspection**, we assume some attempt to validate claims by **execution**

HAVE A LOOK: what to look for in the code-base

TRY IT OUT: experiment with the code-base

Reusable Block-Level Environment

- Block-level verification environment is complete and can be plugged into system level environment for **100% reuse**
- Have a look
 - active/passive settings and usage
 - build control, connectivity, architecture
- Try it out
 - **instantiate a passive shadow environment**
 - in parallel with existing active block-level environment
- [5] ***Pragmatic Verification Reuse in a Vertical World***

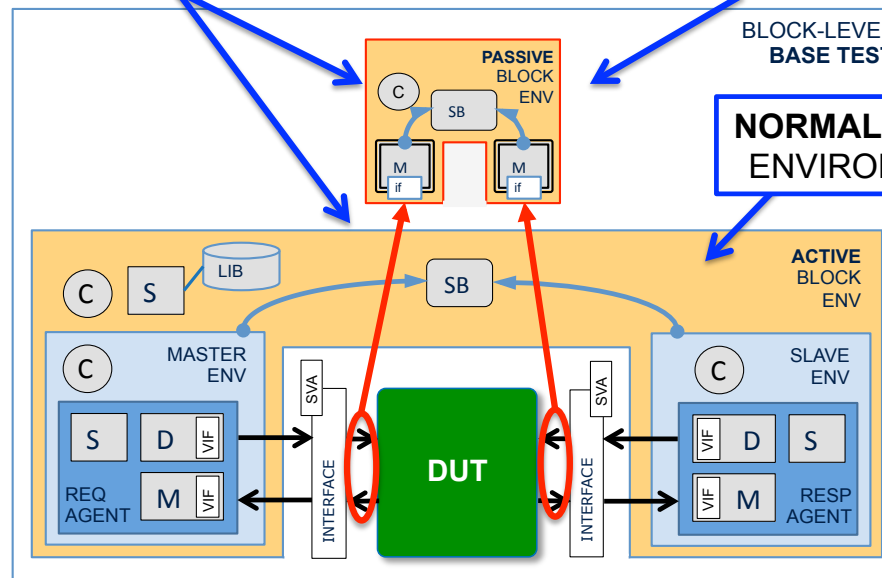
Passive Shadow Environment

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

**SHADOW PASSIVE
ENVIRONMENT**

BLOCK-LEVEL
BASE TEST

**NORMAL ACTIVE
ENVIRONMENT**



PROVE FUNCTIONALITY USING A PASSIVE SHADOW ENV

Comprehensive Sequence Library

- Available sequences provide **comprehensive stimulus** for all sorts of great scenarios
- Have a look:
 - apply **expert knowledge** to see if sequence set & encapsulation is good
- Try it out
 - temporarily **modify** a working test
 - randomize sequences 1000's of times
 - looking for randomization errors etc.
- [1] *Use the Sequence, Luke*

```
// temporarily replace  
'uvm_do(example_seq)
```

```
// with this sort of thing...  
'uvm_create(example_seq)  
repeat(1000)  
  if (!example_seq.randomize())  
    `uvm_error("RNDFLD", "...")  
repeat(1000)  
  if (!example_seq.randomize() with {  
    example_seq.mode == FAST_MODE;  
  }) `uvm_error("RNDFLD", "...")  
'uvm_do(example_seq) // as before
```

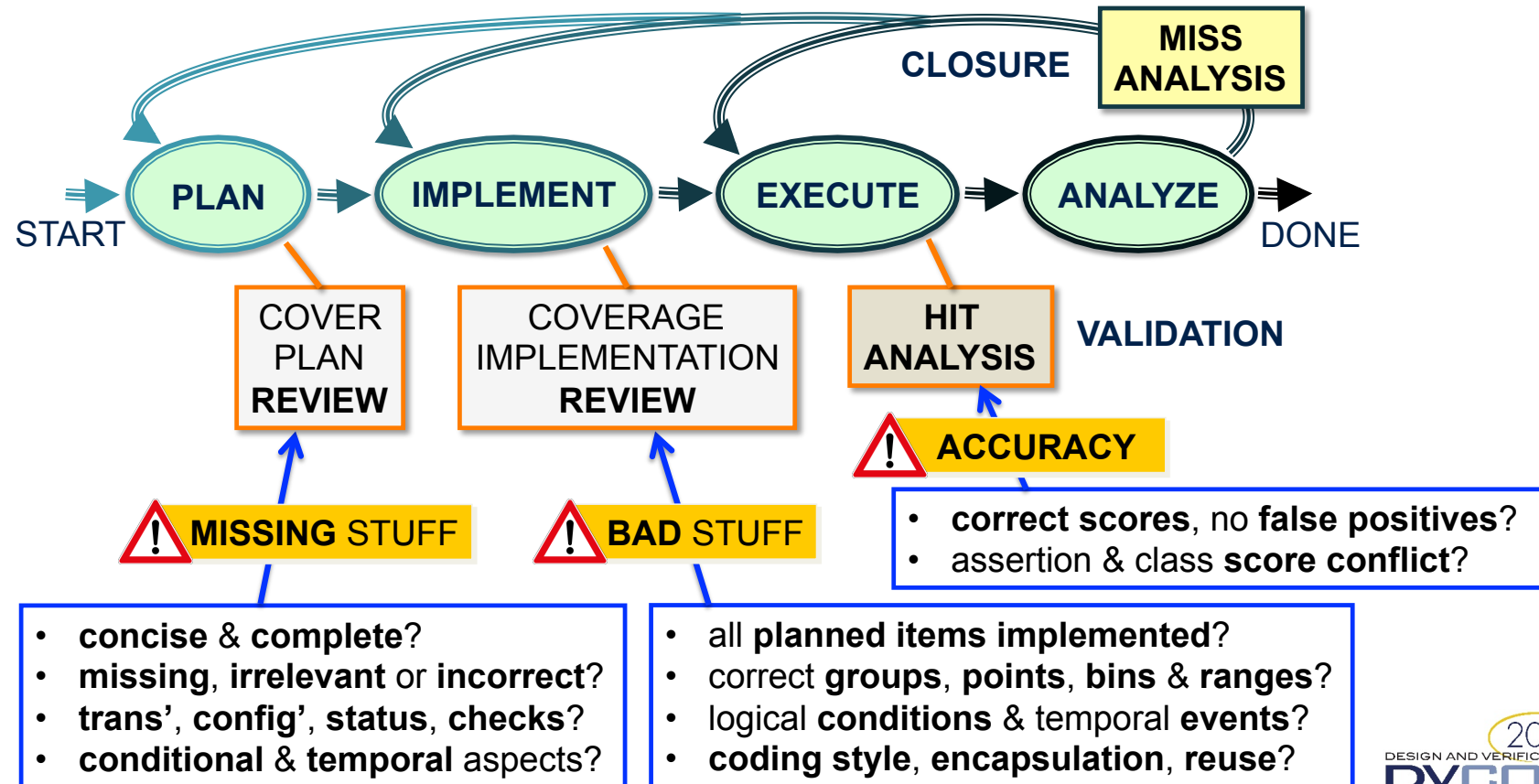
Parameterized Environment

- Environment is **fully parameterized** and will adapt to the next generation of parameter settings with almost no effort
- Have a look
 - are all aspects of the classes **parameterized correctly**?
 - do the **config, stimulus, checks** & functional **coverage** adapt?
- Try it out
 - change the parameter settings in existing environment
 - how painless was that?
 - did environment build and execute as expected?
- [6] ***Advanced UVM Tutorial: Parameterized Classes, Interfaces and Registers***

Comprehensive Functional Coverage

- Comprehensive **functional coverage** with **100%** results
- Have a look
 - does implemented coverage model look **comprehensive**?
 - is the coverage collected at the correct **time** and logical **conditions**?
 - does it include **configuration**, **transaction** and **temporal** relationships?
- Try it out
 - run a few individual tests in isolation, **validate** exact **scores** in all bins
 - does coverage tell *the truth, the whole truth, and nothing but the truth*?
- [3] ***Lies, Damned Lies, and Coverage***

Functional Coverage Analysis



Build Control

- Environments often provide **build control** variables for components
 - e.g. *has_master*, *has_slave*, *num_agents*, etc.
 - fields should be encapsulated inside configuration objects
- Have a look
 - are fields used consistently in component **build hierarchy**?
 - are fields used correctly to tune **sequences**, **checks** and **coverage**?
- Try it out
 - patch (environment) to select **different topology**
 - execute tests at least as far as connect phase
 - does the environment only build in original topology configuration?

Check and Coverage Control

- Agents should provide **check** and (optionally) coverage **enable**
 - e.g. *checks_enable*, *coverage_enable*
 - should be in config objects, sometimes in component class
- Have a look
 - is *checks_enable* used to control **all checks** and **only checks**?
 - does *coverage_enable* **only** affect **coverage collection**, and nothing else?
- Try it out
 - patch (base test) to **disable checks** in working simulation
 - does the **stimulus** function *identically* when checks are off? (compare logs)
 - are there **no error** or check **messages**? (*expect* checks disabled warning)
 - is the corresponding **assertion coverage** score = 0?

Identifying what is not there, but should be

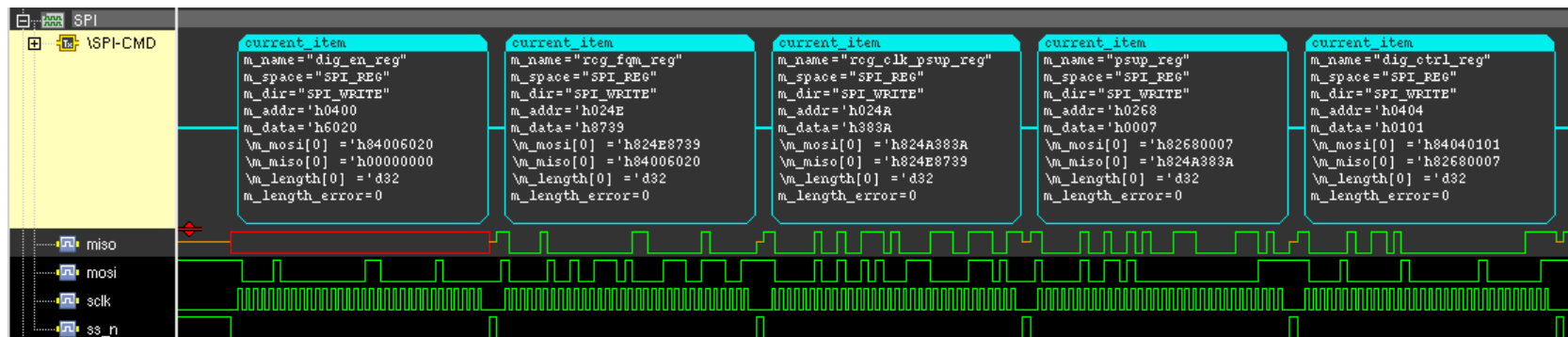
MISSING CODE

Reviewing What is Not There

- Not enough to do a comprehensive **audit** of the **existing code**
 - we also need to assess if **anything important** is **missing**
 - from an application standpoint this is difficult...
 - but for UVM there are specific additional things we expect to see
- Audit perspective:
 - are these **coding patterns there** at all?
 - if present, are they **done correctly**?
- Following slides provide just some additional examples...
 - some gaps may also be revealed due to analysis from previous sections

Transaction Recording

- Transaction recording enhances testbench debug capabilities
- Is **transaction recording used correctly** in monitor components?
 - do the transactions **start** and **end** at appropriate **times**?
 - are transactions instrumented with **informative content** (e.g. reg name)?



Appropriate Messages

- **Concise** informative **messages** with correct **verbosity** control
 - greatly enhance testbench effectiveness and debug efficiency
- **Review** regression **log-files** at low verbosity
 - are they full of inappropriate **clutter**?
 - are there **concise** messages that show **operation** and **context**?
 - do transactions have single-line **summary** (e.g. using *convert2string*)?
 - are all messages at the correct **severity** (e.g. warning for error injection)?

```
UVM_INFO @ ... [ahb_monitor] AHB READ (addr=0x00, data=0x24 => STAT_REG)
UVM_INFO @ ... [ahb_monitor] AHB WRITE(addr=0x02, data=0x01 => CTRL_REG)
UVM_INFO @ ... [rst_monitor] SW RESET observed
UVM_WARNING @..[spi_monitor] SPI READ aborted due to RESET
```



Separation of Concerns in Test Suite

- Regression **test suite** should include tests with:
 - feature-based **isolation** of verification concerns (constrained random)
 - meaningful **combinations** of interacting aspects (constrained random)
 - additional highly **random scenarios** (legal constraints only)
 - specific **application use-cases** (heavily constrained => directed)
- Do not expect to see:
 - ***just directed tests*** for specific features or use-cases
 - ***just*** extremely **random tests** doing everything all the time
- Badly architected test suite also effects **efficiency** of derivative project
 - hard to assess impact if we modify, add or remove features

Traceable Checks

- Not enough to have various **checks** *apparently* implemented
 - we expect them to **fail** when required...
 - but we must also **know** that they **executed** and **passed**
- Requirement for functional safety related verification (ISO-26262)
 - but also good practice for any testbench
- Use **assertions** for **all** DUT-relevant **errors** (=> automatic coverage)
 - *immediate* assertions in procedural code, *concurrent* assertions in interfaces

```
if (data != exp)
  `uvm_error(get_type_name(),"failure info...")
```

```
AS_DATA_CHECK : assert (data == exp) else
  `uvm_error("AS_DATA_CHECK","failure info...")
```



Some Other Things To Look For...

- Does each UVC package define ***timeunit*** & ***timeprecision***?
 - omission can be serious time waster due to timescale order rules
- Does the environment make use of ***real*** and ***time*** variables?
 - these can now be used for *rand* fields (instead of integer and precision)
- Do the interface UVCs provide **error injection** capability?
 - e.g. serial interface (SPI, I2C, etc.) with long/short length errors
 - how are these handled in the transactions and regmodel adapter?
- Are **sanity regressions** setup and do they run successfully?
 - in general is the regression suite well organized and appropriate?

What to say and how to use the audit information

REPORTING FINDINGS

Reporting Audit Results

- **Report format** depends on **audience & goal** of the audit
 - formal audit requires **formal report document**, possibly for 3rd party
 - ad-hoc project ramp-up probably requires **informal notes** to be **shared**
- Amount of **detail** and conclusions depends on **expectations**
 - formal audit: expected to deliver **detailed information** (easy to handle)
 - stealth audit: team may expect **no information** (hard to handle)
- Content should be **positive, constructive** and **respectful**
 - describe what can be improved, how & why (not just identify what is wrong)

Verification **engineers**
are **people** too!

Finding an RTL bug in **verification** => ***always*** good!
Knowing the testbench limitations => ***always*** good!

Action Plan

- What you do with information depends on team **role** & project **maturity**
 - verification **lead** on **new product** family (address **all findings**, **plan** accordingly)
 - **joining** project with **planned derivatives** (ruthless **prioritization**, +post tape-out)
 - **fire-fighting** role on **end-of-line** project (understand **risks**, **minimize** changes)
- Do not change all of the code, all of the time
 - inappropriate to introduce too many changes without **stable regressions**
 - safety net of high-quality **metrics** (functional, assertion & code coverage)
 - **prioritize changes** according to an agreed action plan
- Either way we have more **realistic** picture now, than before the audit
 - e.g. **reuse** from a legacy project might be **limited** or counter-productive

Setting Priorities

- **Identifying problems** and knowing how to fix them is one thing...
- ...but **prioritizing effort** for incremental improvements is another!
- Best case: start of **new project** with **planned derivatives**
 - **do not compromise** on **architecture** or **reuse** aspects
 - roll-out **stimulus**, **checks** and **coverage** (in that order)
 - keep designers busy & build (everyone's) confidence in testbench
- Worst case: **fire-fighting** inherited **mess** with **tight** project **timelines**
 - change **as little as possible**, and **manage risk** through raising awareness
 - focus on **stimulus** improvements (find bugs), then checks and coverage
 - leave architecture and reuse until post tape-out (end-of-line => never)

CONCLUSION

Conclusion

- Presented **pragmatic** approach to various aspects of **verification audit**
 - *focus* on **UVM** and related infrastructure
 - *overview* of development process and reporting findings
 - *details* on architecture, code analysis, digging deeper, and missing items
- Content should **benefit any** level of **audit** or review process
 - formal, ad-hoc or even stealth (uninvited) audits
- Premise:
 - **knowing** the **testbench limitations** is a **good** thing
 - this helps projects with productivity, planning and risk management
- Hope it helps **you** improve quality & effectiveness of your testbenches

References

- 1 *Use the Sequence, Luke* - SNUG 2018**
- 2 *Configuring a Data with a Model* - SNUG 2016**
- 3 *Lies, Damned Lies, and Coverage* - DVCon 2015**
- 4 *Advanced UVM Register Modeling* - DVCon 2014**
- 5 *Pragmatic Verification Reuse in a Vertical World* - DVCon 2013**
- 6 *Advanced UVM Tutorial – I & II* – DVCon 2014 & 2015**

All these **papers** and **presentations** available from:

<http://www.verilab.com/resources/papers-and-presentations/>

Questions