Lies, Damned Lies, and Coverage

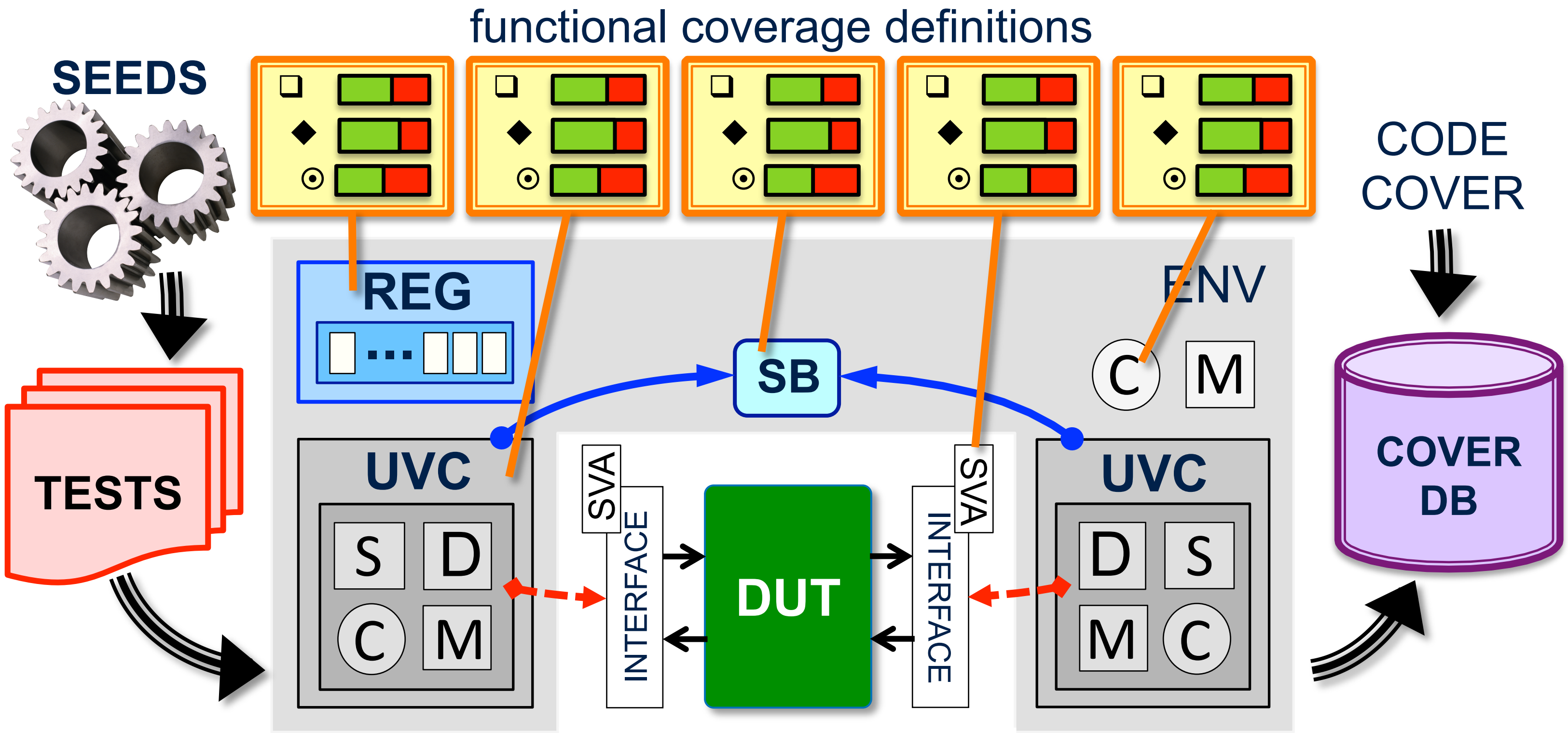
Mark Litterick, Verilab, Germany

verilab

Introduction

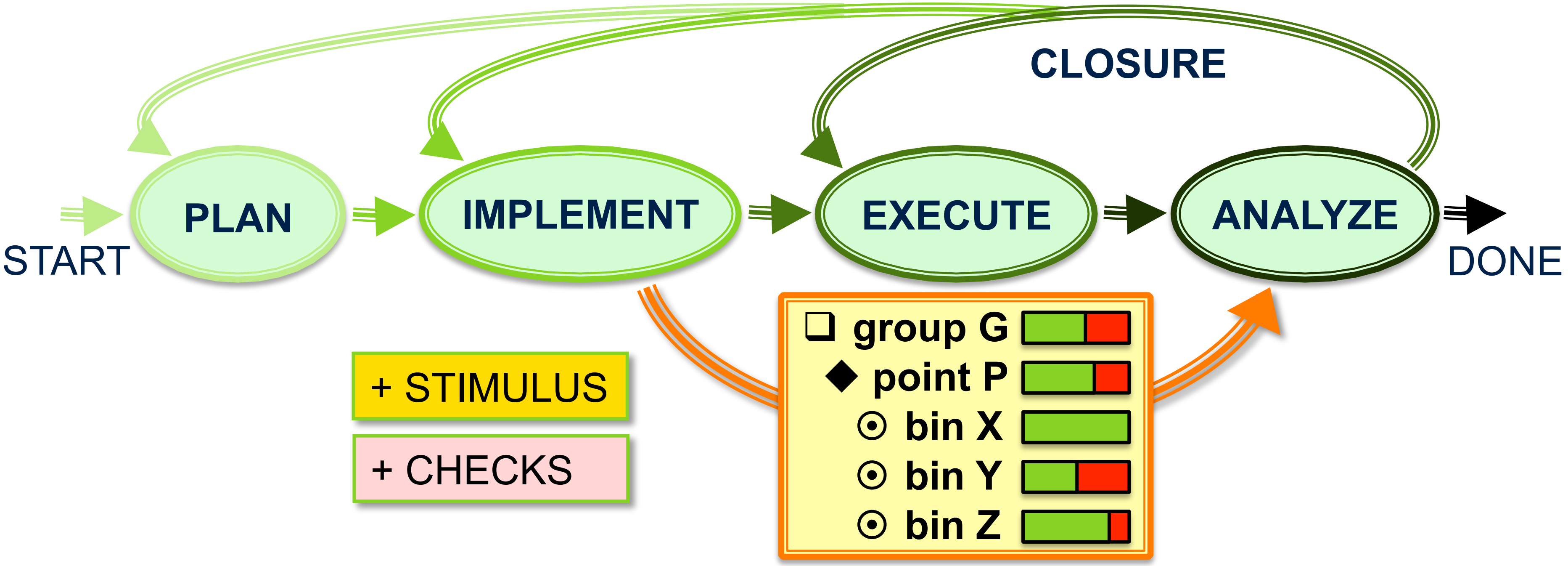
- **Overview** of functional coverage & flow
- The **problem** – "*lies, damned lies, and coverage*"
- Provide **examples**
  - transaction coverage
  - temporal coverage
  - register models
- Discuss **solutions**
  - methodology and reviews
  - hit analysis and cross-referencing
  - automatic coverage validation using UCIS

Functional Coverage



- **Key metric** in establishing verification **completeness**
  - essential for **constrained random**, beneficial for **directed testing**
- Implement ***covergroups***, ***coverpoints***, ***bins***, ***assert/cover***
  - record all **important** artifacts of **stimulus**, **configuration** & **checks**

Coverage Flow



CLOSURE

START

PLAN

IMPLEMENT

EXECUTE

ANALYZE

DONE

+ STIMULUS

+ CHECKS

❑ group G

◆ point P

⊙ bin X

⊙ bin Y

⊙ bin Z

- **Manually specified** items identify important concerns
- Coverage **holes analyzed** to achieve **closure**
  - **execute** more **tests** and/or more **seeds**
  - **improve stimulus** and/or **coverage** implementation
  - ...repeat until **done**! (or tape-out with **known risk**)

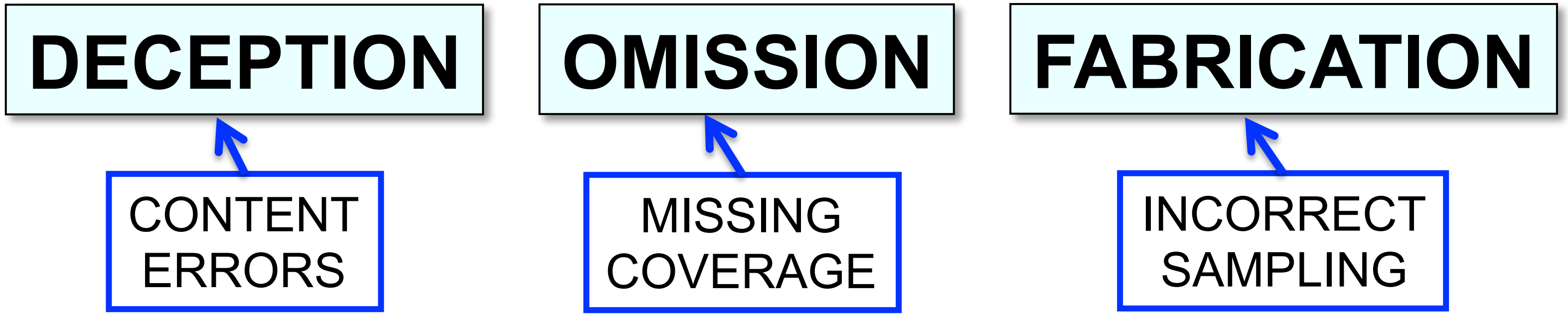
The Truth, The Whole Truth, and Nothing But The Truth...

- **Empirical evidence** suggests **coverage** models are:
  - **inaccurate**
  - **misleading**
  - **incomplete**

Observations based on:

- many **projects**
- different **clients**
- diverse **applications**
- various **languages**

- ...all the symptoms of ***a pack of lies:***



The Problem...

- Lies in the coverage model are a major **problem**, since:
  - coverage **closure** focuses on **holes** in report
  - positive **hits** are taken as **fact** and get little attention
- If coverage does not stand up to **cross examination**
  - **destroy credibility** of verification environment
  - harm **reputation** of verification team
- If coverage **lies** remain **undetected**...
  - key device **features** could remain **unverified**
  - significant **risk** to project **quality**

COVERAGE ERRORS
CAN GO **UNNOTICED**

Non-Malicious Behavior

- Clarification (in general):

LIES IN THE **COVERAGE MODEL** ARE **NOT** A RESULT OF **MALICIOUS BEHAVIOR**

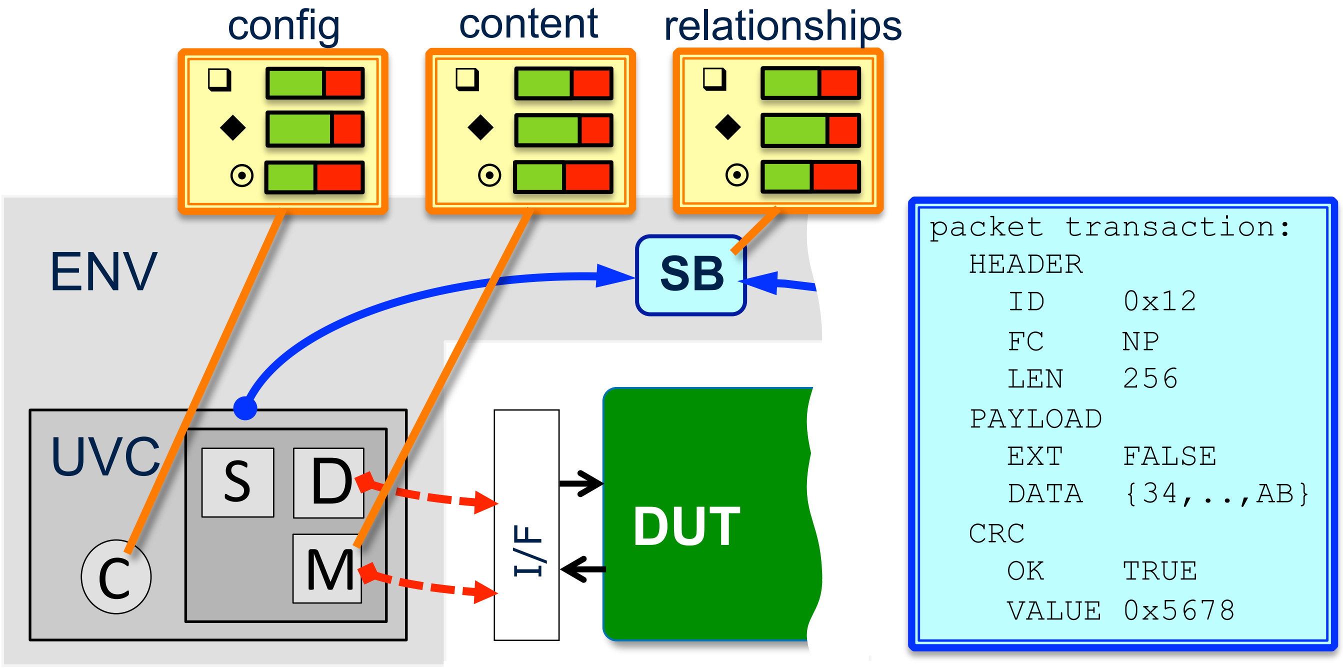
  - errors, omissions and fabrications are **not deliberately introduced**
  - we are **not** trying to **trick others** or **fool ourselves**!
- ...it is ***possible*** to **manipulate** code to get **100% coverage**
  - remove hard-to-reach coverpoints, introduce extra sampling events, manipulate ranges to absorb corner cases, etc.
  - **malicious behavior**, but technically straightforward...

NEVER BEEN OBSERVED ☺

- ...**empirical evidence** suggests **false 100% coverage**!
  - missing coverage, incorrect sampling, bad ranges,...
  - **accidental** root cause, but **same** miraculous **result**!

COMMON PROBLEM

Transaction Coverage



```
packet transaction:
  HEADER
    ID     0x12
    FC     NP
    LEN    256
  PAYLOAD
    EXT    FALSE
    DATA   {34,..,AB}
  CRC
    OK     TRUE
    VALUE  0x5678
```

- required **operations** performed under **all configurations**?
- all **transaction kinds** observed at **each** DUT **interface**?
- all relevant (to DUT) **field values, ranges** and **special cases**?
- every possible **transaction relationship** and **order** observed?
- all appropriate testbench **error injection** and **detection** by DUT?

Example Transaction Lies

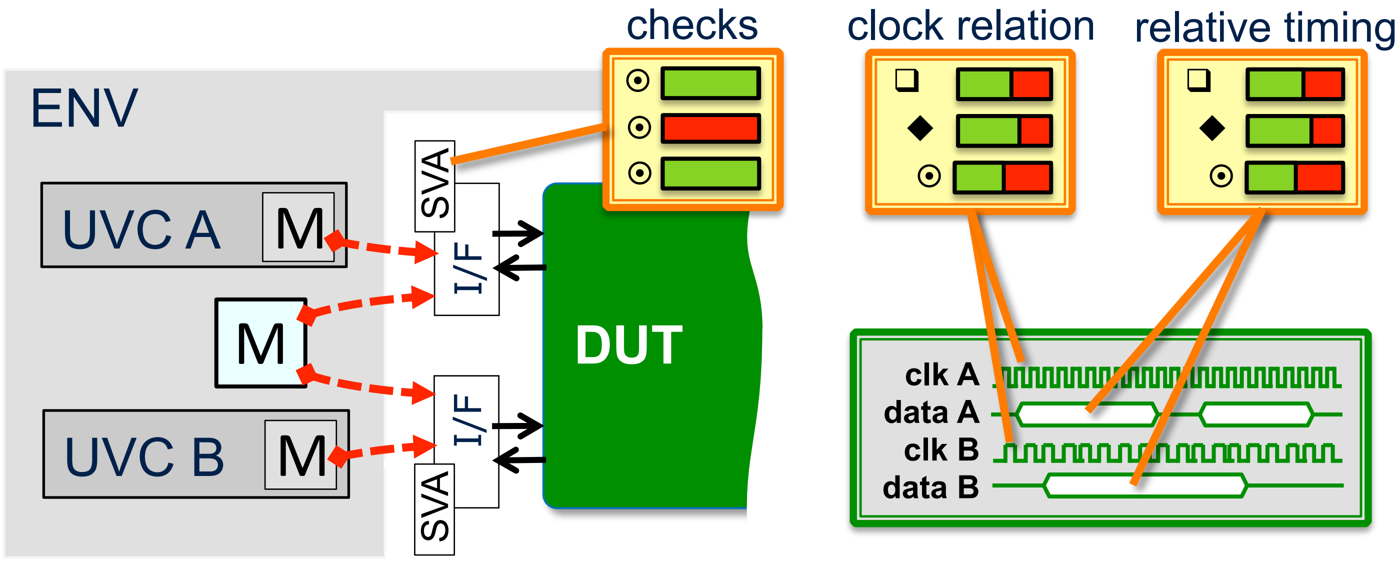
e.g. **TX** AND **RX** CONFIG SAMPLED FOR **TX-ONLY TEST**
(CONFIG SHOULD BE **SAMPLED** WHEN IT IS **USED**)

e.g. BINS “**[1:5],[6:10],[11:20]**” USED WHEN **0** AND **1** ARE **CRITICAL**
(BINS “**0,1,[2:19],20**” BETTER? ACTUAL APPLICATION MINIMUM?)

ASPECT	OBSERVATION	LIE
Ranges	**Incorrect range** that hides key corner values	**Deception**
Conditional	Field values with **incorrect conditional filtering**	**Fabrication**
Configuration	**Sample config** fields **when value is set** or changed	**Fabrication**
Relationships	**Only** single **transaction** coverage, **no relationships**	**Omission**
Error Injection	**Inaccurate** recording of all **error injection** scenarios	**Deception**
Irrelevant Data	Too much data **looks like** lots of **interesting stuff**	**Exaggeration**
...	**...**	**...**

EASY TO **CREATE LOTS** OF USELESS COVERAGE
(**HARD** TO BE COMPREHENSIVE BUT **CONCISE**)

Temporal Coverage



- all appropriate **clock relationships** during observed traffic?
- behavior of (subsequent) **reset** under all **conditions**?
- **relative timing** of transactions on different DUT **interfaces**?
- **timing** of interface **traffic** relative to DUT **internal state**?
- occurrence of **sub-transaction events** that are never published?
- **all** required **checks** happened, how often, under what conditions?

Example Temporal Lies

e.g. **DUT** IS NOT IN A **STATE** WHEN **INITIAL RESET** (CONDITION **SAMPLED** ON **SUBSEQUENT RESET** ONLY)

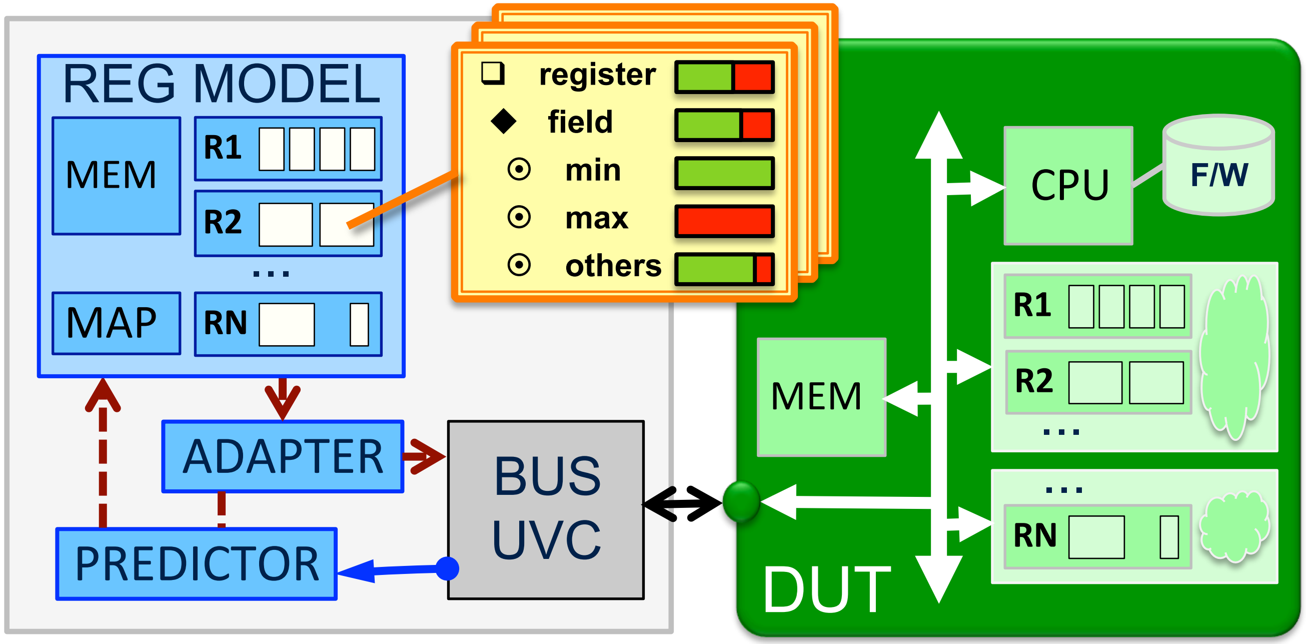
NEED TO VALIDATE **OPERATION** WITH ALL **CLOCK COMBOS** (e.g. NO BUFFER OVERFLOW, FSM INTERACTION, etc.)

ASPECT	OBSERVATION	LIE
Clock Relation	**Missing** or **incorrectly sampled clock relationships**	**Omission**
Reset Conditions	Non-zero reset score after **initial reset**	**Fabrication**
Temporal Relation	Entire model based on **transaction** content **only**	**Omission**
Check Coverage	**Missing** or **incorrectly scoped** coverage of **checks**	**Omission**
Sub-transaction	**Missing sub-transaction event** coverage	**Omission**
…		…

UNLIKELY TO BE **ADEQUATE** FOR DUT WITH MULTIPLE INTERFACES, STORAGE, PIPELINE OR PROCESS **DELAYS**

CAN YOU TELL **FROM** THE **COVERAGE** WHICH FUNCTIONAL **CHECKS PASSED** AND UNDER WHAT **CONDITIONS**?

Register Model Coverage



- **use** all relevant values and ranges in **control** and **configuration**?
- **read** all appropriate **status** responses from the DUT?
- **validate** all the **reset** values from the registers?
- **access** all register **addresses**?
- validate the **access rights** for each register?
- prove all appropriate **access policies** for the register fields?

Example Register Model Lies

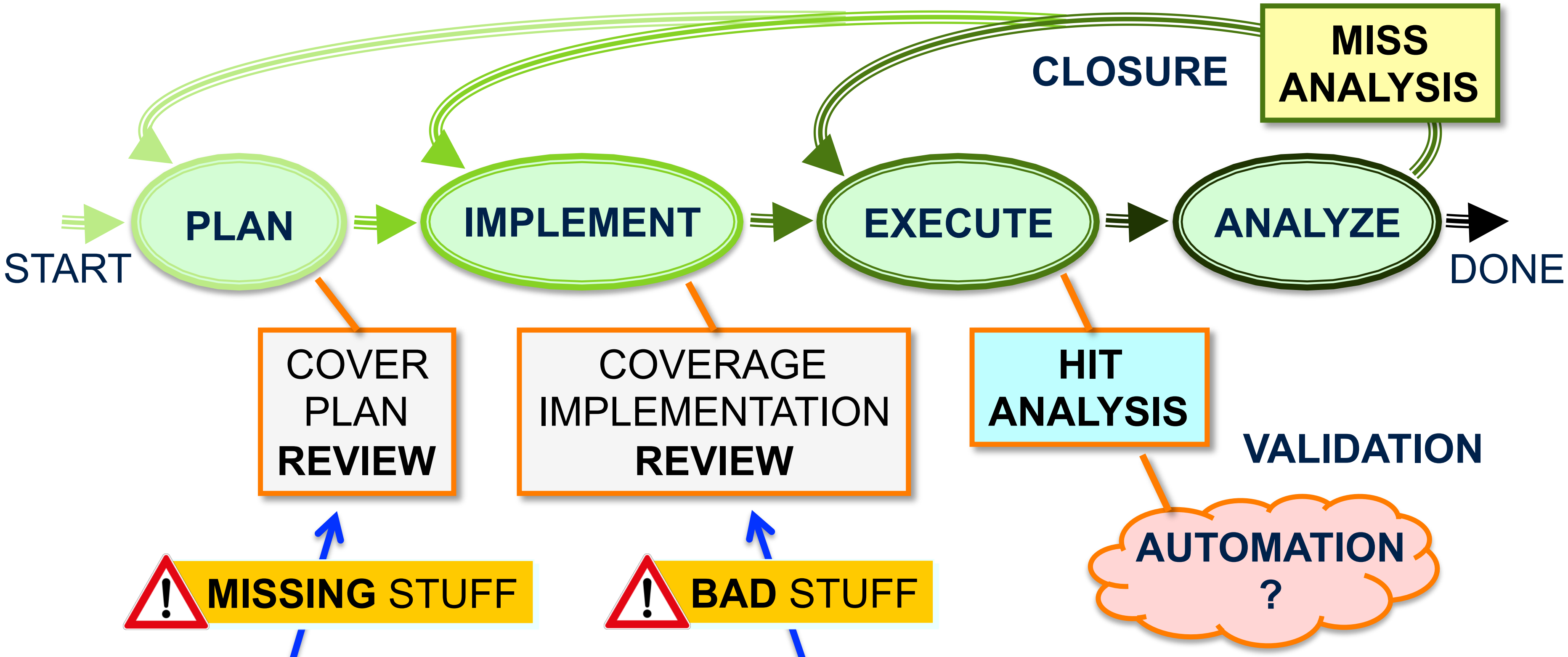
BACKDOOR DOES **NOT VALIDATE** ADDRESS **DECODE**
(EXCLUDE BACKDOOR ACCESS FROM ADDRESS COV)

EASY TO GET **100%** COVER ON **MULTIPLE WRITES**
BUT **MISLEADING** SINCE VALUES **NOT USED** BY DUT

ASPECT	OBSERVATION	LIE
Reg Write	**Control** and **config** values **sampled on write** to register	**Fabrication**
Reg Read	**Status** values **read** from **reset** conditions not DUT operation	**Fabrication**
Reset Value	**Incorrectly conditioned** validation of **reset** values	**Deception**
Address Map	Register **address coverage** from **backdoor** access	**Deception**
Access Right	**Only legal access rights** attempted for restricted registers	**Omission**
Access Policy	**Only legal access policy** recorded in coverage model	**Omission**
…	**…**	**…**

NEED TO **ALSO COVER** ALL RELEVANT **ACCESS ATTEMPTS**
e.g. **WRITE 0** AND 1 FOR **W1C**, **WRITE** AND READ FOR **RO**

Lie Detectors



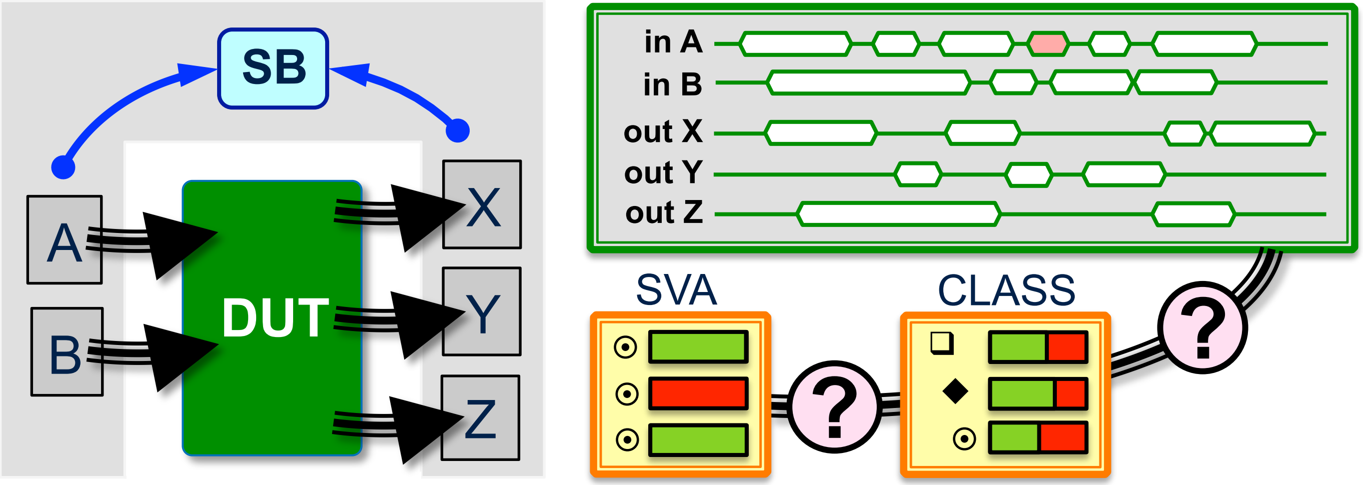
- **concise** & **complete**?
- **missing**, **irrelevant** or **incorrect**?
- **trans'**, **config'**, **status**, **checks**?
- **conditional** & **temporal** aspects?

- all **planned items implemented**?
- correct **groups**, **points**, **bins** & **ranges**?
- logical **conditions** & temporal **events**?
- **coding style**, **encapsulation**, **reuse**?

Hit Analysis

- **Review** of plan and implementation is **not enough**...
  - we need to **validate** if **actual coverage** is **correct**
  - unique **coverage** characteristic: **errors** can go **unnoticed** (unlike stimulus and checks – where errors get noticed!)
- Coverage **closure analysis** is **focused** on **holes**...
  - we also need to **look at *all*** of the **hits**!
- Select a few specific tests and validate that:
  - all **reported coverage** is **exactly** what happened in the test
  - all interesting **stimulus** and **configuration** are **recorded** in coverage
  - all **transaction content** and relevant **relationship** are captured
  - all **checks** that occurred have corresponding coverage reported
  - **no additional coverage** is reported for events that did not happen

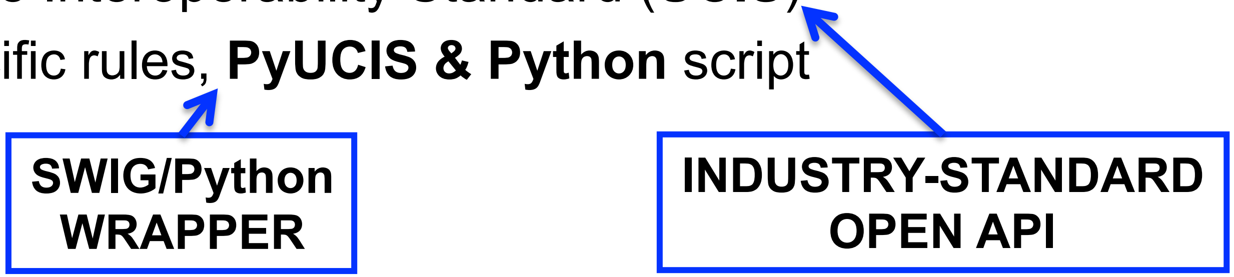
Coverage Analysis Example



- Important to **cross-reference** all aspects of operation
  - compare log file **messages**, **waves** and **assertions** with **coverage**
  - look at the **absolute score** for each and every bin or assertion
- For example (input: 9 good packets & 1 bad packet):
  - all aspects of **transaction content**, **timing** & **relationships** covered?
  - does **coverage reflect** that scoreboard model **dropped error** packet?
  - how many **slices** and/or **packets** were processed in **parallel**?
  - do observed **assertion** scores **match scoreboard** & **transactions**?

Automation

- **Validation** of functional **coverage correctness**:
  - if a ***skilled*** **engineer** can do it by **inspection**...
  - ...can we **automate** the **validation process**?
- Should be possible (to a degree):
  - **rule-based** application of same **cross-checks**
  - ...but **no commercial tools** available
  - (note: only validating coverage scores for implemented code!)
- Ad-hoc **proof-of-concept** demonstrated using:
  - Unified Coverage Interoperability Standard (**UCIS**)
  - application-specific rules, **PyUCIS & Python** script

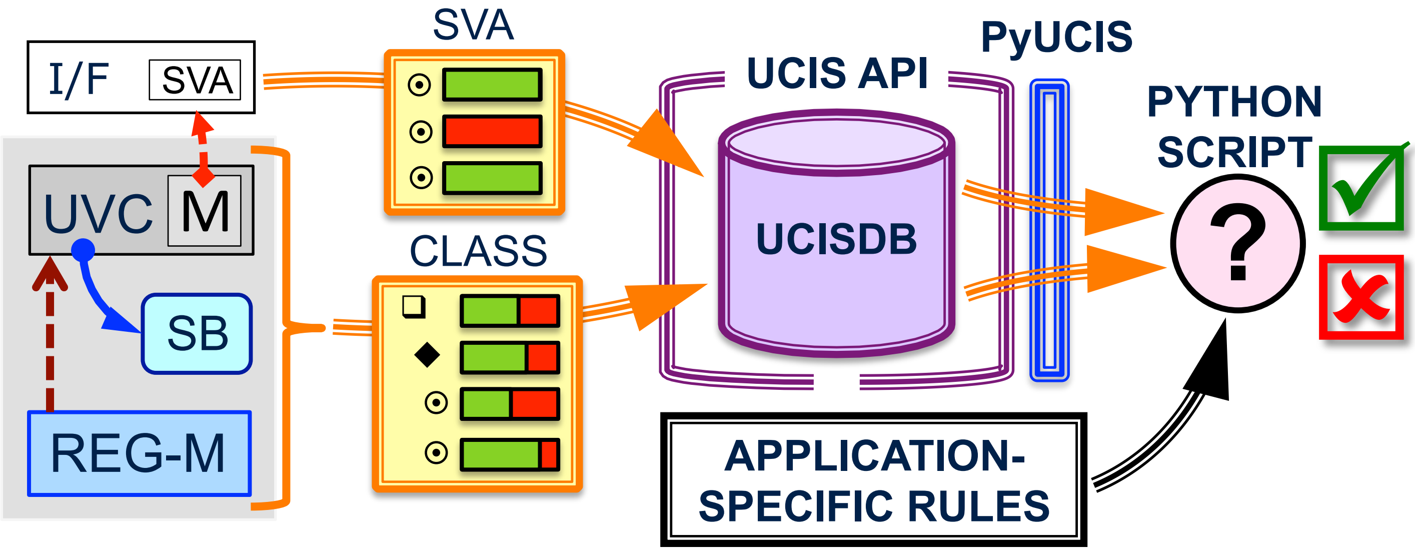


SWIG/Python WRAPPER

INDUSTRY-STANDARD OPEN API

SWIG = Simplified Wrapper and Interface Generator

UCIS Operation



- Using **UCIS** we can access and compare:
  - assertion and class-based coverage scores
  - scores for different assertions in an interface
  - different aspects of class-based coverage

e.g. protocol assertion passing N times ➔ transaction score = N

e.g. transaction content score of N ➔ temporal relationship score = N

e.g. N request phase assertions pass ➔ response assertion score ≤ N

PyUCIS OCP Example

- **UCISDB** stores hierarchy (***scope***) and counts (***coveritem***)
  - to access info - **iterate** through **scopes** for match & extract **count**
  - **PyUCIS** provides simple Python API:

```
ucis_* methods wrapped with SWIG into Python code
pyucis_scope_itr : iterator using ucis_ScopeIterate/ScopeScan
pyucis_cover_itr : iterator using ucis_CoverIterate/CoverScan
pyucis_find_scope, pyucis_get_cov_count, pyucis_get_count,...
```

- OCP application-specific examples (Python script):

```
if (pyucis_get_count(db,”.../checker/a_request_hold_MCmd”)
 != pyucis_get_count(db,”.../monitor/cg_req/cp_cmd”))
  print(“ERROR:
```

cmd type class coverage

cmd hold assertion coverage

```
if (pyucis_get_count(db,”.../monitor/cg_cfg/cp_burstlength/1”)>0)
 if (pyucis_get_count(db,”.../checker/a_request_MBurstLength_0”)
   < pyucis_get_count(db,”.../monitor/cg_req/cp_burst_length”))
    print(“ERROR...”)
```

class score per transaction

this assertion checks on every clk

only if cfg

Conclusion

- Presented **premise** that functional coverage does not tell "***the truth, the whole truth, and nothing but the truth***"
  - based on empirical evidence, observations & experience
- Provided **examples** of what to look out for
  - lies of deception, omission & fabrication in coverage models
- Discussed how to **minimize risk** & **improve quality**
  - plan review, implementation review, hit analysis & raise awareness
- Demonstrated **coverage validation** using **UCIS**
  - proof-of-concept using PyUCIS https://bitbucket.org/verilab/pyucis
  - sanity check for generic environments?
  - part of unit test for VIP providers!

mark.litterick@verilab.com