Building a SystemVerilog Universal Verification Component with the Incisive Plan-to-Closure Methodology

Session 2.12

David Long, John Aynsley and Jonathan Bromley, Doulos

Building a SystemVerilog Universal Verification Component

CONTENTS

- Introduction
- The Verification Plan
- Assertions, Coverage and Constraints
- SystemVerilog Testbench Architecture

DOULOS

SystemVerilog for Verification

- SystemVerilog has verification features not found in VHDL or Verilog:
  - Assertions and a temporal sequence language
  - Functional coverage
  - Constrained random test vectors
  - Dynamic creation of transaction objects
  - Extensible classes for verification components and transactions
  - Features to avoid simulation races between test bench and DUT
- But can be a steep learning curve for RTL/HDL designers
- Some sort of framework for block-level test benches would help …

Incisive Plan-to-Closure Methodology

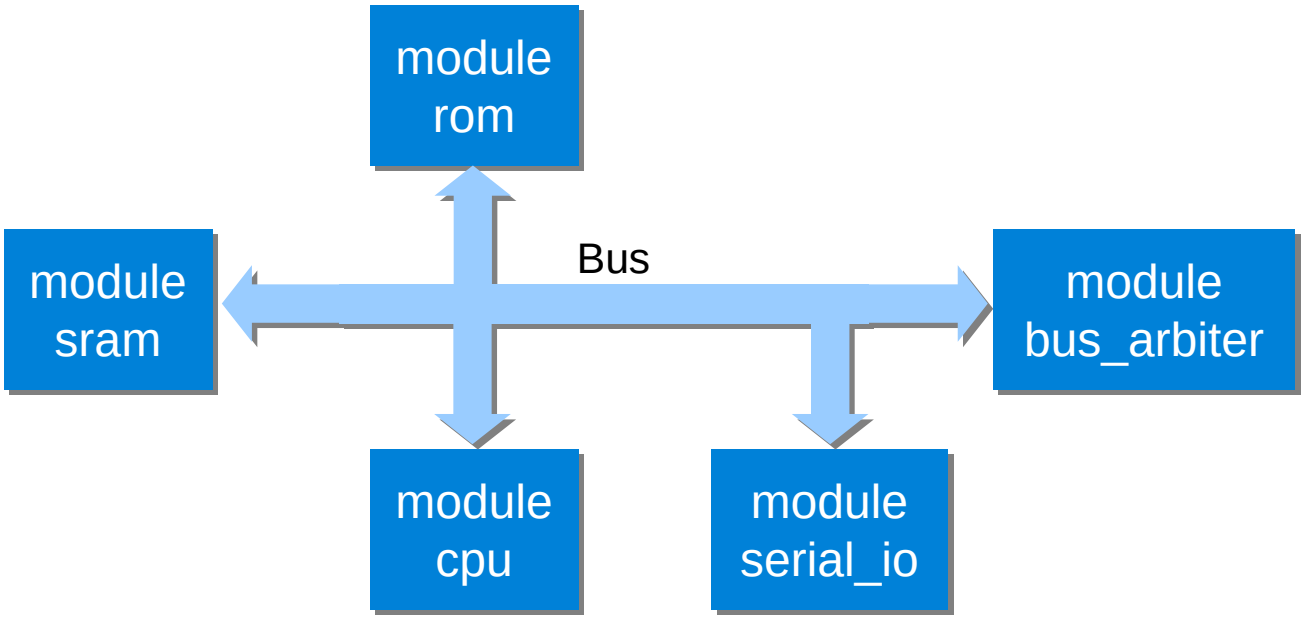
- URM (Universal Reuse Methodology) / “Design Team Verification”

- Module-Based with Classes URM
  - SystemVerilog verification environment
  - Classes for transaction objects
  - Does not require expertise in OOP (polymorphism, virtual methods, etc)
  - Guidelines to create reusable Universal Verification Components (UVCs)

- Our experience of actually getting this to work ...

System to be Verified

- Bus-based sub-system with CPU
- CPU, Serial IO and arbiter are RTL SystemVerilog



UVC used with Verification Plan

- Identify key requirements and verification strategy for each requirement **before** starting to develop testbench!

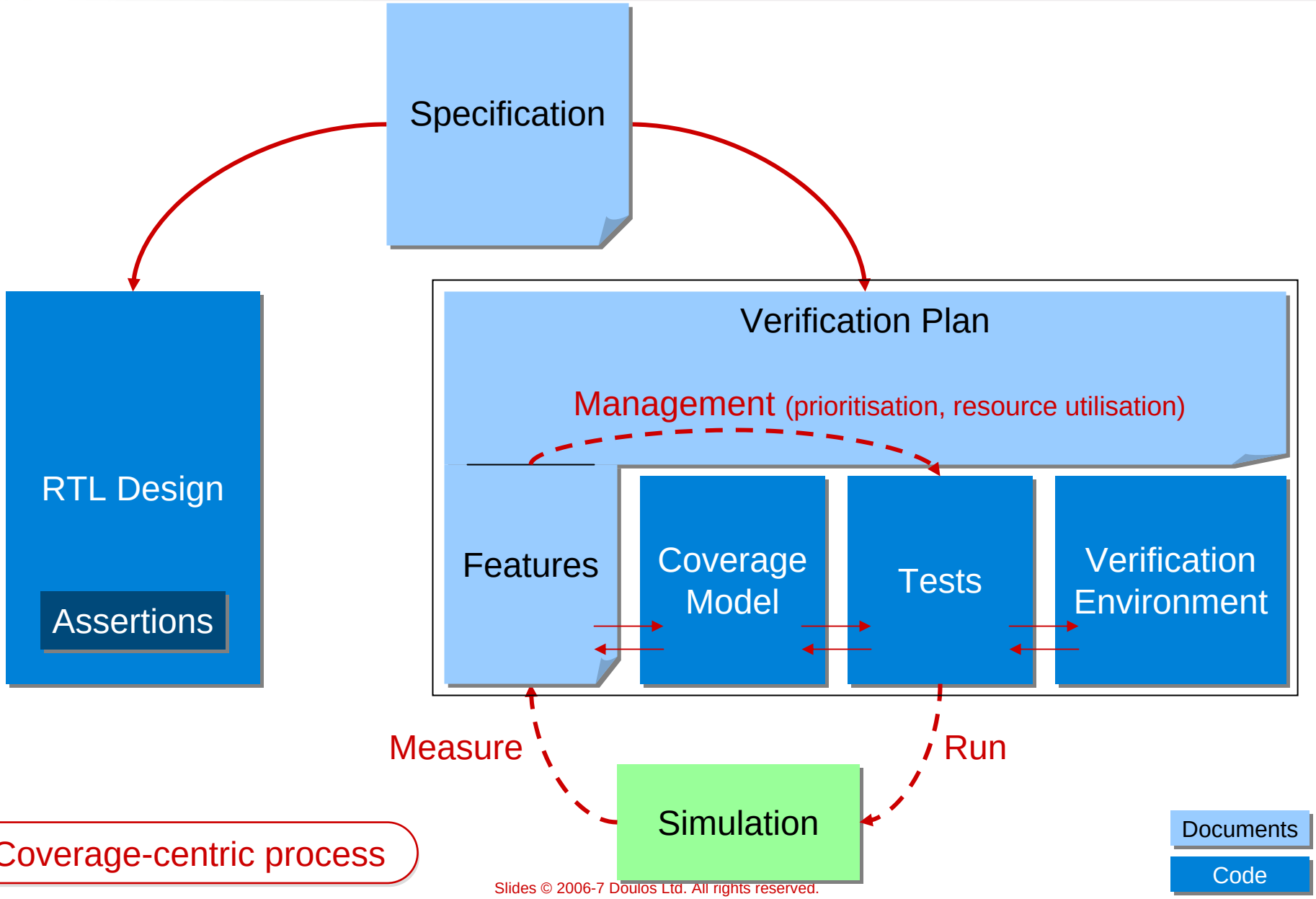
- Assertion-Based Verification - uses SystemVerilog `property` to continuously check design behaviour, e.g. interface protocol

```
Following a write there must not be a read for at
least 1 clock cycle.

The write enables should each stay high for only
one clock cycle
```

- Coverage-Driven Verification - uses SystemVerilog `covergroup` to record how many times a condition has been met

```
Forward and backward jump operations for every step
in the range 1 to 15 should be exercised.
```

The Verification Process

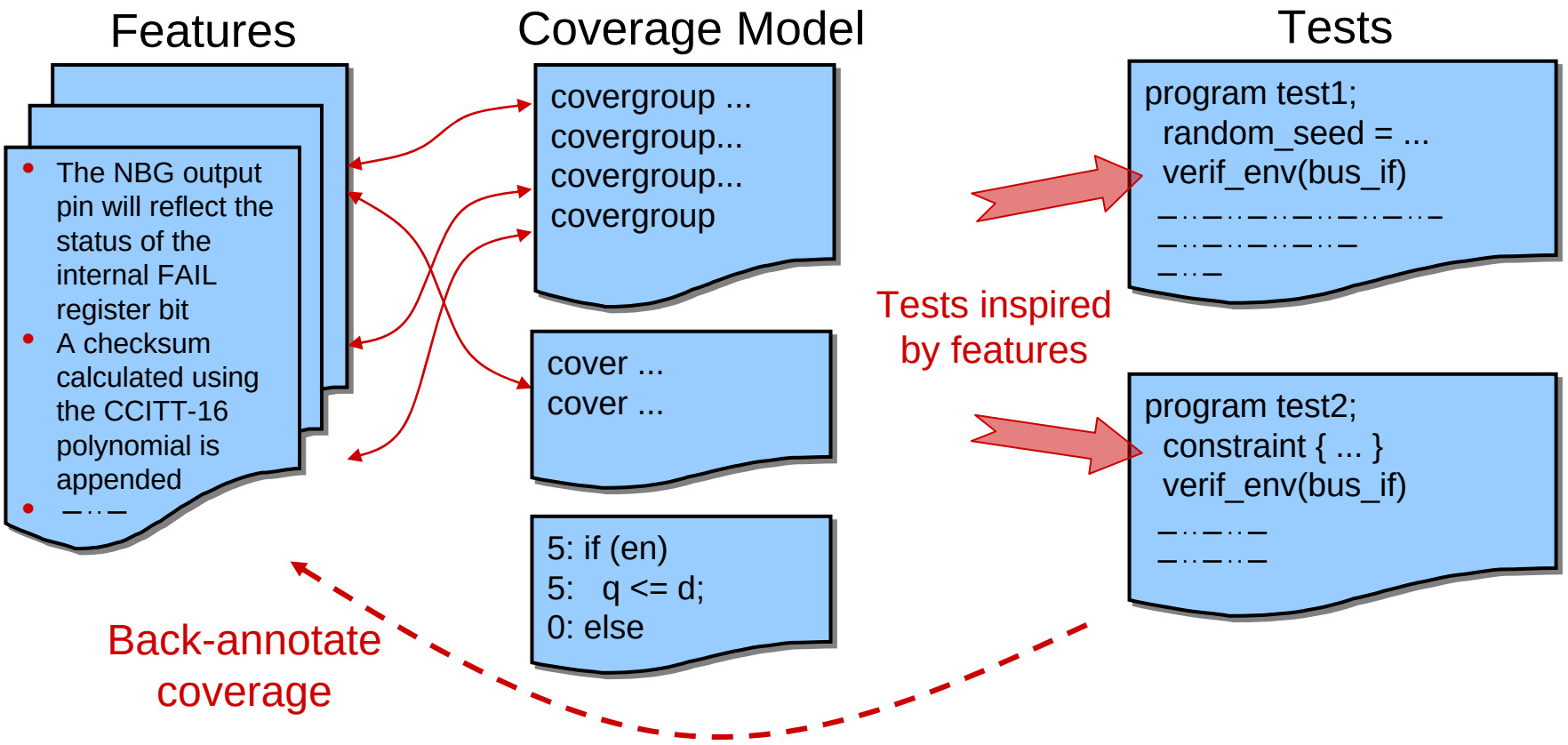


Coverage-centric process

Documents

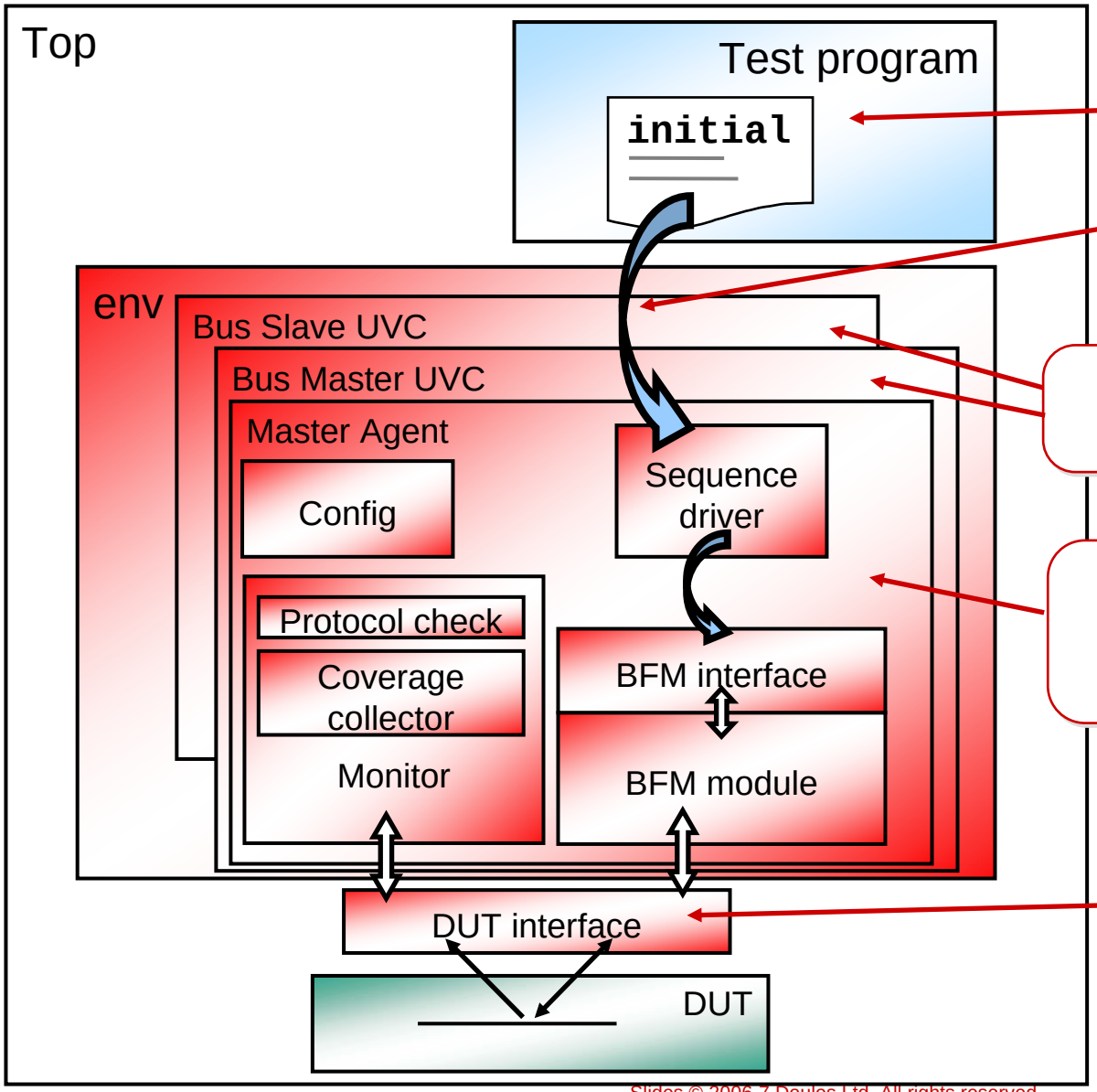
Code

From Features to Tests



- Features grouped by specification, implementation, and functionality
- Many-to-many mapping between features and coverage points
- Coverage and tests implemented in verification language or scripts
- Grade tests by coverage achieved, bugs found, run times etc

URM Testbench Structure



Test definition

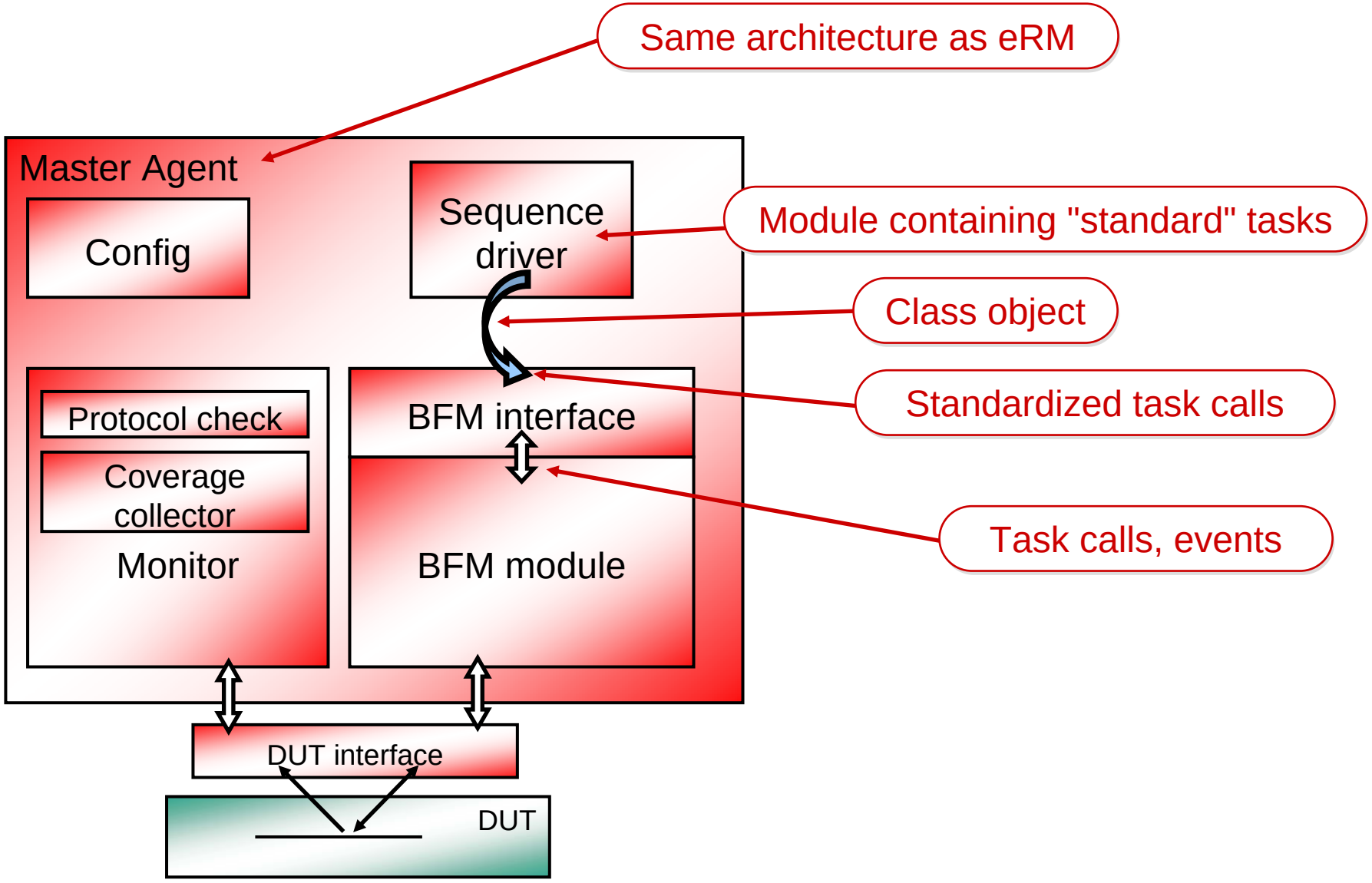
Sequence task calls

Environment contains multiple UVC modules

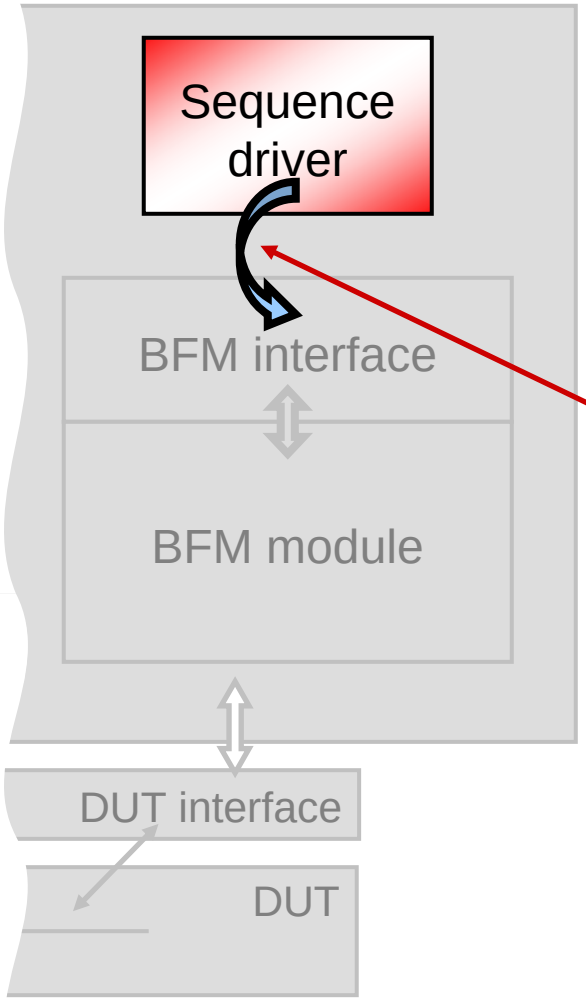
Protocol-specific "Interface UVCs" built from 1 or more agents

Interface encapsulates bus signals

Agent in SystemVerilog URM



Sequence Pushes Transactions



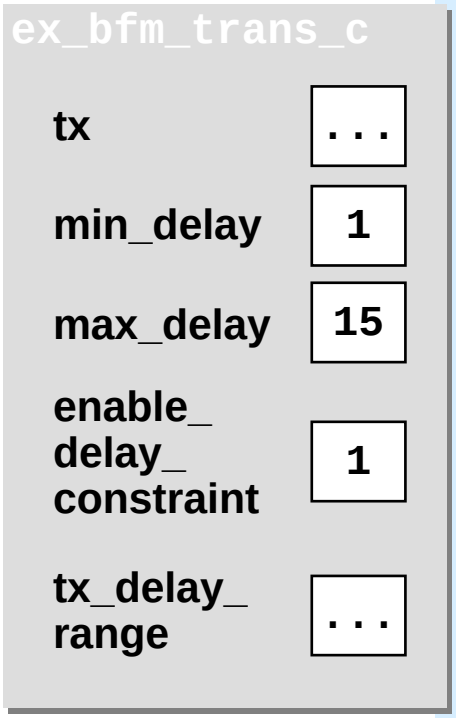

```
module ex_seq_driver_m
    ( ex_bfm_if bfm_if );
  ex_bfm_trans_c cur_trans;

  task simple ( ... );
    ex_bfm_trans_c trans;
    trans = new cur_trans;
    assert(trans.randomize());
    bfm_if.put ( trans );
  endtask : simple

  task scenario_x ( ... );
    ...
    simple ( ... );
    ...
```

Template for new transactions

Independent of BFM details

Sequence built from other sequences

Transaction Class




```
class ex_bfm_trans_c;
  rand ex_bfm_trans_s tx;

  int min_delay;
  int max_delay;
  bit enable_delay_constraint;

  constraint tx_delay_range {
    if (enable_delay_constraint)
      tx.delay inside{[min_delay:max_delay]};
  }

  function new();
    min_delay = 1;
    ...
endclass: ex_bfm_trans_c
```

Transaction data struct

Control

Constraint

Constructor

Derived Transaction Class

```
class derived_trans_c extends ex_bfm_trans_c;

  constraint tx_delay_odd {
    if (enable_delay_constraint)
      tx.delay[0] == 1;
  }

  function new();
    super.new();
    ...
```

Additional constraint

Calls base class constructor

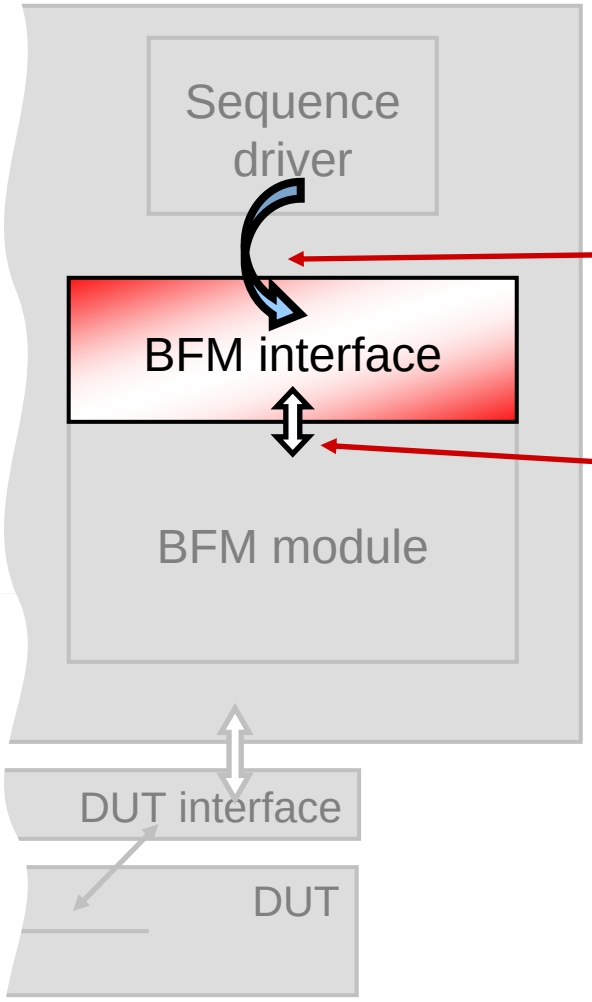
```
task simple ( ... );
    ex_bfm_trans_c trans;
    trans = new cur_trans;
    assert(trans.randomize());
...
```

Handle can point to derived class

Shallow copy

Includes derived class constraints

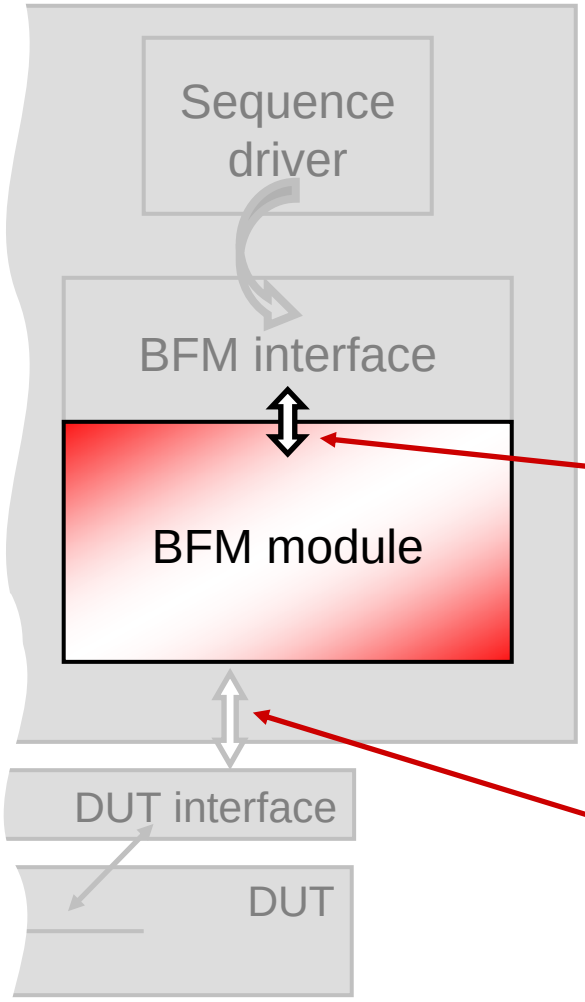
The BFM Interface




```
interface ex_bfm_if;

  ex_bfm_trans_s trans;

  task automatic put
       ( input ex_bfm_trans_c T );
    ...  endtask
  task automatic get
       ( output ex_bfm_trans_c T );
    ...  endtask
  task automatic done
       ( input ex_bfm_trans_c T );
    ...  endtask
...
```

Standardised access to any BFM copied from URM library

BFM Pulls Transactions




```
module ex_bfm_m (
    ex_bfm_if bfm_if ,
    input wire DUT_clk ,
    ... );
  ex_bfm_trans_c trans;

  initial
    forever begin
      bfm_if.get( trans );
      drive_transaction( trans );
      bfm_if.done( trans );
    end

  task drive_transaction
       ( input ex_bfm_trans_c T );
    ...
```

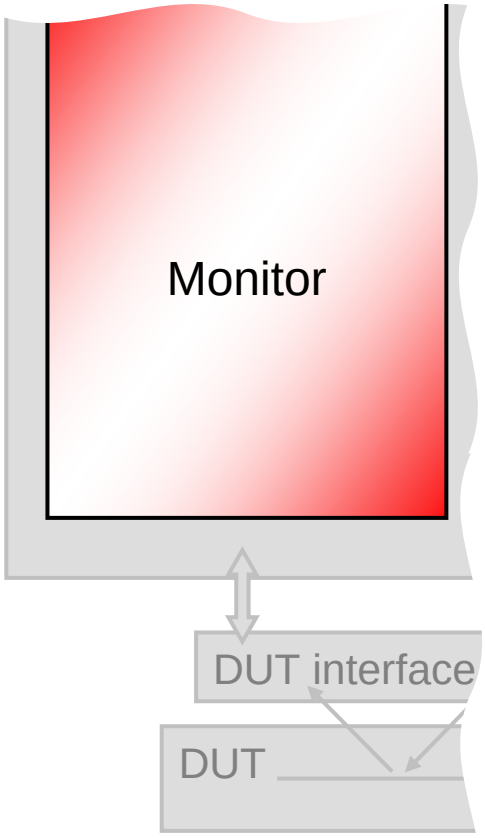
Interface

DUT signals

Blocking

Signal wiggles

Monitor Detects Bus Activity

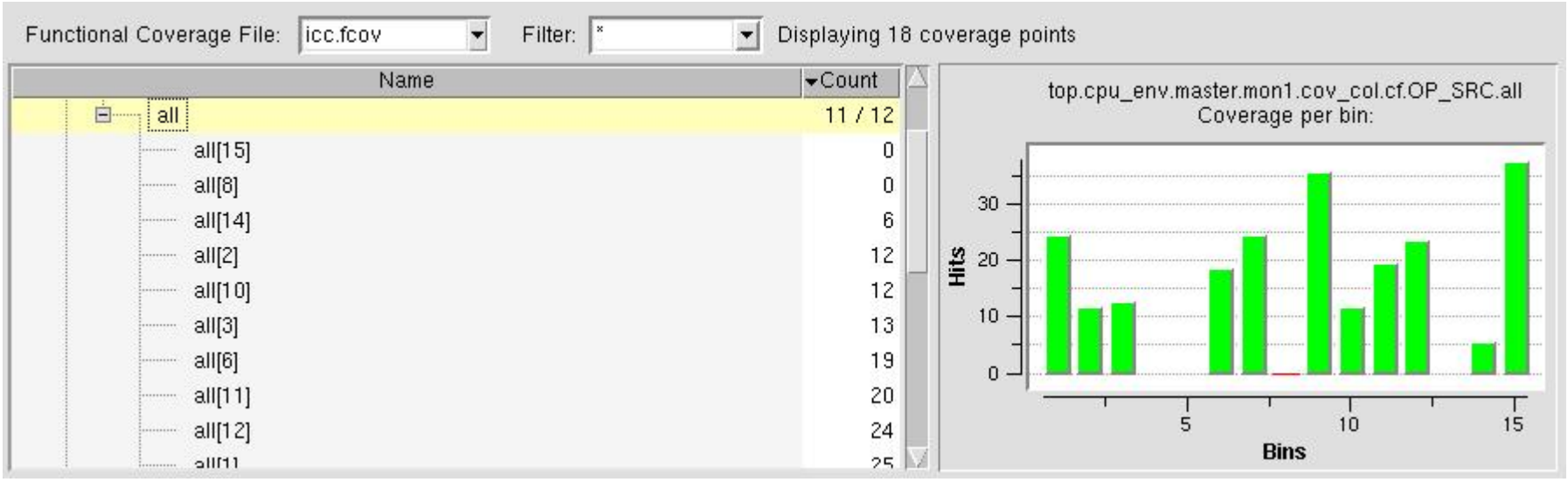



```
module ex_bus_mon_m (
  input wire DUT_clk ,  ... );

  mem_write: cover property (
    @(posedge DUT_clk)
      first_match(
        bus_if.wr
        ##1[1:$]
        bus_if.wack )
  ) cover_mem_wr(bus_if.addr, ...);

  property after_we;
    @(posedge bus_if.clk) disable iff
              (bus_if.reset)
    bus_if.we |=> (
       !bus_if.we && !bus_if.re);
  endproperty: after_we
  assert property (after_we);
```

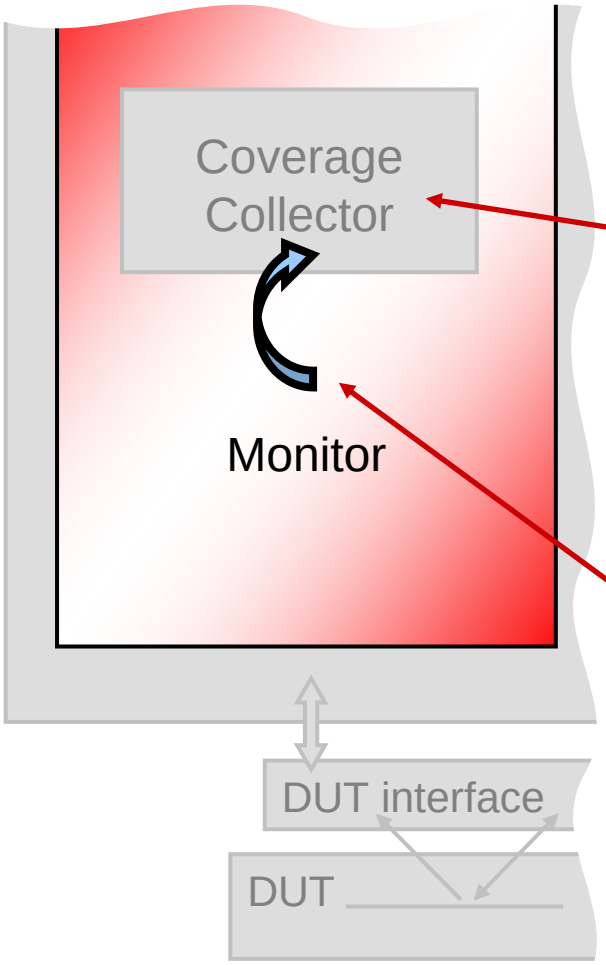
Called when sequence matches

Functional Coverage

- Coverage points do not reveal how conditions are met
- SystemVerilog `covergroup` can record sampled value occurrence in `bins` and cross-coverage between `coverpoint` pairs to measure “Functional” coverage
- Performed by coverage collector module



Writing to Coverage Collector



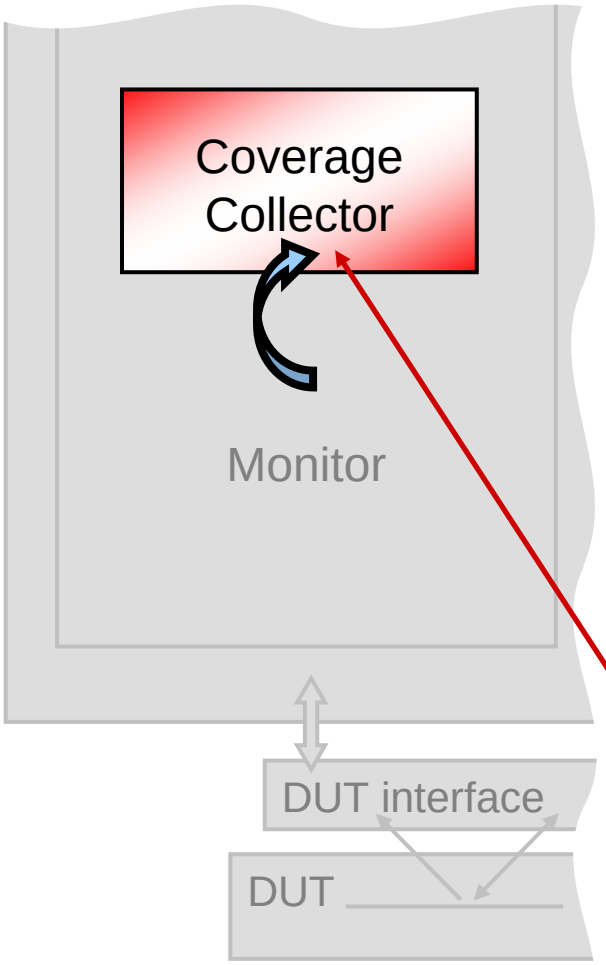

```
module ex_bus_mon_m (
  input wire DUT_clk ,  ... );

  ex_trans_fifo tx();

  ex_cov_collector_m c1(.trans(tx));

  function void cover_mem_wr(...);
    mem_trans_t tdata;
    ...
    tx.put(tdata);
  endfunction
  ...
```

Interface "channel"

Called when sequence matches

Call interface method

The Coverage Collector



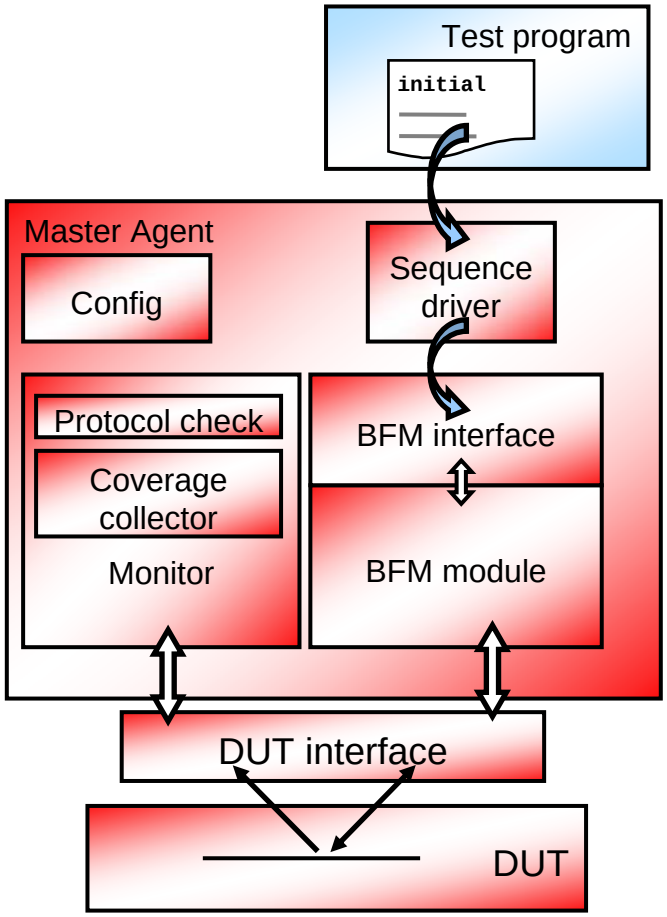

```
module ex_cov_collector_m (
       ex_trans_fifo trans);
  addr_t addr_cp;
  mode_t rw_cp; ...

  covergroup cov_mem_acc;
    mon_addr: coverpoint addr_cp {
      bins all[] = { [0:15] };
      ignore_bins bad = {0,5,8}; }
    ...
    mem_w: cross mon_addr, mon_rw;
  endgroup

  cov_mem_acc mem_c = new;

  always
  begin ...
    trans.get(tx);
    addr = tx.addr; ...
    mem_c.sample();
```

Mirror registers

Instance

Blocking

program vs. module



SystemVerilog `program`-level code:

- `program`
- Randomization, classes
- Postponed sampling and updating
- Dynamic creation of objects

SystemVerilog `module`-level code:

- `module`, `interface`
- Verilog simulation semantics
- Static object hierarchy

Test Configuration and Control

```
program test_p();
initial
begin
  randcase
    60: begin
      derived_trans_c t;
      t = new;
      env.master.a0.seq_inst.set_generated_trans(t);
    end

    40: begin ...
  endcase
  start_test();
  ...
  #100;
  end_test();
```

Transaction template

Sequence generator

Alternative test cases

URM Package methods

Conclusions

- Testbench architecture consists of modules plus a program
  - Coding style similar to Verilog and VHDL
  - DUT, BFM and sequence driver are independent
- Classes used for transactions only
  - Test program can pass randomized transactions to sequence driver
  - Does not require expertise in object-oriented programming
- Test program is compact and easy to modify
  - Uses "standard" infrastructure provided by UVCs
- Assertions check protocol throughout simulation
- Coverage from properties and covergroups recorded in coverage database

cadence™

cadence designer network

cdn LIVE™

CONNECT: IDEAS

CDNLive! 2007 Silicon Valley