Practical Experience in Automatic Functional Coverage Convergence and Reusable Collection Infrastructure in UVM

Roman Wang
roman.wang@amd.com

Suresh Babu & Mike Bartley
sureshbabu.p@testandverification.com
mike@testandverification.com

2014 DESIGN AND VERIFICATION DVCON CONFERENCE AND EXHIBITION EUROPE

The way to meet 100% functional coverage(FCOV)

FCOV Convergence

FCOV Collection Infrastructure

Is the 100% FCOV perfect?

Why is it *so* hard to meet 100% FCOV...

- to qualify all kinds of requirements for FCOV in different levels?
- to meet 100% FCOV in big regression faster?
- to architect FCOV collection infrastructure?
- to write tests well to contribute FCOV data?
- to make both FCOV and code coverage high?

Think about the FCOV requirements

- Verification environment

Test bench fault detect – functional qualification tools could do that well.

- IP Stand alone level

The features listed in IP design specification

Assertion coverage on the bus interfaces

Protocol Checker and VIP coverage

IP use cases in high layer

Power aware coverage if IP residents in different power domains

- SoC Level

The Scenarios listed in SoC design specification

Basic features of each IP

Memory allocation coverage

Performance Coverage (Ex. Fabric NoC, DDR and GPU, etc.)

How do you know FCOV is reaching 100%?

1. Verification engineers usually have to run regression with 10000 constraint random tests with different seeds in complex IP or SoC.
2. Merger the coverage database and find out the coverage holes.
3. Override the constrain or write direct cases to cover the holes in next regression.
4. Loop 2 & 3 to reach the 100% FCOV.

Cost of the legacy

1. More tests to run

More waste in time

More waste in LSF slots

More waste in disk

All of above are about **$$$.**

2. It’s difficult to point out how many times to run one test with different seeds to get closure.

Solution

- **The idea** is to exclude the items(considered to be covered) in next random iteration and set the closure flags if everything is walked.
- An **automatic functional coverage convergence method**.

  Two classes: fcov_smart_gen and fcov_smart_knobs

What’s fcov_smart_knobs?-1

- The fcov_smart_knobs is a unique control class and derived from uvm_object.
- fcov_smart_knobs class has two groups of knobs

1. Knobs to enable/disable the specific functional covergroup
2. Detailed possible knobs and closure knobs for specific functional covergroup

- fcov_smart_knobs class has the associative arrays to record the covered coverage item.   1—cover; 0 -- uncover
- fcov_smart_knobs class has total covering possibility of knobs for every coverpoint and cross coverage.

What’s fcov_smart_knobs?-2

Simple covergroup example:

```
Covergroup cg_example (string name);
    option.per_instance = 1;
    option.name = name;
    cp_point1 : coverpoint item1.value[0];
    cp_point2 : coverpoint item2.value[2:1];
    cp_point3 : coverpoint item2.value[4:3];
    cross1_point2_X_point3 : cross cp_point2, cp_point3;
endgroup
```

To meet 100% FCOV:

```
cp_point1 has 2 possibilities;
cross1_point2_X_point3 has 16 possibilities;
```

What's fcov_smart_knobs?-3

```
--Simple Example--

Class fcov_smart_knobs extend uvm_object;
//  closure knobs
bit cov_point1_closure , re_rand_cov_point1;
bit cov_cross1_closure, re_rand_cov_cross1;

// The total number of possibility
Int total_numofcov_point1;
Int total_numofcov_cross1;

// Associative array to record the covered coverage item
bit cov_point1[bit] ;  // the coverpoint which is NOT involved in the cross
bit cov_corss1 [bit [3:0]];

endclass
```

What’s fcov_smart_gen?-1

- The fcov_smart_knobs is a unique stimulus generator class and derived from uvm_object.
- It defines all random items which contributes to FCOV
- It has pre/post_randomize hooks.

1. In the pre_randomize function

Disable the random item if its coverage is already closure.

2. In the post_randomize function

Check if the randomized data already exists in covered array.

Set the re_rand knobs if it’s covered in the past iterations.

Check if the related coverpoint or cross coverage closure

What’s fcov_smart_gen?-2

```
class fcov_smart_gen extend uvm_object;
fcov_smart_knobs smart_knobs;

// the covergroup
Rand bit clk_mode; // separated cover point
Rand bit [1:0] dma_mode; // involve in cross
Rand bit [1:0] ds_mode;  // ds_mode need to be crossed with dma_mode

function void pre_randomize();
if(smart_knobs.cov_point1_closure)
    clk_mode.rand_mode(0);
if(smart_knobs.cov_cross1_closure) begin
    dma_mode.rand_mode(0);
    ds_mode.rand_mode(0);
end
…….
endfunction
```

What’s fcov_smart_gen?-3

```
function void post_randomize();
// check if it need to re-randomize again
if(smart_knobs.cov_point1.exist(clk_mode))
    re_rand_cov_point1 = 1;
else begin
     re_rand_cov_point1 = 0;
     // record the covered items
     smart_knobs.cov_corss1[{dma_mode , ds_mode }] = 1;
end
if(smart_knobs.cov_cross1.exist({dma_mode , ds_mode }))
    re_rand_cov_cross1= 1;
else begin
    re_rand_cov_cross1= 0;
    smart_knobs.cov_point [{dma_mode , ds_mode }] = 1;
end
// check if the specific group reaches the closure.
if(smart_knobs.cov_point1.num() == smart_knobs.total_numofcov_point1)
    smart_knobs.cov_point1_closure = 1;
```

What will happen if the coverage is close to 99%?

- It's very difficult to hit the corner
- It will become very inefficient.
- The proposed solution. (Ex. Random items for one covergroup)

1. Define the configurable threshold in the fcov_smart_knobs.
2. Calculate the current gain (= hit numbers/total_numofcov_cross1) every iteration and check if it hits the thresholds (Ex. 98%)
3. If yes, abstract the uncovered items(Ex. 2%) to shuffle_array by walking associative arrays with all possibilities.
4. Disable related rand items and take item from shuffle_array every next iterations.
5. It will become faster to cover the remained uncovered items (Ex. 2%)

Functional Coverage Collection Infrastructure

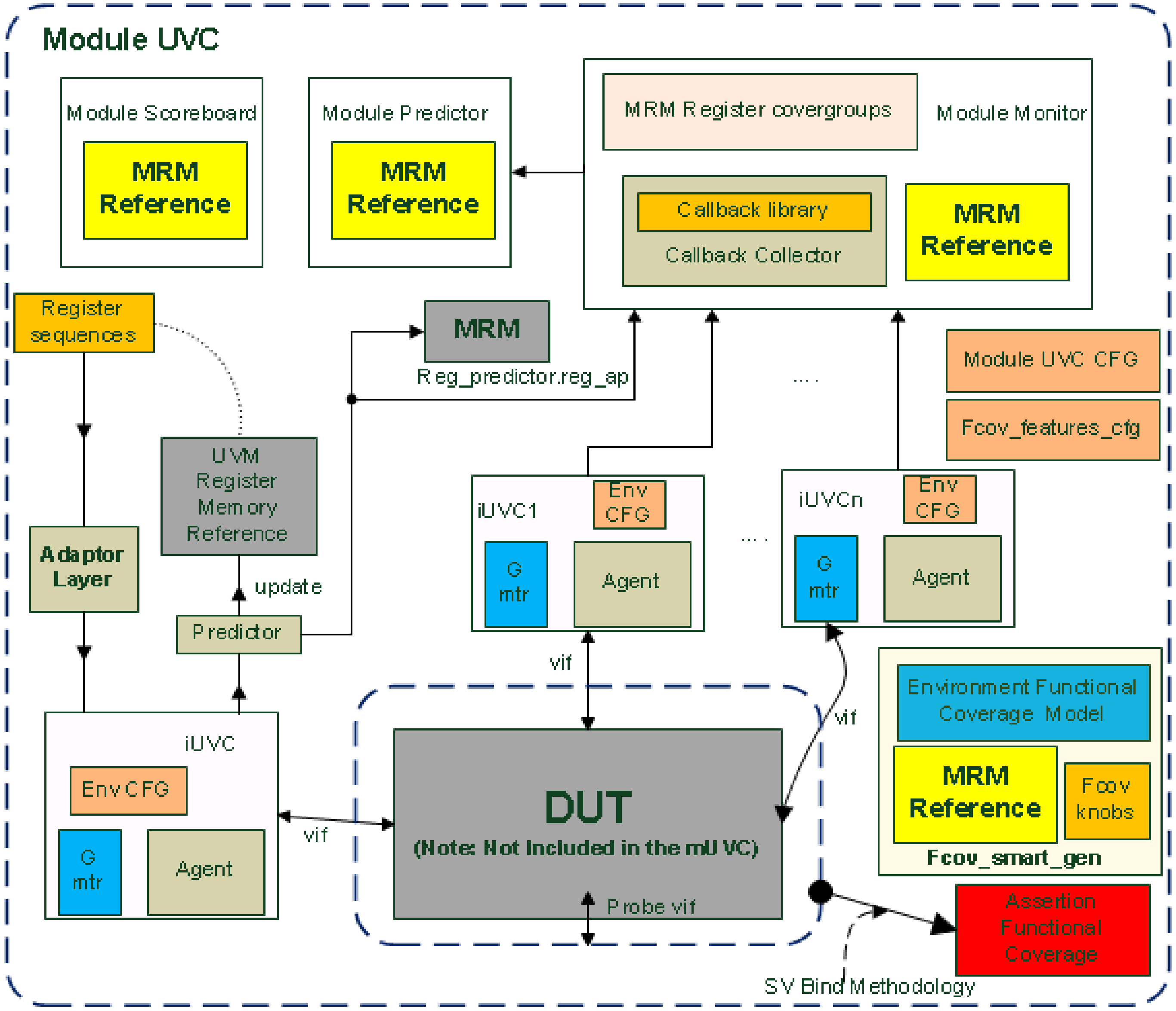
Reusable?
Scalable?
Controlable?

What we should do?

- Use UVM RAL model to write the functional covergroups and do collection in module monitor

  Implement a self-maintain RAL model (**MRM**)
- Use the transaction callback collection methodology with MRM reference
- Use the MRM built-in functional covergroups
- Write the assertions in the interface and bind to DUT
- Write and collect the environment configuration coverage in fcov_smart_gen
- Collect the VIP built-in functional coverage
- Predict the RO RAL registers in module predictor
- Why not use the back-door mechanism?
  - There are too many registers and paths to change at different levels
  - It would by-pass too much logic

FCOV Collection Infrastructure in IP Level UVM



Trick of the RAL coverage sample

- Define a uvm_analysis_imp port and connect it to bus2reg_mod_pdr (uvm_reg_predictor)’s reg_ap port.

```
// ------IP Standalone ::module UVC layer------
uvm_reg_predictor#(uvm_sequence_item) bus2reg_mod_pdr;

// build_phase
bus2reg_mod_pdr =
uvm_reg_predictor#(uvm_sequence_item)::type_id::create(“reg_pdr”,this);

// connect_phase
// connect the predictor to the bus agent monitor analysis port
iuvc_inst.agt.bus_mtr.dataph_ap.connect (bus2reg_mod_pdr.bus_in);
// connect the module monitor functional coverage collection to reg_ap
bus2reg_mod_pdr.reg_ap.connect(module_monitor.uvmreg_fcov_imp);
```

The transaction callback collection method

Three classes:

- Fcov_callback_knobs

It's a uvm_object, a collection knobs to provide granular control the callback objection creation, sample, weight, etc.

- Fcov_callback_library

It's derived from uvm_callback, a collection of transaction based callbacks extends the fcov_callback_base

- Fcov_callback_collector

It's a uvm_component, provides an analysis_imp port to subscribe the transactions from broadcaster(such as monitor)

fcov_callback_knobs/collector

```
class fcov_callback_knobs extends uvm_object;
// enable bit for coverage callback objection creation and sample
bit enable_cg_cov_dma                          = ENABLE;
// weight bit for each coverage point and cross within coverage callback obj
bit wt_cp_dma_mode                             = ENABLE;
bit wt_cp_sts                                  = ENABLE;
bit wt_cp_dma_msg                              = ENABLE;
bit wt_ dma_mode_X_sts_X_dma_msg= ENABLE;
```

```
class fcov_callback_collector extends uvm_component;
    uvm_analysis_imp#(uvm_transaction, fcov_callback_collector) fcov_cb_imp;
   `uvm_register_cb(fcov_callback_collector, fcov_callback_base)
 function new(string name, uvm_component pt);
 virtual function void write(uvm_transaction p);
   `uvm_do_callbacks(fcov_callback_collector, fcov_callback_base,
fcov_sample(p))
endfunction : write
```

fcov_callback_library

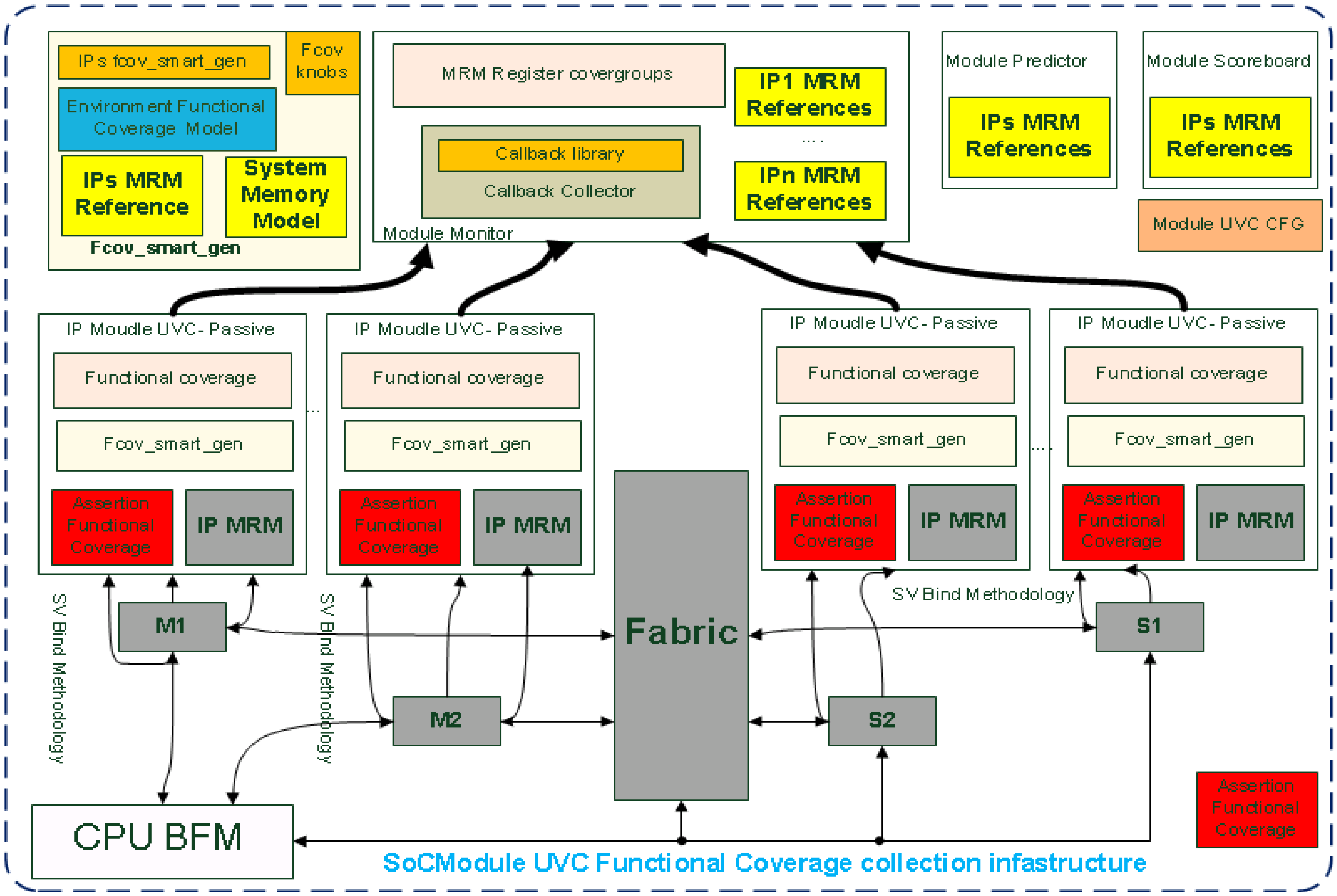
```
class **fcov_callback_base** extends uvm_callback;
  IP_UVM_RM **MRM_ref;**
 function new (string name = “”)  …
  pure virtual function void fcov_sample(ref uvm_transaction _ref);
  function void set_rm(IP_UVM_RM rm);
```

```
class dma_cov_cb extends fcov_callback_base;
  strap_pkt #(`IN_WIDTH,`OUT_WIDTH) dma_strap_pkt;
 covergroup cg_cov_dma (string name);
        cp_dma_mode : coverpoint mrm_ref.IP_BLK.reg1.dma_mode.value[0:1];
        cp_dma_msg : coverpoint tr.dut_outs.value[2]; // Transaction item
        dma_mode_X_dma_msg: cross cp_dma_mode, cp_dma_msg
         { ignore_bins ……;}
 endgroup : cg_cov_dma
 function void set_weight(bit dma_mod_wt, bit sts_wt,…..);
       cg_cov_dma.cp_dma_mode.option.weight = dma_mod_wt;
 virtual function void fcov_sampe(ref uvm_transaction _ref);
```

Transaction callback use model in module monitor

```
type_def uvm_callbacks #(fcov_callback_collector, fcov_callback_base)
cbtype;
uvm_analysis_port #(uvm_transaction) ip_fcov_cb_analysis_port;
fcov_callback_collector fcov_cb_clt;
fcov_callback_knobs  fcov_cb_knobs;
dma_cov_cb dma_cov_cb_inst;
fcov_cb_clt = fcov_callback_collector::typy_id::create(“fcov_cb_clt”,this);
fcov_cb_knobs = fcov_callback_knobs::typy_id::create(“fcov_cb_knobs”,this);
// build_phase
if(fcov_cb_knobs.enable_cg_cov_dma == ENABLE) begin
  dma_cov_cb_inst =new(“dma_cov_cb_inst”);
  dma_cov_cb_inst.set_rm(mrm_ref);
  dma_cov_cb_inst.set_weight(fcov_cb_knobs. wt_cp_dma_mode, …);
  cbtype::add_by_name(“fcov_cb_clt*”, dma_cov_cb_inst,this)
// connect_phase
ip_fcov_cb_analysis_port.connect(fcov_cb_clt. fcov_cb_imp);
// in specific write function
ip_fcov_cb_analysis_port.write(tr_pkt);
```

Vertical reuse from IP to SoC



Horizontal reuse

Define a fcov_feature class to control the knobs (fcov_smart_knobs and fcov_callback_knobs) by different projects.

```
// ------ fcov_feature class ------
bit cov_app_en;
string project_name ="";
function void init_knobs();
  if(project_name == "future") cov_app_en = ENABLE;
  if(project_name == "past") cov_app_en = DISABLE;

// ------Test layer ------
fcov_feature fcov_fea;
fcov_fea = fcov_feature::type_id::create("fcov_fea");
uvm_cmdline_proc.get_arg_value("+PROJ_NAME=",
fcov_fea.project_name);
fcov_fea.init_knobs(); // initialize the knobs by project specific
// ------ additional option on the command line ------
+PROJ_NAME=future
```

Summary

- Practical automatic functional coverage convergence method could dramatically reduce time and verification resource.

- Functional coverage collection infrastructure could be easily reusable and scalable from IP to SoC. The effort to maintain could be dramatically minimized.

Questions?

Thank you !

Roman Wang
roman.wang@amd.com