2014 DESIGN AND VERIFICATION DVCON CONFERENCE AND EXHIBITION EUROPE

Practical experience in automatic functional coverage convergence and reusable collection infrastructure in UVM verification

Roman Wang, +8613482890029, Advanced Micro Devices, Inc., Shanghai, China
(roman.wang@amd.com)

Suresh Babu Pusphaparaj, +8201086830487, Test and Verification Solutions, Inc., India
(sureshbabu.p@testandverification.com)

Abstract— Functional coverage is the main metric for measuring stimulus quality in a metric driven verification (MDV). Verification engineers use it to close off the complex design verification tasks. The constraint random tests will usually cover more comprehensive verification points than traditional directed tests. To get coverage closure, large numbers of simulations tend to be required. It is then necessary to merge the coverage data from those simulations which can take a significant amount of time and effort only to release the total coverage which may still not be 100%. The engineer will therefore need to override the constraints, rerun the tests and merge again to get a complete report as they iterate to achieve the required coverage closure. Even if the functional coverage is 100% based on your current coverage model, it is still reasonable to ask "is anything else still missing in the coverage model?" and "are you confident of your functional coverage convergence?"

The Universal Verification Methodology (UVM) has experienced great adoption and growth for three years, and it's the most widely used methodology for building reusable and scalable verification environments. The functional coverage is built and collected from the block to SoC level, so the challenge of reusability and controllability of UVM functional coverage infrastructure becomes critical and important.

Keywords— Functional Coverage, UVM, Coverage Covergence, Reusable Infrastructure

I. INTRODUCTION

Today's RTL is becoming more and more complex and big, however the limited time-to-market needs reduces the ASIC/SoC verification time. How could we have enough confidence for functional verification closure? The functional coverage is a very important and key metric in verification methodology. UVM well defines an advanced methodology to help users create reusable and scalable test benches to verify IPs for complex SoCs. End users write constraint random sequences to drive the design and collect coverage to make sure of convergence as early as possible. When we qualify the functional coverage, we should think about following items at different levels.

--Verification environment requirement:

1. Test bench fault detect

--IP stand-alone level requirements:

1. The features listed in IP design specification
2. Timing coverage on the bus interface
3. Protocol checking and VIP coverage
4. IP use cases in high layer
5. Power aware coverage (NLP)

--SoC level requirements:

1. The scenarios listed in SoC design specification
2. Basic functions of each IP
3. Memory allocation coverage
4. Performance coverage (Ex. Fabric NoC or DDR)

This paper outlines the functional coverage challenges we faced in terms of fast convergence, efficient verification resources (LSF slots, simulation time and disk space), reusability and controllability. The paper discusses proposed methods to make automatic functional coverage convergence and introduces a reusable functional coverage collection infrastructure from the IP level to the SoC level.

II. AUTOMATIC FUNCTIONAL COVERAGE COVERGENCE METHOD

Getting functional coverage convergence in a short time is hard. When making efforts to get functional coverage convergence, several challenges are usually met. We have created many constraint random tests to cover a more comprehensive verification than traditional directed tests. To reach 100% functional coverage with a large regression, we have to run each test with several different seeds (the number is usually difficult to qualify and given experience value). It will lead to a huge number of test threads in the regression, and then it needs more server LSF slots, regression time, disk space and will generate many coverage databases which will cause the database merging to be a little slow. In most cases, the total functional coverage still might not reach 100%. We have to override the constraints and rerun a specific test like directed test to cover the holes. And finally, we have to merge the coverage again. This definitely wastes a lot of time, LSF slots and additional effort, all of which are about money. In this paper, to address this challenge, we propose one automatic functional coverage convergence method and it should be faster than ever before. In our method, we expect to exclude the items (which are considered to be already covered) in the next random iteration and set the closure flag if all of the rand items are marked covered.

```
Class fcov_smart_knobs extend uvm_object;
Bit total_num4cov_point;
Bit total_num4cov_cross1;
// Associative array to record the covered coverage item
bit cov_point1[bit] ;  // the coverpoint is not involved in the cross
bit cov_corss1 [bit [3:0]];
bit cov_point1_closure , re_rand_cov_point1;
bit cov_cross1_closure, re_rand_cov_cross1;
Endclass

Class fcov_smart_gen extend uvm_object;
Fcov_smart_knobs smart_knobs;
// the covergroup is make of clk_mode, dma_mode and ds_mode
Rand bit clk_mode; // separated cover point
Rand bit [2:0] dma_mode; // involve in cross
Rand bit ds_mode;  // ds_mode need to be crossed with dma_mode
//decides the covergroup closure.
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

function void post_randomize();
// check if it need to re-randomize again
if(smart_knobs.cov_point.exist(clk_mode))
re_rand_cov_point1 = 1; else begin re_rand_cov_point1 = 0;
smart_knobs.cov_point[clk_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_corss1[{dma_mode , ds_mode }] = 1; end
// check if the specific group reaches the closure.
if(smart_knobs.cov_point.num() == smart_knobs.total_num4cov_point)
smart_knobs.cov_point1_closure = 1;
if(smart_knobs.cov_cross1.num() == smart_knobs.total_num4cov_cross1)
smart_knobs.cov_cross1_closure = 1;
//something print for further debug
// sample the environment functional coverage.
endfunction
```

Example 1: fcov_smart_knobs and fcov_smart_gen

We define two classes (fcov_smart_gen and fcov_smart_knobs) to implement our method.

- The fcov_smart_knobs is a unique granular control class. The items in fcov_smart_knobs are divided into two groups: 1. Knobs to enable and disable a specific functional group considering not all of the rand items need to be randomized in every iteration. 2. Detailed possible knobs and closure knobs for specific functional groups. The user could change the knobs at runtime.

- The fcov_smart_gen is a unique stimulus generate class and defines all random items and closure flags. You could implement the environment functional coverage, and the type of functional coverage that will be implemented and collected in the infrastructure mentioned in following section. It defines two groups of an associative array table according to each functional coverage to record the covered items. One is for a separated coverpoint, another is for cross coverage.

  The value in an associative array indicates the covered (1) or uncovered (0). It should pre-calculate the total number of possibilities for every table based on the related functional coverage and fill the related possible knobs in the fcov_smart_knobs. The functional group closure knob usually depends on all associative arrays' value being at 1 (means covered) for one functional group. If this is the case, then the closure knob will be set to 1. Our plan is to make a cross coverage closure first and then make sure of the separated one.
  In the post_randomize function, it sets the data (composed of randomized items) to related associative array as the "index" and set 1 to the related index, check if it need to be randomized again, check if the specific group reaches the closure and set the related knobs. It also makes the display and samples the environment functional coverage. In the pre_randomize function, it makes rand items nonrandom based on the closure knobs.
  In this paper, we present an algorithm that controls the random items and the distribution to disable covered items. The implementation complexity depends on functional coverage.

```
// ------The sequence layer-----
fcov_smart_gen smart_gen;
// body
If(!smart_gen.smart_knobs.cov_cross1_closure) begin // priority one
  do
     smart_gen.randomize();
  while(smart_gen.re_rand_cov_cross1);  end
else if(!smart_gen.smart_knobs.cov_point1_closure) begin // priority two
  do
     smart_gen.randomize();
  while(smart_gen.re_rand_cov_point1);
end
```

Example 2: How fcov_smart_gen Works in Sequence

Current implementation has a little flaw when the functional coverage is closed to 99%. It's very difficult to hit the remained 1% corner and becoming very inefficient. To address this issue, we proposed following improvements. Ex. there are some random items which contribute to one specific functional covergroup.

1. Define the configurable threshold (Ex. 98%) in the fcov_smart_knobs.
2. Calculate the current gain (= hit numbers/total_numofcov_cross1) in every iteration and check if it hits the thresholds or not.
3. If yes, abstract the uncovered items (Ex. 2%) to shuffle_array by walking associative arrays with all possibilities.
   Disable related rand items and take item from shuffle_array in next iterations.
4. It will become faster to cover the remained uncovered items

The fcov_smart_gen is created on the test layer (base test) and retrieved on the sequence layer (base sequence). We use the priority-based mechanism to decide how random it is in sequence. In the example, we first make sure of the cross coverage closure and then each separated cover point. If the re_rand_* knobs is set, it means this round of randomized data has been covered before and should be randomized again. You will get the new random data within an exclusive space compared with the past iteration. The functional coverage inside fcov_smart_gen could be reused if the sequence will be reused as well. You could reuse the fcov_smart_gen and fcov_smart_knobs in high layer, enable the interested rand item and related cover items by high level knobs configuration.

III. REUSABLE AND CONTEOLABLE UVM FUNCTIONAL COVERAGE COLLECTION INFRASTRUCTURE

A. IP Level functional coverage collection infrastructure

Before we start to create and collect the IP specific functional coverage, it's really important to think about how to qualify the UVM test bench code, the size and complexity of which often dwarfs the size of the RTL design to be verified. We verify the design assuming a perfect testbench, but the coding mistakes are equally likely in the testbench as the design, therefore the quality of the test bench always affects the quality of verification. A good testbench must be able to measure the environment ability to activate, propagate and detect bugs within the design.

The functional qualification tools are intended to address this challenge and fill the gap in our verification environment, and is out of the scope of this paper.

Figure 1 depicts functional coverage collection infrastructure in UVM IP Standalone level. There are two UVM register models in the module UVC layer. One is a register model reference (**RMR**), built in test layer and used for register sequences to program the DUT registers; another is the module register model (**MRM**, built in module UVC layer, and its reference is assigned to fcov_smart_gen, module monitor, module predictor and module scoreboard. The **RMR** has the downstream (reg2bus function is to translate uvm_reg_item to bus_item) and upstream (bus2reg function is to update register model) data path, but **MRM** only has upstream data path to update UVM register model called a "self-maintained attribute". When module UVCs are reused to SoC level, if it doesn't adopt UVM register layering to program register (e.g. C++ test cases via DPI with interface VIP to drive system bus), the **RMR** doesn't exist in the high layer. That's two UVM register models are used. Every interface UVC (iUVC) broadcasts the bus transaction to module monitor which work for following functions, sample the **MRM** register covergroups, and trigger the sample function in callback collector which installs the callback library including the functional coverage combined transaction with **MRM**. The fcov_smart_gen is instanced in module UVC layer and contains the **MRM** reference, environment functional coverage model and fcov_knobs. The assertion functional coverage implemented in the System Verilog (**SV**) interfaces are connected to the design at top or internal layer through virtual interface using **SV** bind methodology to ensure the reusability. Fcov_feature_cfg is used to list the functional coverage knobs and callback functional coverage knobs' reference for vertical reuse or project specific horizontal reuse, and its reference is assigned to module_UVC_cfg class. The following items describe the detailed automatic and reusable functional coverage collection method based on the infrastructure mentioned above.

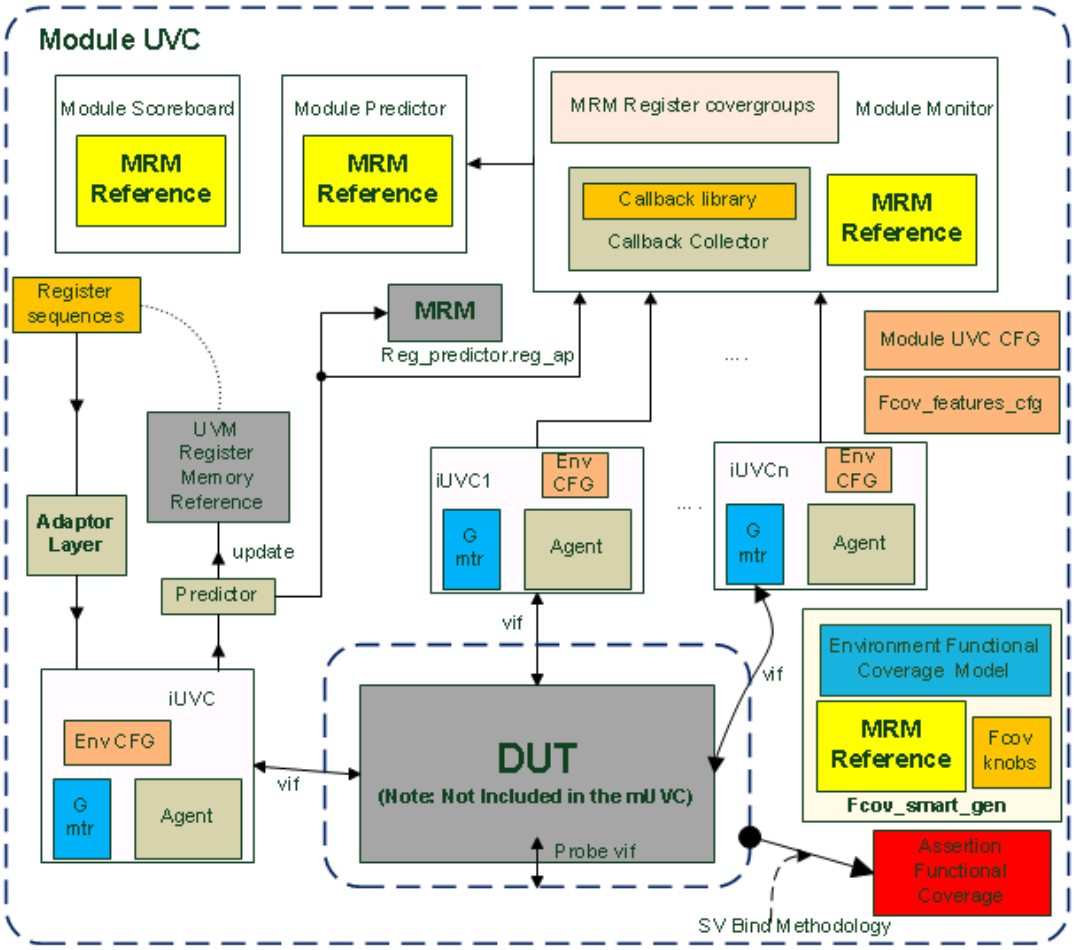

Figure 1: Functional Coverage Collection in UVM Standalone

*a) To collect IP features relative functional coverage using **MRM** register covergroups.*

In the module monitor layer, we create the functional covergroups based on **MRM** reference which could be updated after each bus register access**.** Every functional covergroup indicates the separated IP feature and gets sampled in the write function which subscribes uvm_reg_predictor's reg_ap analysis port. Covergroups in the write function that subscribes to the agent monitor analysis port in the interface UVC are not sampled because if the register bus transaction is broadcasted to the module monitor and uvm_reg_predictor at the same time, the mirror value of **MRM** will be updated after the covergroups get a sample. In the UVM library source code, we could see the reg.ap.write operation happens after **MRM** updates, so we expect to only use the reg.ap.write to trigger covergroups samples and drop the uvm_reg_item.

```
// ------ IP Standalone ::module monitor layer ------
uvm_analysis_imp_ uvmreg_fcov #(uvm_reg_item, ip_module_monitor)
uvmreg_fcov_imp;
extern function void write_uvmreg_fcov (ref uvm_reg_item p);
fcov_feastures_cfg fcov_fea_cfg;
// MRM register covergroup example
covergroup cg_cov_app (string name);
   Option.per_instance = 1;
   Option.name = name;
  cp_ds_mode : coverpoint mrm_ref.IP_BLK.reg1.ds.value[0];
  cp_clk_mode : coverpoint mrm_ref.IP_BLK.reg2.blk.value[1:0];
  cp_req_type : coverpoint mrm_ref.IP_BLK.reg1.req.value[0];
  dsm_X_clkm_X_req_type: cross cp_ds_mode, cp_clk_mode, cp_req_type
endgroup : cg_cov_app
// granular weight control
function void set_ cg_cov_app _weight(bit ds_mod_wt, bit clk_mode_wt,…..);
   cg_cov_app.cp_ds_mode.option.weight = ds_mod_wt;
   cg_cov_app.cp_ds_mode.type_option.weight = ds_mod_wt;
   …
endfunction
//  module_monitor::new(…)
cg_cov_app = new("cg_cov_app");
uvmreg_fcov_imp = new("uvmreg_fcov_imp",this);
// module_monitor::write_uvmreg_fcov(ref  uvm_reg_item p);
// use features reference to enable/disable specific covergroup sample
If(mod_cfg.features_ref.cov_app_en == ENABLE) begin
  set_ cg_cov_app _weight(fcov_fea_cfg._ds_mode_wt, ….);
  cg_cov_app.sample();  end
```

Example 3: Functional Coverage using **MRM**

```
// ------The write function in the uvm_reg_predictor------
adapter.bus2reg(tr,rw);
// pre_predict and rg.do_predict operations
// publish <uvm_reg_item> transactions converted from bus transactions
// received on ~bus_in~.
reg.ap.write(reg_item);
```

Example 4: The trick of triggering the coverage collection

```
// ------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);
```

Example 5: The connection in module UVC layer

*b) To collect IP features relative functional coverage using covergroups composed of bus transactions and **MRM**.*

The specific registers program may usually introduce the bus or signals changing on the top layer of the design. For example, when programming specific registers to trigger the design to perform the DMA transfer, the design may send out a message or toggle relative pins on the top layer of design after DMA transfer is complete. It's not enough to only qualify the **MRM** register coverage, because design may have a bug to block toggle to the bus even with the correct registers program. It would be better not only to separately cover the registers program and bus transaction (which stands for the message or pins' toggle), but also cover the proper crosses of them. In this case, the attribute of the registers is usually the read-only (**RO**) or write-1-to-clear (**W1C**). These registers in **MRM** reference could not be changed automatically to align with the design registers' change without the backdoor method. However we don't adopt the backdoor method in our UVM environments, we promote another way to update the RO and W1C related register model. The interface UVC's monitor will broadcast the transaction (which looks like the message or a set of pins in this example) to module predictor that could update the **MRM** registers/fields with special attribute (such as RO and W1C) via the call predict() function. The register checking is located in free-running event service routines (**ESR**). For example, when the environment captures the DMA complete message, the **ESR** could check if the DUT related **RO** and **W1C field** is set and write 1 to and read back from these fields to check if they could be clear well.

```
// ------ IP Standalone ::module predictor layer ------
// module_predictor::write_dma_pdr(ref uvm_transaction p)
strap_pkt #(`IN_WIDTH,`OUT_WIDTH) dma_strap_pkt;
$case(dma_strap_pkt,p);
If(dma_strap_pkt.dut_outs.value[2])
mrm_ref.IP_BLK.reg1.dma_sts.predict(dma_strap_pkt.dut_outs.value[3]);
```

Example 6: Predict the RO/W1C registers

In order to enable this kind of functional coverage collection, we implement three files: fcov_callback_collector, fcov_callback_library and fcov_callback_knobs.

The fcov_callback_collector class is the uvm_component, it provides an analysis_imp port to subscribe the transaction item from the analysis port inside the module monitor, registers the fcov_callback_base, and invokes the fcov_callback in the write function.

```
// ------ fcov_callback_knobs overview ------
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;
```

Example 7: fcov_callback_knobs

The fcov_callback_library class is the uvm_callback, a collection of user defined coverage callback extends the fcov_callback_base, contains the **MRM** reference and pure virtual function fcov_sp. In each user defined callback, it implements the covergroups composed of bus transactions and **MRM** and sample the coverage in the fcov_sp function.

```
// ------ fcov_callback_collector overview ------
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_sp(p))
endfunction : write
```

Example 8: fcov_callback_collector

The fcov_callback_knobs class is uvm_object, and a collection of knobs to provide granular control that enable bits for coverage callback objection creation, sample and weight bit for each coverage point and cross within the coverage callback object.

c) To collect IP features relative functional coverage on system point of view early.

In the IP level verification, we should consider how to cover the features of IP itself and how the scenarios it works in high layer, implement and verify early in IP standalone. I ever met one real design bug it escapes from 3 silicon tape out in my past experience. When IP performs the DMA transfer and moves the data from system memory to another memory model on a special memory interface, it needs to get the descriptor from system memory first and resolve it to let FSM move on correctly. However, when the clock of system memory bus is three times faster than the clock of special memory interface, the design's FSM dies due to the long wait time for descriptor causing a deadlock. We implement and sample this kind of functional coverage in the fcov_smart_gen.

```
//------ fcov_callback_library overview ------
class fcov_callback_base extends uvm_callback;
  IP_UVM_RM mrm_ref;
 function new (string name = "")  …
  pure virtual function void fcov_sp(ref uvm_transaction _ref);
  function void set_rm(IP_UVM_RM rm);
endclass
// user defined callback
class dma_cov_cb extends fcov_callback_base;
  strap_pkt #(`IN_WIDTH,`OUT_WIDTH) dma_strap_pkt;
 covergroup cg_cov_dma (string name);
   Option.per_instance = 1;
   Option.name = name;
  cp_dma_mode : coverpoint mrm_ref.IP_BLK.reg1.dma_mode.value[0:1];
  cp_sts : coverpoint mrm_ref.IP_BLK.reg1.dma_sts.value[0]; // RO register
  cp_dma_msg : coverpoint tr.dut_outs.value[2]; // Transaction item
  dma_mode_X_sts_X_dma_msg: cross cp_dma_mode, cp_sts, 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;
   cg_cov_dma.cp_dma_mode.type_option.weight = dma_mod_wt;
   …
endfunction
 function new (string _name = "dma_cov_cb");
   super.new();  cg_cov_dma = new("cg_cov_dma");
 endfunction : new
 virtual function void fcov_sp(ref uvm_transaction _ref);
   if(_tr.get_type_name() == "strap_pkt #(`IN_WIDTH,`OUT_WIDTH)") begin
     $cast(tr,_ref);  cg_cov_dma.sample();
   end
 endfunction
```

Example 9: fcov_callback_library

d) To collect IP register functional coverage using built-in covergroups in UVM register model.

The UVM register model could have the built-in covergroups which are determined by a local variable called m_has_cover. This variable is a type of uvm_coverage_model_e and it should be initialized by a build_coverage() call within the constructor of the register object which assigns the value of the include_coverage resource to m_has_cover. Once this variable has been set up, the covergroups within the register model object should be constructed according to which coverage category they fit into. There are two methods to sample the covergroups: sample() automatically on register access or as the result of an external sample_values() call. The sample() method is called automatically for each register and register block on each access. The sample_values() method is called by users somewhere in the testbench.

e) To collect IP assertion properties functional coverage.

The assertion is usually written to check bus protocol, design timing for critical paths, FSM transaction etc. The check is not enough, because assertion will not display an error if it isn't triggered. In most cases, users only see

the fail but not the pass, so it could miss this kind of bug. The assertion coverage helps the user qualify the assertions be triggered or not triggered. In the power aware (**PA**) simulation, following three types of assertion helps find the bugs faster. 1. Write the PA assertions to power ports that check the port values in different power modes. 2. Write PA assertions for the power mode transitions to find the mode transition related bugs faster. 3. Write power mode transition timing related **PA** assertions to find the bugs related to timings (e.g. the maximum and minimum time for different power mode transactions.). It's important to collect this kind of power aware assertion coverage. In our test bench, we create the assertions in the System Verilog (SV) interface and use the system Verilog binding methodology to design. The interface bind methodology ensures the assertions be reusable vertically and horizontally.

```
// ------ IP Standalone ::module monitor layer ------
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)
end
// connect_phase
ip_fcov_cb_analysis_port.connect(fcov_cb_clt. fcov_cb_imp);
// In different write_* functions, it broadcasts the specific bus tr_pkt to clt
ip_fcov_cb_analysis_port.write(tr_pkt);
```

Example 10: Callback method in module monitor

```
// ------ IP Standalone ::module UVC layer ------
// build_phase – all register coverage types enables
uvm_reg::include_coverage (UVM_CVR_ALL);
```

Example 11: Enable built-in UVMREG coverage

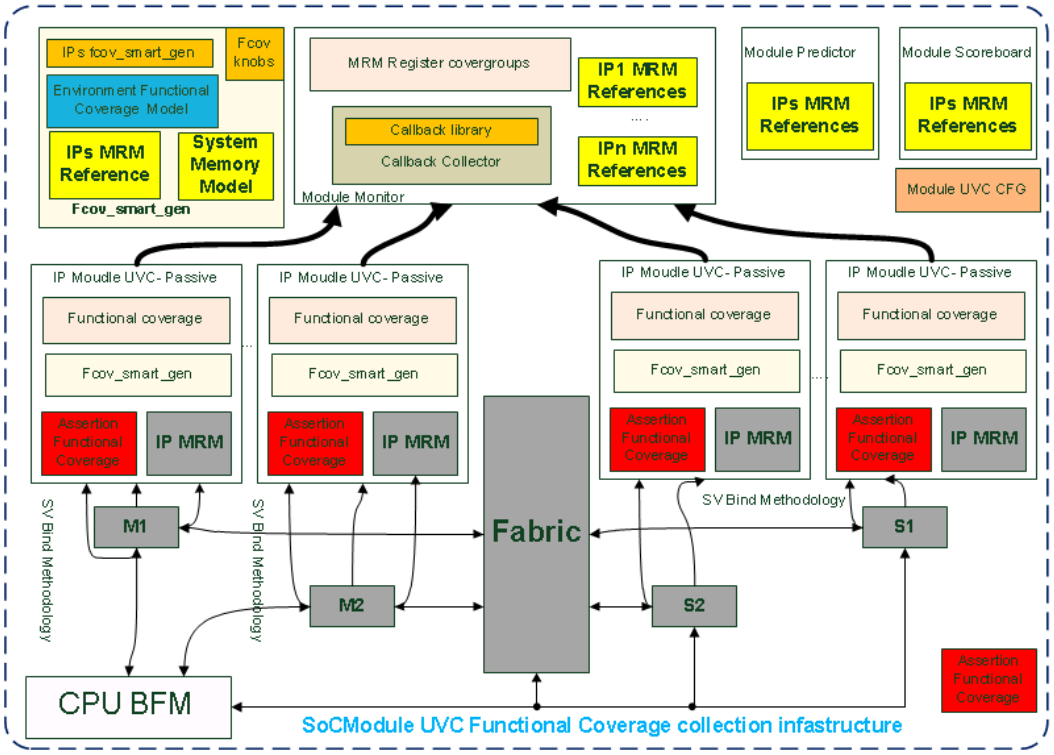

Figure 2: Reuse from IP to SoC

B. Vertical reuse functional coverage infrastrucutre to SoC

The SoC level functional coverage infrastructure looks similar to IP level (as pictured). All of the IP module UVCs will be integrated into an SoC module UVC as passive mode. The passive mode means separated IP module UVCs only help checking IP features and collecting functional coverage. The SV interface bind methodology ensures reusability without more changes. Every IP module UVCs provides the analysis ports to broadcast the transactions to the SoC module monitor where the SoC level functional coverage collection will happen. Each IP MRM reference is assigned to a module monitor, module predictor, module scoreboard and fcov_smart_gen, the user could use them to create the SoC level functional covergroups based on the SoC use cases. The users could change the knobs of fcov_feature_cfg and fcov_callback_knobs objects to get granular control and enable the interested functional coverage collection at the SoC level.

At the SoC level, the user may also create the soc_fcov_smart_gen and soc_fcov_knobs, and reuse the IP level fcov_smart_gen and fcov_knobs. It also contains the system memory model and functional coverage about that. We could also define some performance specific functional coverage to ensure the system could meet our requirements.

```
// ------ 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;
  endfunction
// ------ IP Standalone 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
```

Example 12: how to use the fcov_feature with command line

C. Herizonal reuse functional coverage infrastrucutre to other projects

The same IP design may be reused in different projects with selective features, so we have to make granular control to enable selective functional coverage. At the IP level fcov_feature class, we define a string property to indicate the current project and specify the coverage knobs for different project names. At the IP UVM test base layer, we use the UVM command line processor to get the project name on the command-line and set the project string to initial the fcov_feature object.

D. Beyond the functional coverage collection

When you get the coverage data including the functional coverage and code coverage after regression, you could use coverage analysis tool (such as: asureSign (TVS), vManager (Cadence), QVRM (Mentor) and DVE (Synopsys)) to evaluate test case quality and how many features covered for particular test case. The coverage report will show customer requirement details with corresponding features functional coverage numbers, it's very useful and necessary to review and help to refine the verification plan and functional coverage if any requirement is updated.

IV. CONCLUSION

Adopting the practical automatic functional coverage convergence method can dramatically reduce the number of tests in regression by at least 70% and the verification resource (LSF slots, simulation time and disk space) could be saved dramatically. The functional coverage could reach closure faster than ever before.

With the adoption of the proposed infrastructure, the functional coverage collection infrastructure could be easily reusable and scalable from IP to SoCs and the effort to maintain could be dramatically minimized. It has been adopted in more than six UVM IP standalones and SoC environments and we benefited from it.

ACKNOWLEDGEMENTS

We would like to thank for continued support to my wife (Liangliang Li) and AMD managers (Davis.Wan& Leo.zhang).

REFERENCES

[1] IEEE 1800-2009 SystemVerilog

[2] http://www.accellera.org/apps/org/workgroup/uvm/