SoC Verification Quantitative Closed-loop Management Based on Automation and DPI-C Library

Yongqi Sang, Bin Liu, Rui Zhao, Yifei Zhang

Modem Hardware Group

Intel Inc

Xi'an, Shaanxi, China

www.intel.com

ABSTRACT

With modern SoC size growing exponentially, how to implement a high quality SoC level verification is more challenging. There is no doubt that functional coverage is a key criterion for verification sign off, but normally due to the different strategy with module level, it is not easy to evaluate the functional coverage quantitatively on SoC level. This paper figured out a closed-loop management flow, with the DPI-C interface it can implement the functional and assertion coverage on SoC level conveniently, and with the HVP to map the verification objectives, test cases and coverage, it can get clear quantitative coverage status. The automation tool is also implemented. The flow can reduce verifiers' work load and give the verification manager more objective quantitative indicators to evaluate the SoC level verification quality.

key words： *SoC level, coverage, verification, HVP, quantitative, evaluation, closed-loop, automation*

Table of Contents

Table of Figures

Table of Tables

1. Introduction

With the rapid growth of chip scale, current SoC level verification itself is very complex and involves collaboration between multiple modules, which is one of the difficulties that make verification quality evaluation difficult. Another important reason is that due to hundreds of verifiers in the process of multi-billion-door IC system verification, the different coding style of test code is difficult to guarantee uniformity. Therefore, we have to further standardize various verification standards at SoC level, including the standardization of verification platform[1], the standardization of test instructions[1](no matter it is UVM or C test instruction set), the standardization of verification flow[2] and the standardization of quantitatively evaluate verification quality, which will be mainly described in this paper.

The main verification quality evaluation criteria include code coverage, functional coverage, assertion coverage, register coverage, and defect growth curves, etc. Among them, the defect growth curve can be recorded and visualized through the defect tracking tool[2]. The association between various coverages, test cases and verification objectives can be achieved by verification management tools. The verification planner and hierarchical verification plan(HVP) we are currently using plays such an important role. The solution we proposed can insert various coverages at SoC level easily and map those coverages result with verification objectives conveniently.

2. Background

As mentioned above, in traditional verification work, we mainly rely on criteria such as functional coverage and code coverage to evaluate the quality of verification. Code coverage can be collected automatically by the tools, and functional coverage need verifiers to perform verification plan decomposition and functional coverage modeling (mostly implemented by SV), it is no problem at module level, but when rise to large subsystem level and SoC level, the sharp increased recompilation time results in this method no longer applicable in these high-level test bench structures. In previous projects SoC level verification, actually we mainly relied on test cases pass rate to evaluate the quality of verification, but this method is not perfect enough, it cannot accurately check whether the test case really covers the verification objective. It is unreasonable to rely entirely on the test case review, this introduces too much extra time and cannot perform quantitative evaluation very well. Moreover, the module level verifiers often build the module level verification environment by themselves, and the SoC level verification environment is normally by centralized maintainance, these non-uniform standardized implementation will make our SoC level verification and review more difficultly.

In previous projects, we also used HVP to facilitate the management of verification objectives. The code coverage is normally analyzed after concentrated merge, less analysis corresponding to a single verification object. And on SoC level because there is no functional coverage and other coverage, the result of SoC level HVP corresponding to the verification objective is only the test cases, which did not fully exploit the potential of HVP. And for module level HVP editing, users have to manually fill in each functional coverage group name, which is relatively inefficient.

Let us summarize the problems mentioned above, we have the following challenges in current SoC level verification,

- Due to the cost of time, the insertion of SoC level functional coverage cannot follow the module level method.
- Need more reasonable evaluation criteria to increase the real value of review work.

- Need supplement SoC level HVP content, to make better use of its features.
- Need automation method to reduce repetitive work.

In response to above problems, we firstly developed one in-house verification IP called Hawkeye, which defined SoC level C test interface to meet both the compilation time demand and the SoC level test format[3], we have further optimized and added some more useful features, which considering the method of quantitatively evaluate verification quality. Secondly we also defined the HVP template, which includes everything related to verification quantification, such as verification objectives, test cases, functional coverage descriptions, assertion descriptions, and their mapping relations. And finally we also developed one automation tool, it uses the HVP as input. Along with the corresponding automation tool, we can automatically generate the functional and assertion coverage statements. These statements can be directly embedded into SoC level test to achieve coverage collection and ultimately mapping to verification objectives. So that the verification manager can focus on reviewing the functional coverage modeling and final quantitative analysis of each verification objective in the HVP.

3. Solution

As shown in Figure 1, our solution flow mainly includes the following steps,

1. The starting point is the HVP template, which defines several needed attributes for the Hawkeye interface, these attributes content need user input to enable the coverage model construction. These literal information is more easier to review at the verification beginning, and it is the only information which need user to supplement.
2. Based on this updated HVP, the automation script can generate relevant coverage statements and annotate the coverage groups to HVP metrics. These coverage statements can generate coverage groups dynamically when running test cases through DPI-C interface, which is implemented by Hawkeye.
3. With the generated statement in step 2, user can easily insert it into test cases, and run needed test case or regression to get the coverage database.
4. Then users can visually check the coverage map results of each covergroup defined in the HVP by Verdi/DVE coverage tool, in which the verification planner is natively integrated. And user can update test case if there are gap, similar as module level verification.

We can see that the flow uses HVP as the main line, the user defined verification objectives related criteria are in HVP, and the annotated results obtained from the test cases can also be quantitative evaluated with it, thus forming a closed-loop management flow. And the verification manager can focus on coverage modeling review and the final results review, verifiers can focus on coverage modeling and test case construction, which allows for more efficient use of precious project time and minimizes unnecessary redundancies of effort.

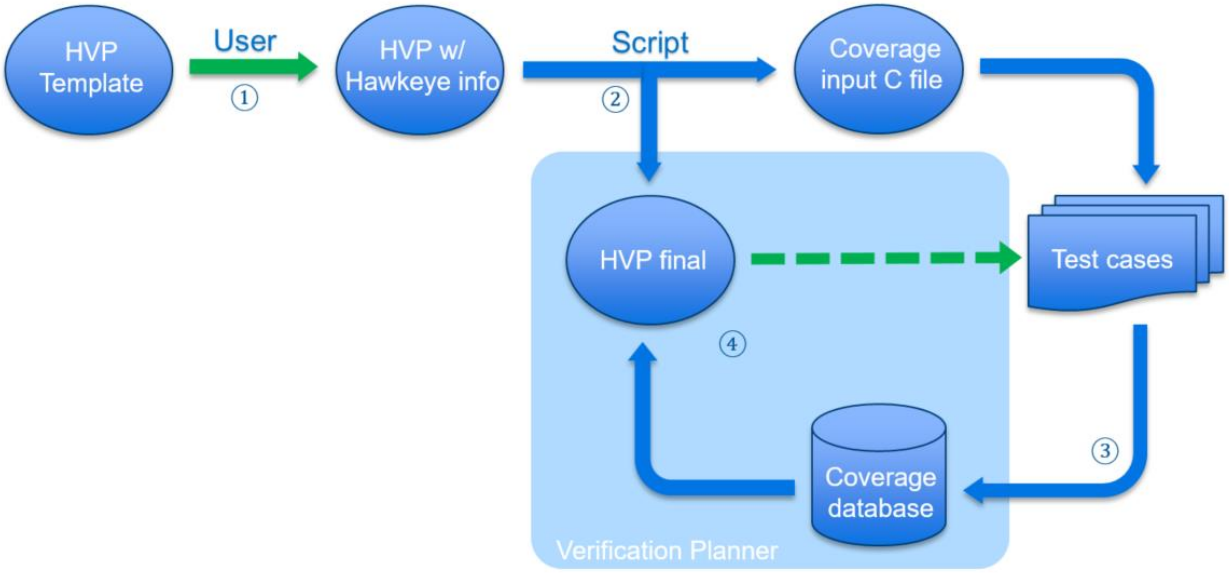

Figure 1. SoC verification quantitative closed-loop management flow

3.1 Hawkeye

Hawkeye is a full scale system monitor and evaluation solution[3], which include several sub-components and can easily be integrated. We are only using its functional coverage utility for this flow, and we also added the System Verilog Assertion(SVA) coverage modeling, which as a supplement to the functional coverage, can meet more complex practical demands from users.

3.1.1 Functional Coverage Utility

The functional coverage utility implemented basic coverage point definition methods by SV and also exported them as DPI-C APIs. C testers can employ those formal APIs to define coverage, and Hawkeye would dynamically create covergroups, which will generate the same coverage data format with module level functional coverage. Therefore, with the unified coverage APIs, it not only supplies the uniform coverage interface for better readability and reuse, and also closes the coverage database gap between module level and SoC level.

Since the information of final SV functional coverage modeling comes from the users' input content in the HVP, such as the coverage name, the signal under test, the clock, and the reset signal, etc., all of them are given in the form of string (always includes hierarchy information as well), so both the C and SV sides need receive the correct data type (as shown in Table 1). And in the SV side implementation, we have to perform the data type conversion and the binding between string and logic, so that the SV side can implement correctly when there are multiple modeling instance passed in. These details are mainly implemented by the VCS system task $hdl_xmr and the SV data type associative array. During the functions implementation procedure, it was also blocked by the insufficiency of SV language that it could not support the dynamic sized vector selection, and therefore, we have to copy quite a lot code for the methods to satisfy the possible different signal vector size.

Table 1. DPI C layer mapping data types[4]

SystemVerilog type	C type
byte	char
shortint	short int
int	int
longint	long long
real	double
shortreal	float
chandle	void *
string	const char *
bit	unsigned char
logic/reg	unsigned char

Figure 2 gives the defined coverage DPI-C APIs. Those APIs' names imply they would be taken to define single data coverage point, data range coverage point, data transition coverage point or cross data coverage point. In general, the exported APIs would satisfy the most need on SoC level. At last, SoC level functional coverage would be merged with module level coverage as a whole for functional coverage measurement, so it requires users to put unique coverage names while calling the APIs. Users can define the rules in advance, for instance, it is suggested to put the verification objective name + sampled signal name + sampled value to make the covergroup name the one and only.

```
extern void add_single_coverage(const char* sample_e, int
sample_width, const char* obj, int obj_width, int binval, const char*
name, int edge_mode = 2, int condition_val = -1);

extern void add_range_coverage(const char* sample_e, int
sample_width, const char* obj, int obj_width, int binval1, int
binval2, const char* name, int edge_mode = 2, int condition_val=-1);

extern void add_transition_coverage(const char* sample_e, int
sample_width, const char* obj, int obj_width, int binval1, int
binval2, const char* name, int edge_mode = 2 , int condition_val= -
1);

extern void add_cross_coverage(const char* sample_e, int
sample_width, const char* obj1, int obj1_width, int binval1, const
char* obj2, int obj2_width, int binval2, const char* name, int
edge_mode = 2, int condition_val = -1);
```

Figure 2. Functional coverage definition DPI-C APIs

3.1.2 SVA Coverage Utility

The SVA coverage utility implementation is somewhat similar with the functional coverage utility. We also have to implement the corresponding function to complete binding the users input string information to the internal logic signal, and instantiate the corresponding assertion and coverage.

We initially considered directly referencing the OVL[5] and Intel's built-in SVA checker libraries as these encapsulated SVA libraries are more friendly for users. But unfortunately, because the DPI-C interface requires SV task or function implementation, and the SVA concurrent assertion cannot be

called directly in the task or function, so we can only rewrite this part of the logic. The automatic task can perform the detection logic of most assertions[6], although this introduces us a lot of extra coding and testing effort.

Figure 3 shows one example, to make it easier to understand the complex implementations are replaced with pseudo code (uppercase). The forever monitor can get same effect as assertion. In the check part, the if-else can be used to replace the SVA implication. Since we cannot call the native SVA statement, so the corresponding assertion coverage cannot be seen, but we can dynamically create the function covergroup in the rewritten task when the detection condition is satisfied, from final coverage database it can also be used to observe whether these assertions are actually executed in the test.

```
task automatic ASSERT_NAME(string sva_name, string sig, int width,
string clk, string rst, int clk_edge, int rst_polarity);
  bit has_cov = 1;  //used for covergroup one time creation

  BIND string sig, clk, rst TO INTERNAL LOGIC SIG, CLK, RST

  fork  //fork join_none for multiple calling
    forever @(POSEDGE OR NEGEDGE IS DEPEND ON INPUT clk_edge))
    begin
      if(RST DEASSERT VALUE IS DEPEND ON INPUT rst_polarity)
      begin  //enable check when reset released
        if(width == 1)  //sig is single bit
          SINGLE BIT SIG CHECK, PRINT ERROR WHEN VIOLATION
        else  ////sig is vector
        begin
          for(int i=0; i<width; i++)
            VECTOR SIG BIT CHECK, PRINT ERROR WHEN VIOLATION
        end        //enable coverage when assert active
        if(has_cov == 1)
        begin
          CREATE FUNCTION COVERGROUP NAMED sva_name
          ENABLE COVERGROUP SAMPLE
          has_cov = 0;  //for each instance only create one cov
        end
      end
    end
  join_none  //fork join_none for multiple calling
endtask
```

Figure 3. Task implemented SVA example with pseudo code

Due to time insufficient, we didn't rewrite all the library assertions (more than 100). Instead, we just implemented some of the most common and most wanted assertions from users according to the project requirements. These will continue to improve according to the feedback from the later projects.

3.2 HVP

3.2.1 HVP Basis

HVP is a comprehensive model that allows user to hierarchically describe a verification plan[7]. As figure 4 shows one example, it contains feature declarations, attribute/annotation declarations, goals and metrics, etc. Attributes are named values specified in the plan, whereas metrics are named values annotated from HVP data files. Metrics can be coverage information extracted from merged simulation runs. We can use measure specifications to declare which metrics to annotate a feature from the verification database.

```
#plan identifier
plan PLAN_NAME;
  #metric-declaration
  metric enum {PASSED, UNKNOWN, FAILED} test_status;
    aggregator = sum;
    goal = ((test_status.FAILED + test_status.UNKNOWN)==0);
  endmetric
  #attribute and annotation declaration
  attribute string prio = "";
  ……
  annotation string hky_cg_name = "";
  ……
  #feature declaration
  feature Function;
    #sub feature declaration
    feature VO_NAME1;
      owner = "user1";
      description = "……";
      #measure declaration
      measure test_status;
        source = "TESTCASE_PATH1/TESTCASE1";
      endmeasure
      feature COV_NAME1;
        hky_cg_name = "cov_name1";
        ……
        measure Group grp;
          source = "group instance: hky_pkg::cov_name1";
        endmeasure
      endfeature
      feature COV_NAME2;
        ……
      endfeature
      ……
    endfeature
    ……
  endfeature
  #subplan declaration
  subplan subplan1;
endplan
```

Figure 4. HVP basic syntax example

When HVP is loaded in Verdi/DVE coverate tool, it is shown as a static two dimentional table, the definitions of attributes, annotations and metrics can be seen as its columns, the plan and feature trees, which are the basic building blocks, can be seen as its rows.

3.2.2 HVP Template

As feature part is what user can edit through Verdi/DVE coverage GUI, so we defined such HVP template, the verification objectives are defined as its root feature, the function/SVA coverage and test cases status, etc. are its sub hierarchical feature or metrics measure. As the built-in metrics have included assertion and functional coverage, etc., we can use it directly without modification. And then following the Hawkeye DPI-C APIs interface parameters, we abstracted it and defined them as new attributes/annotations, from users' point of view, they only need supplement these attributes value based on the functional coverage modeling intention.

As figure 5 shows, when users need add one new functional coverage in the template, they just need add one new feature by click the 'create a new feature' button (shown in circle 1), then one new feature with default name 'Feature_x' will be added (shown in circle 2), users can ignore the default name because with the automation tool it will finally be annanoted by the value of attribute 'hky_cg_name', user need update the Hawkeye related parameters value in the right side list, where all the Hawkeye needed parameters are shown. (shown in circle 3).

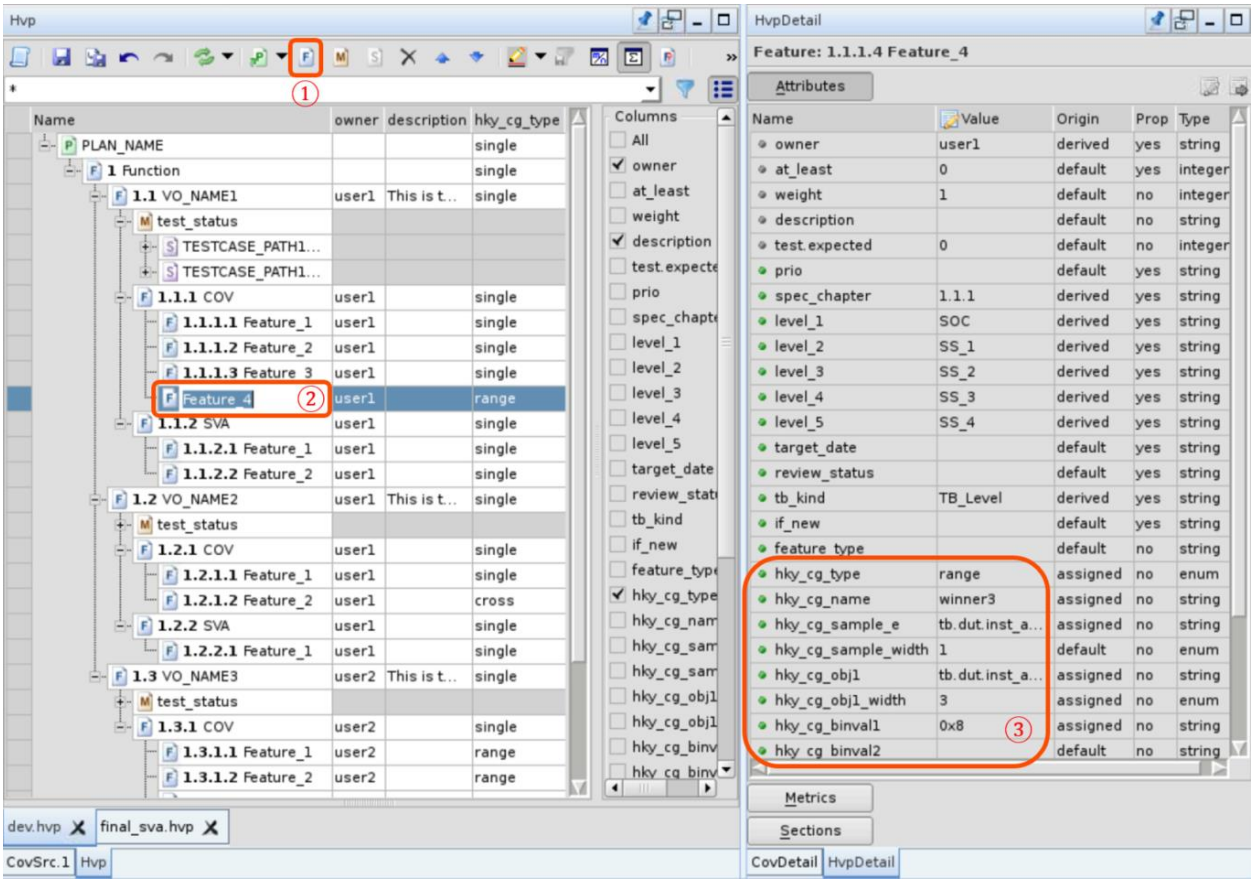

Figure 5. HVP template update example

As HVP is finally shown as a static two dimentional table, we cannot dynamically switch its display like the excel pivot table, and the number of different Hawkeye interface function parameters are not exactly the same, which introduces some inconvenience to our application. We have to define the attributes according to the function with the most parameters. Users may only need a part of parameters when using some functions, the extra parameters display may introduce a bit trouble to the primary users. Although we have detail document to avoid such issues, it would be better to dynamically gray out those items not required, maybe Synopsys can consider adding similar features to future updates.

3.2.3 Final Mapping Result Show

Figure 6 shows the HVP mapping result example in Verdi coverage. After there added the function/SVA coverage information and related metrics measure in the HVP (shown in circles), we get a final version HVP. And after C tests with Hawkeye coverage definition run done, we get the coverage database with those Hawkeye defined covergroup inside, they can be mapped intuitively. We can see these coverage groups instantiated in the coverage list on the left, and the coverage can be quantified from the middle HVP mapping results. It can be seen that the mapping relations between verification objectives, test cases and coverages are clear. Based on these mapping relations, we even can enable only some of the specific coverage related with specific test case. This may be useful on SoC level, and it can be controlled in the coverage C test file by 'ifdef' easily.

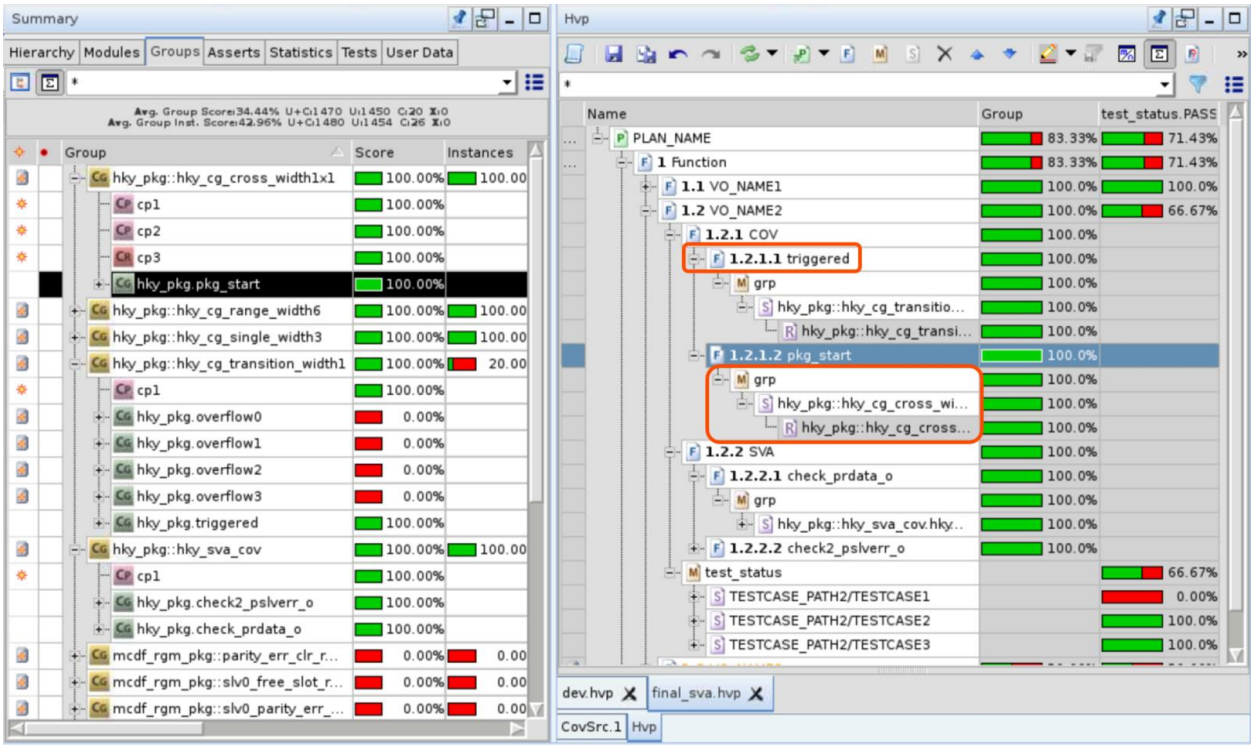

Figure 6. HVP map example in Verdi coverage

3.3 Toolkit Delivery

3.3.1 Requirement

From above introduction, we can see there are a lot of repetitive work actually in this flow, such as define the Hawkeye parameters in HVP, update the coverage feature name, add covergroup name in the HVP measure (Figure 5 to Figure 6), write the C test, etc. These repetitive works are boring and easy to mistake, so we developed the automation script to implement it. It can be automated because all these info use same value defined in Hawkeye parameters, which from users' original input. These common information are the essencial of the automation.

And in the work of integrating Hawkeye into different modules, there are also some repetitive work. It can be hiden for users who are not familiar with Hawkeye, So the automation tools also implemented some test bench Hawkeye insertion and makefile update functions, etc.

So as shown in Figure 7, our final toolkit delivery includes the HVP template, the automation scripts, and the Hawkeye integration needed SV files and makefiles.

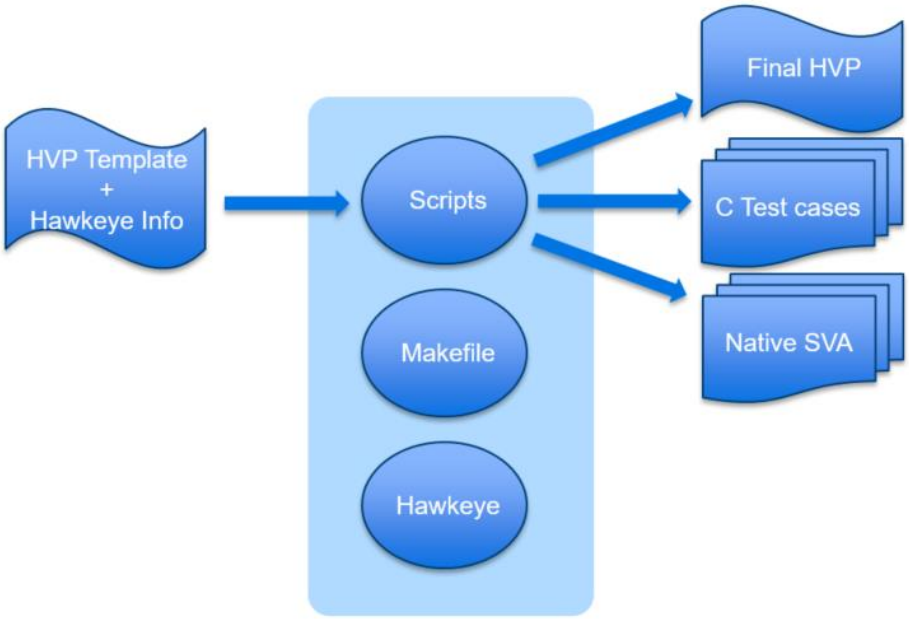

Figure 7. Toolkit delivery

3.3.2 Automation Script

The automation tool is batch mode scripts developed by Python. As figure 7 shows, the input is the HVP file with Hawkeye parameters which added by user, the output are,

- The final version HVP that updates the coverage feature name and metrics measure, etc., which can be used for final mapping directly.
- The C test files with Hawkeye function calling, these C test files can be compiled and run through the DPI-C interface during simulation, whether it's direct or SV/UVM testbench.
- The calling of native SVA (OVL and Intel's built-in SVA checker libraries), which we added later. It's more considering the demand from module level verifiers and design engineers, so

they can either embed the native SVA statements into RTL or bind with their test bench easily, and can correspong to the verification objectives.

Therefore SoC level verifiers can focus on the verification objectives and coverage modeling and of course the test case development (which shown in figure1 step 1 with green color), all other rest works (which shown in figure1 step 2-4 with blue color) can be automated. The script also have the necessary features such as HVP check to ensure that the entire closed-loop operation can be carried out correctly.

3.3.3 Makefile and Others

Other deliveries include the Hawkeye complete SV packages and the Makefile which used to integrate it. Users can integrate Hawkeye into their own test bench by merge the Makefile content to their compilation environment. Since we have considered various common issues for users, the integration is actually very easy.

4. Summary

In this paper we introduced a SoC level verification quantitative closed-loop management flow, let us correspond the solution with the problems in the background section,

- We developed Hawkeye to implement SoC level functional and SVA coverage modeling through DPI-C, it doesn't need modify the test bench, just need a little compilation time that is almost negligible.
- We defined HVP template and format, which unified the SoC level verification quality evaluation criteria. With verification objectives verification managers can focus on the literal checking of functional coverage modeling and final mapping result, both the load and efficiency of review work are improved.
- We delivered toolkit with automated sript, which increased the verifiers' efficiency and reduced the errors possibility. Verifiers can focus on the coverage modeling literal information and test case construction without worry about lots of low level redundant coding. Obviously, this is also beneficial to the improvement of the entire project efficiency.

Of course, because we limit the coverage modeling only through Hawkeye inteface, we somewhat lost flexibility of functional coverage contrast to module level. And it also brings a little extra learning cost to the verifiers, but compare with our benefit, it is worthy.

The scheme in this pater has been proven to be usable, and will be used in the following big project.Before the SNUG conference in June, we may supplement more practical applications into the solution and summary if there are.

5. References

[1] Bin Liu, "Unified Verification Framework Automation and Test Standardization with UVM", SNUG 2017
[2] Bin Liu, "A Walking Guide to SoC Verification: The panorama of Verification from System to UVM", Publishing House of Electronices Industry, 2018
[3] Bin Liu, "A Full-scale System Monitor and Evaluation Solution for SoC Verificaiton", DVCon China, April 2018
[4] IEEE Std 1800™-2017, IEEE Standard for SystemVerilog

[5] https://accellera.org/downloads/standards/ovl
[6] https://verificationacademy.com/verification-horizons/march-2018-volume-14-issue-1/sva-alternative-for-complex-assertions
[7] $VCS_HOME/doc/UserGuide/pdf/v_planner.pdf