Hitchhikers Guide to Structural and Functional Coverage Merging and Mapping with VCS®, SystemVerilog and VMM

Jacob Andersen
Benny Andersson
Peter Jensen

SyoSil ApS
Systems on Silicon

Himmelev bugade 53, 4000 Roskilde, Denmark
{jacob, benny, peter}@syosil.com

www.syosil.com

ABSTRACT

This paper provides the reader with guidelines and advice on how to successfully deploy structural and functional coverage with VCS® 2009.12. The paper takes its offset in experiences gained from real industrial projects, and provides general solutions for several merging problems.

In general, coverage is demystified through a down to Earth view on the topic. Three specific scenarios of coverage merging and mapping are presented. Each scenario addresses common misunderstandings related to coverage. Especially, when non-configurable and configurable (through parameters etc.) RTL blocks are verified in multiple testbenches and when they are instantiated multiple times in the different test benches.

Table of Contents

Table of Figures

Table of Tables

1 Definitions

CRV: Constrained Random Verification
ASIC: Application Specific Integrated Circuit
ABV: Assertion Based Verification
DUT: Device Under Test
VCS: Verilog/SystemVerilog simulator from Synopsys
URG: Unified Report Generator (Part of the VCS® tool chain)
DVE: Discovery Visualization Environment (Part of the VCS® tool chain)
LSF: Load Sharing Facility
SGE: Sun Grid Engine
VMM: Verification Methodology Manual (For SystemVerilog)
HVP: Hierarchical Verification Plan

2 Introduction

This paper provides the reader with guidelines and advice on how to successfully deploy structural-, functional- and assertion coverage with VCS® 2009.12. The paper takes its offset in experiences gained from real industrial projects, and provides general solutions for several coverage related problems. Common misunderstandings related to coverage merging are addressed, especially when RTL blocks are verified in multiple test benches and they are instantiated multiple times in the different test benches.

The paper presents a down to Earth view on coverage merging and demystifies the topic, while presenting specific scenarios of coverage merging and mapping. The scenarios are presented through some simple examples, which resemble the blocks used in the real industrial RTL design.

Experiences gained from verification of a real design shows that just reading the VCS® documentation about coverage will simply not get the job done when a project grows to a substantial size.

Basically, the VCS® documentation provides a classic feature presentation, typical for software user guides. For each feature, it provides almost a complete set of information, related solely to that feature, but it lacks information, when the features are used together. Thus, it delivers answers to many aspects of coverage, but the information is presented per feature. What if you have a block, which needs all of these coverage features in the same project?

Hence, the verification engineer, who is responsible for the coverage is put in a tough situation. He/she face several coverage tasks, which should be carried out by utilizing several VCS® coverage features (potentially using multiple of those features at the same time), but the VCS® documentation does not provide this kind of information. This is actually quite a big job, when one is inexperienced or when the verification task is big, e.g. for a whole ASIC with several testbenches etc.

The overall goal of this paper is to provide the perpendicular, task oriented view on coverage which aids the verification engineer. In that aspect, this paper is a hitchhiker's guide to coverage with VCS.

Section number 3 defines a general set of coverage tasks, which a verification engineer must carry out on large ASIC projects. The section takes its offset by presenting an abstraction of a real verification project. The ASIC in the real project was fairly big, with approximately 700 unique RTL modules and 6-7 VMM/Vera testbenches. The daily regression produced about 10GB of data to be analyzed each day (Coverage reports, log files etc.)

The general tasks are presented based upon this real verification project and finally they are mapped to VCS® coverage features. This paper is example based. Hence, section 4 presents the example designs and testbenches used throughout this paper. Each of the following sections deals with one of the general coverage tasks by presenting a short description of the task, followed by a list of observed problems, derived from the real project. Each section ends with some recommendations to how the problems could be tackled.

The scope of this paper is limited to structural coverage only for simplicity. Thus, all examples are only shown for structural coverage if not stated explicitly. The recommendations given in section 5 to 9 still applies for functional and assertion coverage. Moreover, the paper should be viewed as an addendum to the VCS® documentation and it does not try to replace it in any way.

Furthermore, the reader is expected to be familiar with the VCS® tool chain. The different binaries and how to enable coverage is out of the scope for this paper.

3 General Coverage Tasks

Today, in a typical real ASIC verification project, coverage plays a vital role, due to the coverage based verification methodologies, which are deployed. This calls for definition, collection and reporting of coverage. This paper deals with the collection and reporting of coverage. Over the last couple of years these two tasks has grown tremendously, due to many topics:

- Coverage based verification methodologies (CRV (VMM), ABV)
- Improved implementation/verification languages (SystemVerilog)
- More complex designs, which utilize reuse
- Very large daily/weekly regressions, producing massive amount of coverage data from multiple testbenches, testing the same blocks in different ways.

For example, take Figure 1. It shows an abstraction of a part of a real verification project. Block A has its own testbench, in which it can be instantiated either as A0 or A1. This is controlled through a `` `define `` . Additionally, block B has its own testbench and not all of the functionality implemented by block B is used in block C, so coverage exclusion was introduced. Together block A and block B are instantiated in Block C, which then as well has its own testbench. Block A is instantiated multiple times, both as A0 and A1. Finally, block C is instantiated inside the top level testbench along with some other blocks. Thus, block A and B are present in three testbenches from which coverage should be collected via merging across tests, testbenches etc., so reports that answered well known manager questions like: “What is the overall coverage for block A?” could be generated.

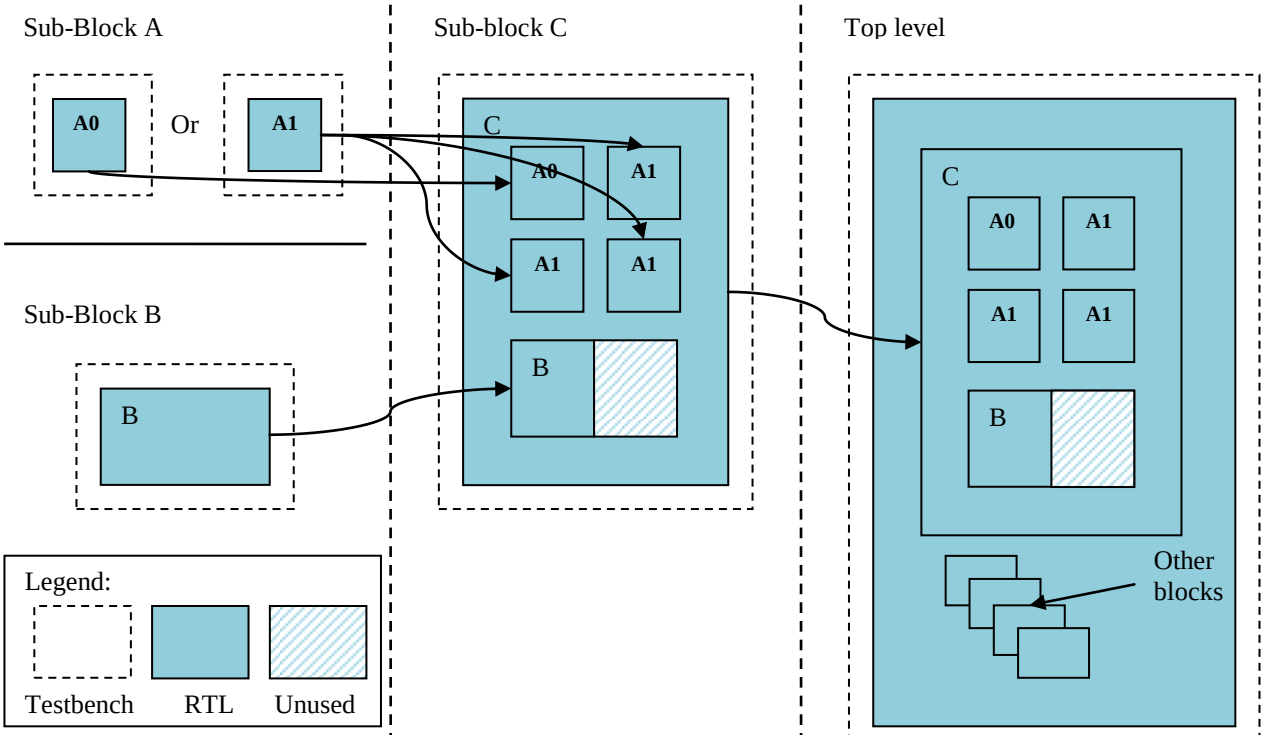


Figure 1: Abstraction of a part of a real verification project

A lot of experience was gained by working with the verification setup depicted in Figure 1. The project revealed that a lot of VCS® coverage features need to be utilized in an industrial size ASIC project, when trying to generate coverage reports, where multiple blocks can be parameterized, instantiated multiple times in multiple testbenches etc. A set of general tasks/features was derived based on these experiences. Figure 2 shows these general tasks/features, which was identified during the verification work of block A, B, C etc.

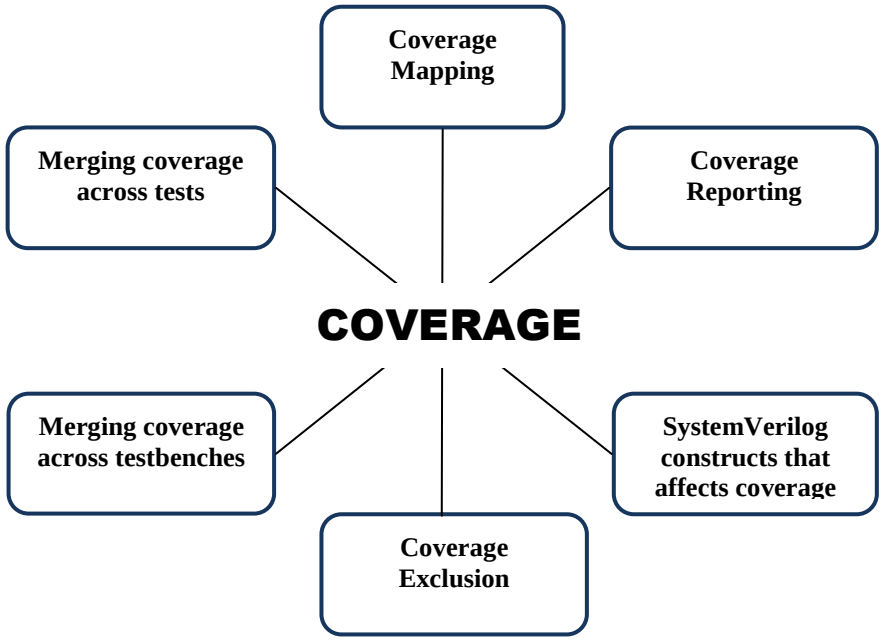


Figure 2: General coverage tasks and features

The following sections cover these general tasks/features and provide guidelines, recommendations and advice on how they are solved by utilizing VCS® coverage features.

3.1. VCS® Coverage Features Mapped to General Coverage Tasks

Figure 2 in the previous section defined a set of general coverage tasks/features, which were obtained by working with a real industrial ASIC verification project, but how does this map to VCS® coverage features?

In general VCS® provides coverage mechanisms for structural, functional and assertion coverage for the SystemVerilog HDL. The VCS® coverage features matrix defined in Table 1 gives an overview of how the general coverage related tasks/features relate to the VCS® tool chain.

Table 1: Coverage feature matrix

	Coverage Type		
General Task/Feature	**Structural**	**Functional**	**Assertion**
Merging across tests	VCS/URG	VCS/URG	VCS/URG
Coverage merging across test benches	URG	URG	URG
Coverage mapping	URG	Not Supported	Not Applicable
SV constructs which affects coverage	VCS	VCS	VCS
Coverage Exclusion	URG/DVE	URG/DVE	URG/DVE
Coverage Reporting	URG/DVE/VMM Planner	URG/DVE/VMM Planner	URG/DVE/VMM Planner

The following sections will provide example based guide lines and advice on the general tasks/features related to VCS® features in the coverage feature matrix and to the combination of those.

4 Example RTL Designs and Testbenches

The table below defines four different RTL designs, which are going to be used throughout this paper:

- RTLA: This is a standard RTL module and input mapped to an output through some very simple logic. It contains no `` `defines `` or parameters.
- RTLB: Same as RTLA, but with simple `` `define ``
- RTLC: As RTLB, but the `` `defines `` enables two completely different code sections
- RTLD: Standard RTL module with a parameter for controlling the width of the input and output vectors.

```
RTLA:
module modbase(
  input  bit         clk,
  input  bit         rst,
  input  logic [2:0] inv,
  output logic [2:0] outv);
```

```
RTLB:
module modbase_def(
  input  bit         clk,
  input  bit         rst,
  input  logic [2:0] inv,
  output logic [2:0] outv);
```

```
  always_ff @(posedge clk) begin
    if(rst) begin
      outv <= 3'b000;
    end else begin
      unique case(inv[2:0])
      3'b000 : outv[2:0] <= 3'b000;
      3'b001 : outv[2:0] <= 3'b001;
      3'b010 : outv[2:0] <= 3'b010;
      3'b011 : outv[2:0] <= 3'b011;
      3'b100 : outv[2:0] <= 3'b100;
      3'b101 : outv[2:0] <= 3'b101;
      3'b110 : outv[2:0] <= 3'b110;
      3'b111 : outv[2:0] <= 3'b111;
      endcase // case (inv[2:0])
    end
  end // always
endmodule // modbase
```

```
RTLC:
module modbase_def1(
  input  bit         clk,
  input  bit         rst,
  input  logic [2:0] inv,
  output logic [2:0] outv);

  always_ff @(posedge clk) begin
     if(rst) begin
      outv <= 3'b000;
     end else begin
`ifdef MY_DEFINE
      unique case(inv[2:0])
      3'b000 : outv[2:0] <= 3'b000;
      3'b001 : outv[2:0] <= 3'b001;
      3'b010 : outv[2:0] <= 3'b010;
      3'b011 : outv[2:0] <= 3'b011;
      3'b100 : outv[2:0] <= 3'b100;
      3'b101 : outv[2:0] <= 3'b101;
      3'b110 : outv[2:0] <= 3'b110;
      3'b111 : outv[2:0] <= 3'b111;
      endcase
`else
      if(inv[2:0] === 3'b000)
       outv[2:0] <= 3'b111;
      else
```

```
  always_ff @(posedge clk) begin
     if(rst) begin
      outv <= 3'b000;
     end else begin
`ifdef MY_DEFINE
      unique case(inv[2:0])
      3'b000 : outv[2:0] <= 3'b000;
      3'b001 : outv[2:0] <= 3'b001;
      3'b010 : outv[2:0] <= 3'b010;
      3'b011 : outv[2:0] <= 3'b011;
      3'b100 : outv[2:0] <= 3'b100;
      3'b101 : outv[2:0] <= 3'b101;
      3'b110 : outv[2:0] <= 3'b110;
      3'b111 : outv[2:0] <= 3'b111;
      endcase
`else
      unique case(inv[2:0])
      3'b000 : outv[2:0] <= 3'b111;
      3'b001 : outv[2:0] <= 3'b000;
      3'b010 : outv[2:0] <= 3'b001;
      3'b011 : outv[2:0] <= 3'b010;
      3'b100 : outv[2:0] <= 3'b011;
      3'b101 : outv[2:0] <= 3'b100;
      3'b110 : outv[2:0] <= 3'b101;
      3'b111 : outv[2:0] <= 3'b110;
      endcase
`endif
     end
  end // always
endmodule // modbase_def
```

```
RTLD:
module modbase_param #(WIDTH = 5) (
  input  bit         clk,
  input  bit         rst,
  input  logic [WIDTH-1:0] inv,
  output logic [WIDTH-1:0] outv);

  always_ff @(posedge clk) begin
    if(rst) begin
      outv <= 'b0;
    end else begin
      if (inv < 6)
        outv <= inv;
      else if (inv < 13)
        outv <= inv + 1;
      else
        outv <= inv + 2;
    end
  end // always
endmodule // modbase_param
```

```
       outv[2:0] <= 3'b001;
`endif
     end
  end // always
endmodule // modbase_def1
```

Several small testbenches are created to drive stimuli into these modules. The following table lists two typical testbenches used in this paper. Not all testbenches are listed here for simplicity.

Testbench for RTLA:

```
module tbbase;
  bit  clk;
  bit  rst;
  logic [2:0] inv;
  logic [2:0] outv;

  modbase modbase_i(.clk(clk),
   .rst(rst), .inv(inv), .outv(outv));

  initial begin
     clk = 1'b0;

     forever begin
       #50;
       clk = ~clk;
     end
  end

  initial begin
    logic [2:0] inv_val;
    int status =
      $value$plusargs("inv_val=%b",
                      inv_val);

    $display("%t: inv_val = %b(%0d)",
              $time, inv_val, inv_val);

    rst <= 1'b1;
    repeat (5) @(posedge clk);
    rst <= 1'b0;

    inv[2:0] = inv_val[2:0];
    repeat (5) @(posedge clk);

    $display("%t: outv = %b", $time,
             outv);

    $finish;
 end
endmodule
```

Testbench for RTLD:

```
module tbbase_param_3;
   bit  clk;
   bit  rst;
   logic [2:0] inv;
   logic [2:0] outv;

   modbase_param #(.WIDTH(3)) modbase_i(
     .clk(clk), .inv(inv), .outv(outv));

   initial begin
      clk = 1'b0;

      forever begin
        #50;
        clk = ~clk;
      end
   end

   initial begin
      logic [2:0] inv_val;
      int status =
        $value$plusargs("inv_val=%b",
                        inv_val);

      $display("%t: inv_val = %b(%0d)",
               $time, inv_val, inv_val);

      rst <= 1'b1;
      repeat (5) @(posedge clk);
      rst <= 1'b0;

      inv <= inv_val;
      repeat (5) @(posedge clk);

      $display("%t: outv = %b", $time,
               outv);

      $finish;
    end
endmodule
```

5 Coverage Merging across Tests

Description:
Coverage merging across tests is the fundamental operation with respect to coverage. Hence, it would be used with any testbench in Figure 1 to generate coverage reports. The task of merging across tests is well covered by the VCS® documentation as long as the design contains no features described in section 7 to 9.

The design and testbench is compiled once into a single executable with the preferred coverage enabled. Then, the design can be simulated a number of times to obtain coverage. Finally, URG is used to merge the coverage. Thus, the recommended flow is:

Compile : `vcs …`
Simulate : `./simv …`
Report : `urg -dir simv.cm`

Then URG will generate a directory named `urgReport`, which contains the coverage report. Consult the VCS® documentation for more information on this topic.

Problem:
We experienced two problems in the project related to the fundamental merging of coverage:

1. Initially very low overall coverage was obtained.
2. It was hard for verification engineers to locate specific tests coverage contribution

Recommendation:
Even though the problems described above may seem minor, they still need to be address among all the other coverage related issues, if a well defined coverage methodology should be followed in a real life project. Problem (1) above occurred due to the fact that by default VCS® gathers coverage on everything; the design testbench etc. Typically, there is no point in obtaining coverage on the testbench, so it is recommended to exclude everything but the RTL. This is done by using the `-cm_hier` option for VCS. The second problem, is caused by the fact that basic flows used for simulation typically does not name each test specifically. Thus, it is recommended to add the `-cm_name` option for structural coverage, when simulating multiple times with the same `simv`. This will name each test run in the `simv.cm` directory. The recommended flow is now:

Compile : `vcs…`
Simulate 1: `./simv -cm_hier … -cm_name test1 …`
…
Simulate N: `./simv -cm_hier … -cm_name testN …`
Report : `urg -dir simv.cm`

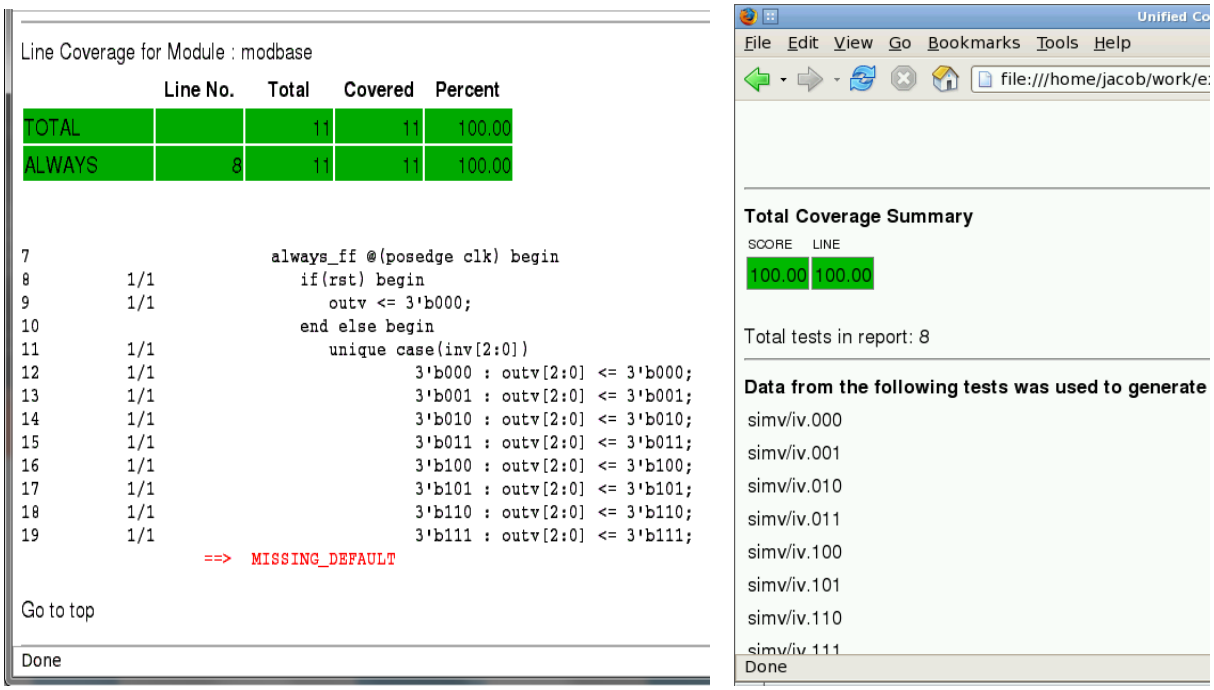


Figure 3: Snippet of the generated coverage report

Figure 3 shows an example of the generated coverage report when running the testbench for RTLA with all combinations of `inv_val`. Furthermore, Figure 3 also shows how each test was named: `iv.000-iv.111`.

6 Coverage Merging across Testbenches

Description:
The task of merging across testbenches is similar to merging across tests. It covers the situation, where the same `simv` (testbench) is compiled multiple times and run in different locations. Thus, the testbench is simulated like described in section 5 and the flow is the same with a single exception. The merged coverage report is generated by changing the report command to:

Report : `urg -dir tb0/simv.cm tb1/simv.cm … tbN/simv.cm`

Where `tb0` to `tbN` are directories containing the compiled simv's for each testbench. Again, it is presumed that the design contains no features described in section 7 to 9. This is the typical scenario, when for instance multiple simulations are executed in parallel via SGE or LSF. The task of merging across testbenches is also well covered by the VCS® documentation.

Problem:
The problems are the same as in the previous section.

Recommendation:
The recommendations are the same as in the previous section.

7 Coverage Mapping

Description:
Coverage mapping is used, when coverage for a certain module needs to be obtained from different testbenches. Typically, this is needed, when an RTL block for instance is not 100% verified in its own block testbench, but parts are perhaps verified in the top level testbench.

When looking at any particular block in a design, one can easily see that there are 4 general instantiation scenarios for any given block:

- **Scenario 1:** One RTL block is only instantiated once in a single testbench (1:1). This is the normal case. No mapping is need. This is covered in section 5.
- **Scenario 2:** One RTL block is only instantiated once in multiple testbenches (1:M). Mapping should be used here, since each of the testbenches might not be equivalent. Mapping will provide coverage numbers for a specific block. This resembles the situation depicted for sub-block B and C in Figure 1.
- **Scenario 3**: One RTL block is instantiated multiple times in a single testbench (N:1).This is also a normal case, since the same block is instantiated in the same testbench. Thus, no mapping is needed to get the merged coverage. Sub block A in sub block C in Figure 1 depicts this scenario.
- **Scenario 4**: One RTL block is instantiated multiple times in multiple testbenches (N:M). This definitely calls for coverage mapping, since it is a merge of scenario 2 and 3. Sub block A in Figure 1 has its own testbench and it is instantiated multiple times in sub block C etc.

Problem:
The URG documentation defines two terms mapped design and base design, but it is a bit vague on which one should be put first on the command line after `urg -dir ...` etc. Thus, in which order should the coverage directories be placed in order to get for instance total coverage for a module. This lead to the fundamental question, how is module and instance coverage accumulated when mapping downwards (from a testbench with multiple instances to a testbench with fewer instances) and upwards (the other way).

Furthermore, the URG documentation states that there must only be one instance in the base design of the mapped design. Hence, only downwards mapping is supported. The problem is that one could easily make an error and put the coverage database in the wrong order. Thus, upwards merging will be carried out.

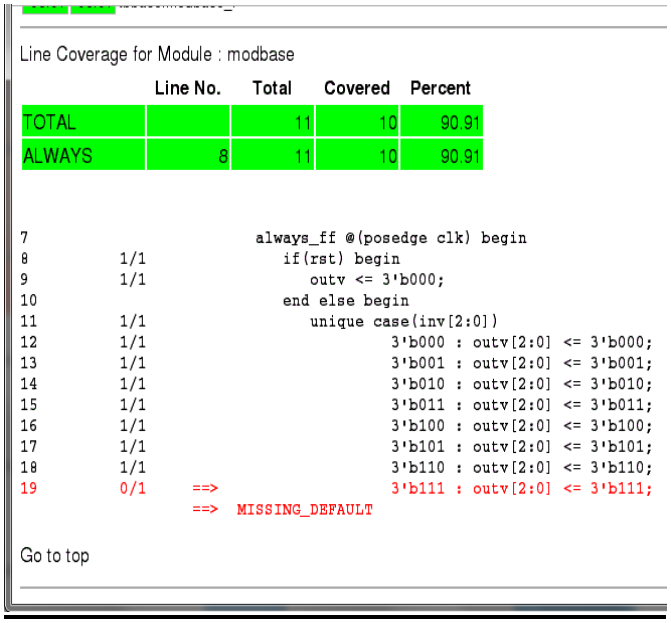
To look at what happens when doing upwards merge, scenario 4 is analyzed. Design RTLA is instantiated in 3 different testbenches called TB1, TB2 and TB3, with 1, 2 and 3 instances of RTLA. Each testbench is then simulated with a specific set of input vectors, so the results can easily be analyzed:

- **TB1:** Simulated 2 times with `inv = 3'b000` and `3'b001`, providing line coverage for line 12 and 13 in RTLA.
- **TB2:** Simulated 1 time with `inv = 3'b010` (1. instance) and `inv = 3'b011` (2nd instance) providing line coverage for line 14 and 15 in RTLA.
- **TB3**: Simulated 1 time with `inv = 3'b100` (1st instance), `inv2 = 3'b101` (2nd instance) and `inv3= 3'b110` (3rd instance), providing line coverage for line 16, 17 and 18 in RTLA.

Now, the coverage databases from each of the four runs are merged in all permutations with URG:

```
urg –map modbase –dir tb1/simv.cm tb2/simv.cm tb3/simv.cm
urg –map modbase –dir tb1/simv.cm tb3/simv.cm tb2/simv.cm
urg –map modbase –dir tb2/simv.cm tb1/simv.cm tb3/simv.cm
urg –map modbase –dir tb2/simv.cm tb3/simv.cm tb1/simv.cm
urg –map modbase –dir tb3/simv.cm tb1/simv.cm tb2/simv.cm
urg –map modbase –dir tb3/simv.cm tb2/simv.cm tb1/simv.cm
```

Where, the first URG command is a complete downwards merge and the last one is a complete upwards merge. The URG commands in between are something in between a downwards and upwards merge. Figure 4 shows the module coverage for the `modbase` module for the downwards merge (the first URG command above). Note that module and instance coverage is the same, since the module is only instantiated once in the base module.



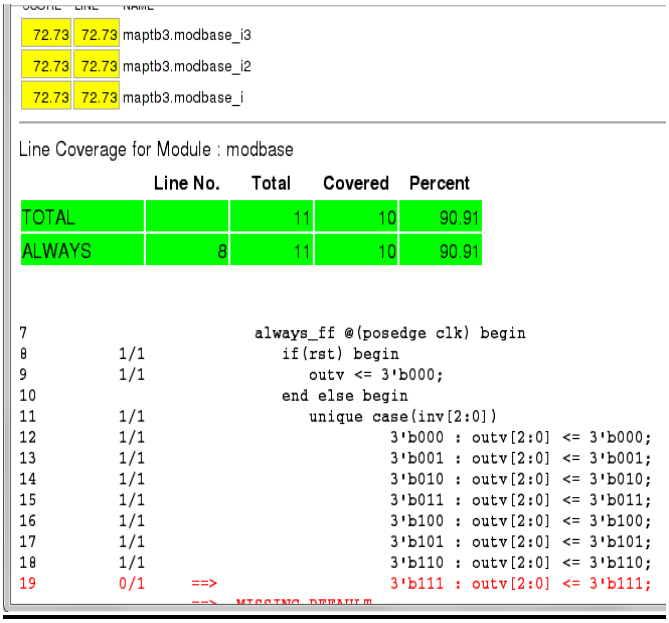
Line Coverage for Module : modbase

	Line No.	Total	Covered	Percent
TOTAL		11	10	90.91
ALWAYS	8	11	10	90.91

```
7                       always_ff @(posedge clk) begin
8          1/1             if(rst) begin
9          1/1                outv <= 3'b000;
10                         end else begin
11         1/1                unique case(inv[2:0])
12         1/1                        3'b000 : outv[2:0] <= 3'b000;
13         1/1                        3'b001 : outv[2:0] <= 3'b001;
14         1/1                        3'b010 : outv[2:0] <= 3'b010;
15         1/1                        3'b011 : outv[2:0] <= 3'b011;
16         1/1                        3'b100 : outv[2:0] <= 3'b100;
17         1/1                        3'b101 : outv[2:0] <= 3'b101;
18         1/1                        3'b110 : outv[2:0] <= 3'b110;
19         0/1     ==>                3'b111 : outv[2:0] <= 3'b111;
                   ==>  MISSING_DEFAULT
```

Go to top

Figure 4: Downwards module coverage



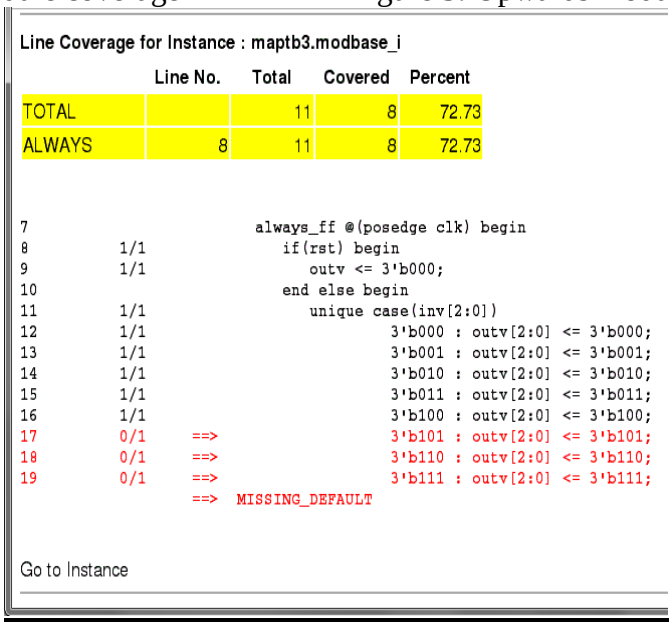
72.73	72.73	maptb3.modbase_i3
72.73	72.73	maptb3.modbase_i2
72.73	72.73	maptb3.modbase_i

Line Coverage for Module : modbase

	Line No.	Total	Covered	Percent
TOTAL		11	10	90.91
ALWAYS	8	11	10	90.91

```
7                       always_ff @(posedge clk) begin
8          1/1             if(rst) begin
9          1/1                outv <= 3'b000;
10                         end else begin
11         1/1                unique case(inv[2:0])
12         1/1                        3'b000 : outv[2:0] <= 3'b000;
13         1/1                        3'b001 : outv[2:0] <= 3'b001;
14         1/1                        3'b010 : outv[2:0] <= 3'b010;
15         1/1                        3'b011 : outv[2:0] <= 3'b011;
16         1/1                        3'b100 : outv[2:0] <= 3'b100;
17         1/1                        3'b101 : outv[2:0] <= 3'b101;
18         1/1                        3'b110 : outv[2:0] <= 3'b110;
19         0/1     ==>                3'b111 : outv[2:0] <= 3'b111;
```

Figure 5: Upwards module coverage



Line Coverage for Instance : maptb3.modbase_i

	Line No.	Total	Covered	Percent
TOTAL		11	8	72.73
ALWAYS	8	11	8	72.73

```
7                       always_ff @(posedge clk) begin
8          1/1             if(rst) begin
9          1/1                outv <= 3'b000;
10                         end else begin
11         1/1                unique case(inv[2:0])
12         1/1                        3'b000 : outv[2:0] <= 3'b000;
13         1/1                        3'b001 : outv[2:0] <= 3'b001;
14         1/1                        3'b010 : outv[2:0] <= 3'b010;
15         1/1                        3'b011 : outv[2:0] <= 3'b011;
16         1/1                        3'b100 : outv[2:0] <= 3'b100;
17         0/1     ==>                3'b101 : outv[2:0] <= 3'b101;
18         0/1     ==>                3'b110 : outv[2:0] <= 3'b110;
19         0/1     ==>                3'b111 : outv[2:0] <= 3'b111;
                   ==>  MISSING_DEFAULT
```

Go to Instance

Figure 6: Upwards instance coverage for instance `modbase_i`

Figure 5 shows the module coverage for the modbase module for the upwards merge (the last URG command above). Luckily this is the same as what was obtained from the downwards merge.

Figure 6 shows the instance coverage of the instance of `modbase` in called `modbase_i` in TB3. The numbers show that the instance coverage generated from TB3 for this instance is merged with the module coverage generated by TB1 and TB2. This means that typically the instance coverage when doing upwards merging will be a merge of what ever the base design generated and the

Recommendation:

Depending on which coverage number you are interested in, then you should choose between downwards or upwards coverage mapping. If you are interested in
No matter whether you choose to do upwards or downwards mapping, then only module coverage makes sense and the merged numbers are the correct independently of the mapping direction. Thus, mapping provides a grand total for the particular mapped module.

8 SystemVerilog Constructs that Affects Coverage

Description:

Until now, this paper has only been dealing with very simple RTL, without any parameters or `defines. Based on experience, having a block which contains parameters or `defines in the design introduces a lot of problems when working with coverage and especially when merging coverage from multiple testbenches, where the block is instantiated with different settings of `defines and/or parameter values.

Sub block A in Figure 1 shows such a block, which can be instantiated in multiple versions depending on the value of a `define or parameter.

This section will present some of the typical problems with `defines and parameters, which are related to coverage. Thus, it will show fragments from the URG reports, generated by simulating RTLA, RTLB, RTLC and RTLD with the related tesbenches.

8.1. `defines

Description:

`defines have been around for quite a while, and need no further introduction.

Problem:

Wrong or unexpected coverage is obtained when simulating a design with `defines.

When simulating designs with `defines, one has to be careful. The situation often occurs, when a design needs to be configurable. When running regressions on this design, it is typically compiled with the `define defined and undefined. At the end of the regression, all coverage is merged. This does not always turn out to work as expected.

Design RTLB is simulated as follows in two separate directories:

```
Dir1:
vcs -cm line –sverilog +define+MY_DEFINE …
./simv -cm line +inv_val=000 -cm_name my_define.100
./simv -cm line +inv_val=001 -cm_name my_define.101
./simv -cm line +inv_val=010 -cm_name my_define.110

Dir2:
vcs -cm line –sverilog …
./simv -cm line +inv_val=000 -cm_name no_my_define.000
./simv -cm line +inv_val=001 -cm_name no_my_define.001
./simv -cm line +inv_val=010 -cm_name no_my_define.010
```

Figure 7 and Figure 8 shows a fragment from the URG generated report with and without the `define set. The figures shows how URG is capable of merging line coverage, when the `defines does not define a different number of lines. One has to be careful though, since the lines covered can contain different information, potentially shadowing for actual lines of code, which has not been covered. The report is generated by the following URG command:

```
urg –report urgReport0-1 -dir Dir1/simv.cm Dir2/simv.cm
```

Additionally, putting Dir2/simv.cm first is also tried:

```
urg -report urgReport0-1 -dir Dir2/simv.cm Dir1/simv.cm
```

Now, what happens, when the `define selects between two different code sections with a different number of lines? Design RTLC is now simulated in the same manner as design RTLB just have been simulated. Again, the results are merged, using two different URG commands:

```
urg -report urgReport1-0 -dir Dir1/simv.cm Dir2/simv.cm
urg -report urgReport1-1 -dir Dir2/simv.cm Dir1/simv.cm
```

The results are shown in Figure 9 and Figure 10. Running both URG commands generates some errors:

```
URG Version D-2009.12-1 Copyright (c) 1990-2009 by Synopsys, Inc.

Error-[CMR-VCBLKDIFF] Version Check Error:Block Difference
  Database mismatch: Number of blocks for instance "tbbase.modbase_i" in test
  file my_define1/simv.cm/coverage/verilog/test.line is not the same as in
  base design. Line coverage data of this instance will not be merged.
  Please ensure that all coverage tests are generated from same design. If
  problem still persists please contact vcs_support@synopsys.com.


Note-[URG-RDG] Report directory generated
  Report written to directory urgReport1-0
```

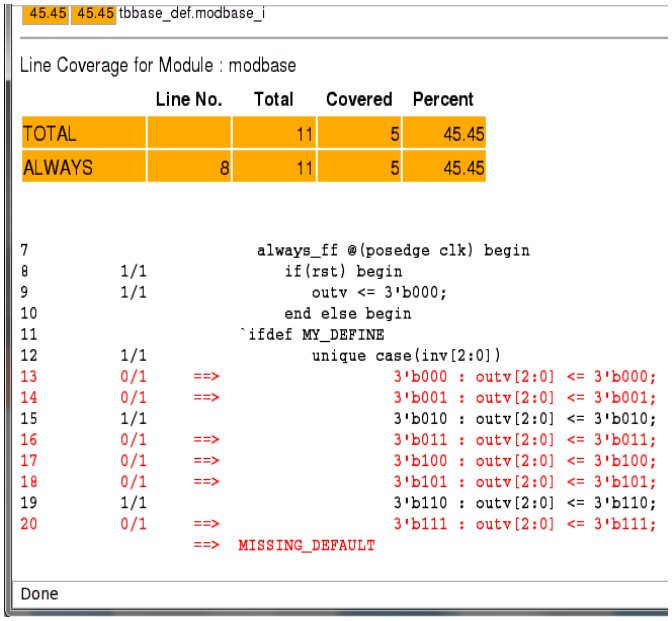


Figure 7: Coverage report for RTLB with `MY_DEFINE` set

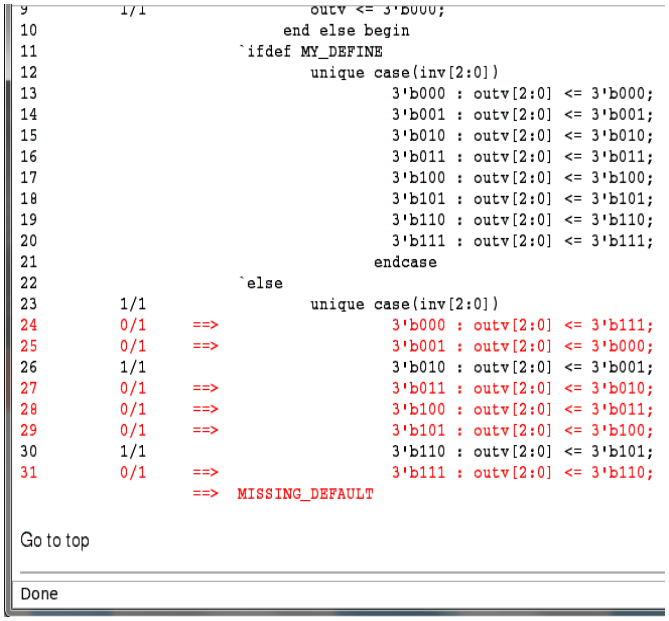


Figure 8: Coverage report for RTLB without `MY_DEFINE` set

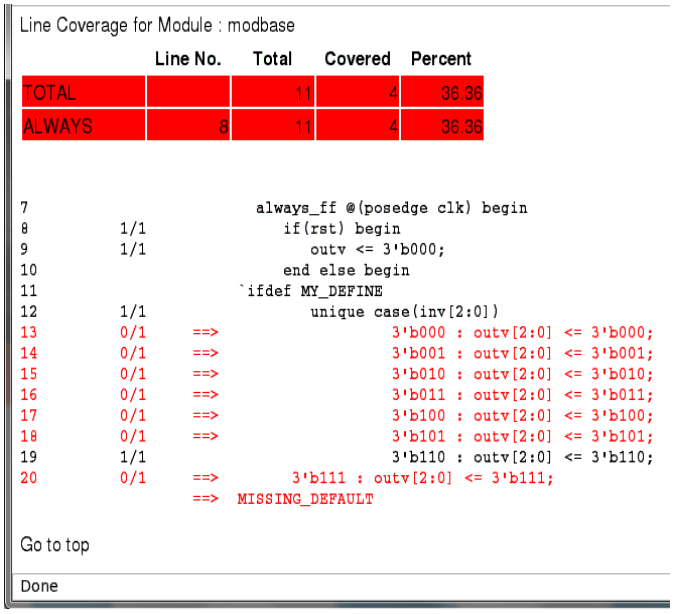


Figure 9: Coverage report for RTLC with MY_DEFINE set

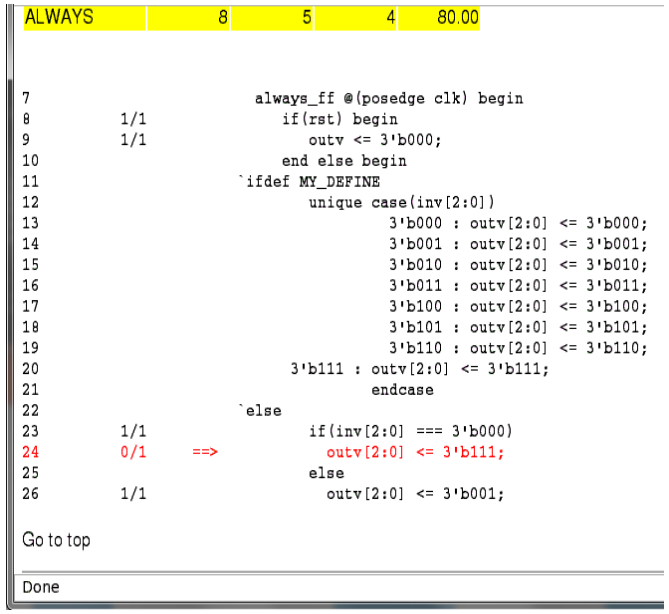


Figure 10: Coverage report for RTLC without `MY_DEFINE` set

Since the two designs are different due to the value of the `` `define ``. The generated errors were expected, but the experiment also showed that the reported coverage numbers depends on which directory is specified as the base coverage directory. Hence, the generated coverage reports might be missing some coverage numbers, when merging coverage with `` `defines `` involved.

The last situation resembles the scenario shown for block A in Figure 1, which could be instantiated in two different ways, controlled by a `` `define ``. In the real project we had to separate the merged coverage data by running two separate regression, one with and one without the define set.

Recommendation:
Based on the findings presented in this section, the recommendation is to analyze the RTL carefully if it contains any `defines and what they control. In most cases the RTL block should be simulated in separate regressions for each value of the `define. Basically, the RTL block should be treated as different RTL blocks for each value of the `define.

8.2. Parameters

Description:
Parameters are an important SystemVerilog construct, since it provides the RTL designer with the ability to produce reusable code. Reusable code is good, for many reasons, one of them being that there is less to verify. However, seen from a coverage point of view, parameters are a bit troublesome, just like `defines.

Problem:
Experience shows that verification engineers have a hard time understanding the coverage reports generated from merged coverage data from simulating designs with parameters.

RTLD is compiled and simulated with 3 different testbenches. One, which instantiates RTLD once with the default parameter value (parm3), one testbench, where it is instantiated with the value 4 (parm4) and at last, one which instantiates RTLD twice with the WIDTH parameter set to the default value and 5 (parm3_5),

To depict the problem, each testbench is executed a single time and the generated coverage databases are then merged in all possible permutations.

First, using the parm3 as the base coverage database:

```
urg -report urgReport0-0 -dir parm3/simv.cm parm4/simv.cm parm3_5/simv.cm

Partial output:
URG Version D-2009.12-1 Copyright (c) 1990-2009 by Synopsys, Inc.

Warning-[CMR-IID] Illegal Instance Data
  There is an inconsistency between the compile and runtime database. The
  instance tbbase found in compile time database is not present in the runtime
  database
  Please ensure that the compile time data is not overwritten since
  simulation. If the problem persist please contact vcs-support


Warning-[CMR-IRIF] Improper Reference in File
  The reference to scope "inv1[2:0]" in file
  "parm3_5/simv.cm/coverage/verilog/test.tgl" is not found in the design.
  Please check if the above coverage data and the compile time data are of the
  same design.

urg -report urgReport0-1 -dir parm3/simv.cm parm3_5/simv.cm parm4/simv.cm

Partial output:
```

```
URG Version D-2009.12-1 Copyright (c) 1990-2009 by Synopsys, Inc.

Warning-[CMR-VCROLI] Register Indices mismatch
  Database mismatch: Indices for TGL coverage data entry "inv[3:0]" for
  instance "tbbase" in test file parm4/simv.cm/coverage/verilog/test.tgl do
  not match "inv[2:0]" in base design.  Only data for bits that match will be
  merged.
```

The warnings show that only matching bits are merged. **Fejl! Henvisningskilde ikke fundet.** to **Fejl! Henvisningskilde ikke fundet.** shows the hierarchy list (**Fejl! Henvisningskilde ikke fundet.**), module list (**Fejl! Henvisningskilde ikke fundet.**) and the coverage for module RTLD (`mod-base_param`, **Fejl! Henvisningskilde ikke fundet.**) for the two reports generated by the two URG commands stated above.

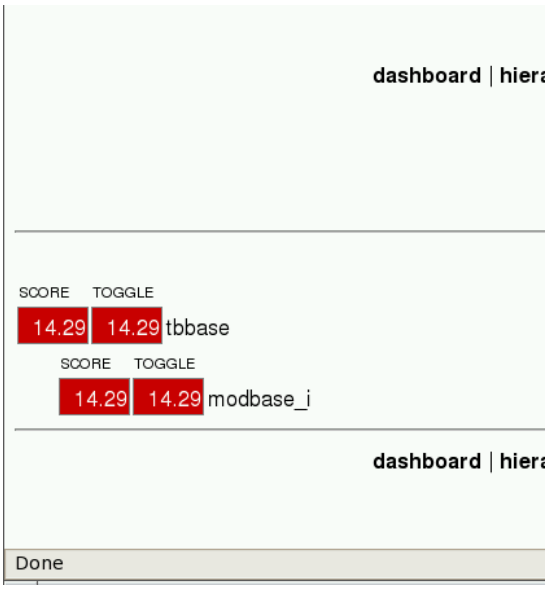


Figure 11: parm3 RTLD hierarchy list

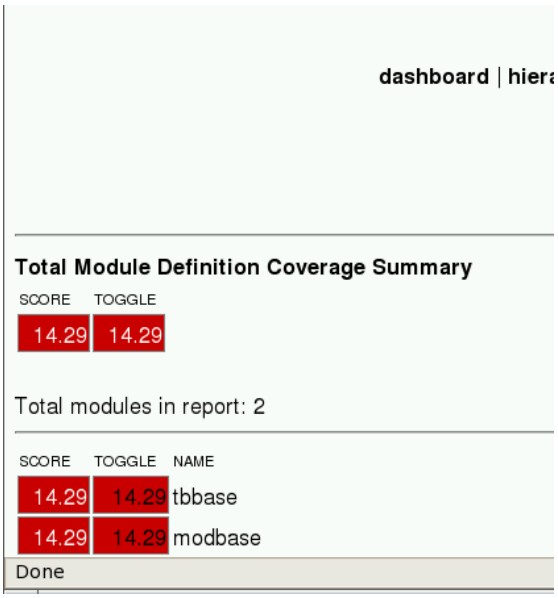


Figure 12: parm3 RTLD modlist

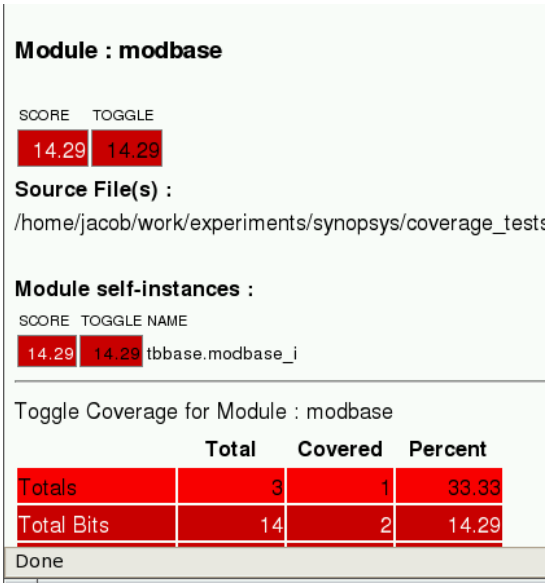


Figure 13: parm3 RTLD modbase toggle coverage

Running URG with parm4 and parm3_5 as base coverage databases:

```
urg -report urgReport1-0 -dir parm4/simv.cm   parm3/simv.cm   parm3_5/simv.cm
urg -report urgReport1-1 -dir parm4/simv.cm   parm3_5/simv.cm parm3/simv.cm
urg -report urgReport2-0 -dir parm3_5/simv.cm parm3/simv.cm   parm4/simv.cm
urg -report urgReport2-1 -dir parm3_5/simv.cm parm4/simv.cm   parm3/simv.cm
```

Returns the same kind of error messages. Now, the hierarchy and module list shows different coverage results for the different generated databases, even though they are generated from the same databases. Figure 14 to Figure 16 provides snippets oft he hierarchy list, modlist and `modbase` toggle coverage for parm4. Note that the hierarchical coverage is now 11.11% instead of 14.29% and that the total number

of bits in the `modbase` coverage report is 14 for parm3 and 18 for parm4. Figure 17 to Figure 20 shows the same for parm3_5 (Note, that there are two instances of `modbase` with `WIDTH` set to 3 and 5).

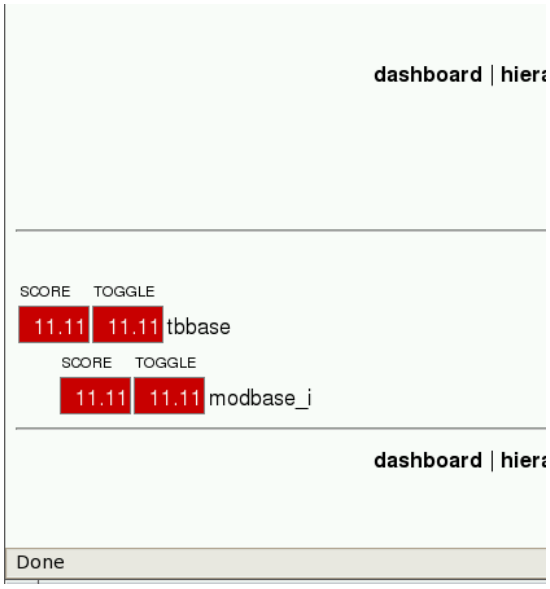


Figure 14: parm4 RTLD hierarchy list

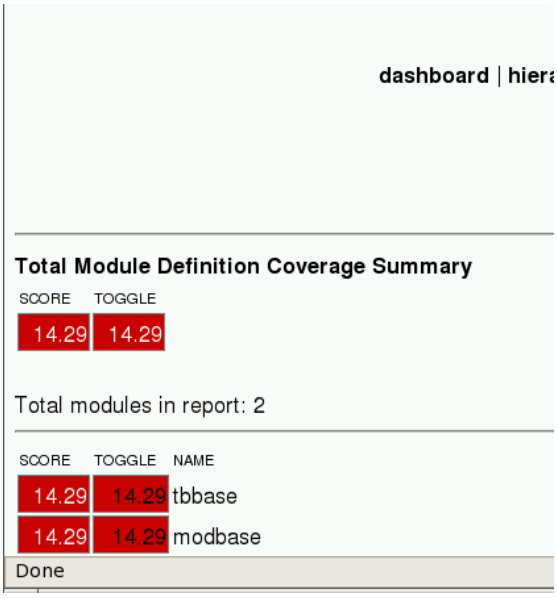


Figure 15: parm4 RTLD modlist

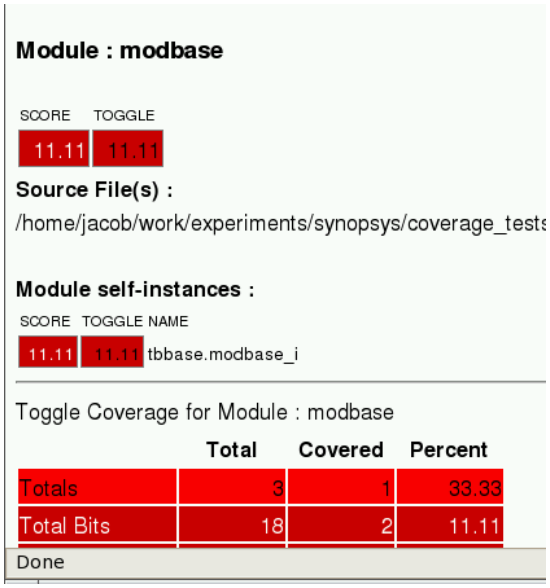


Figure 16: parm4 RTLD modbase toggle coverage

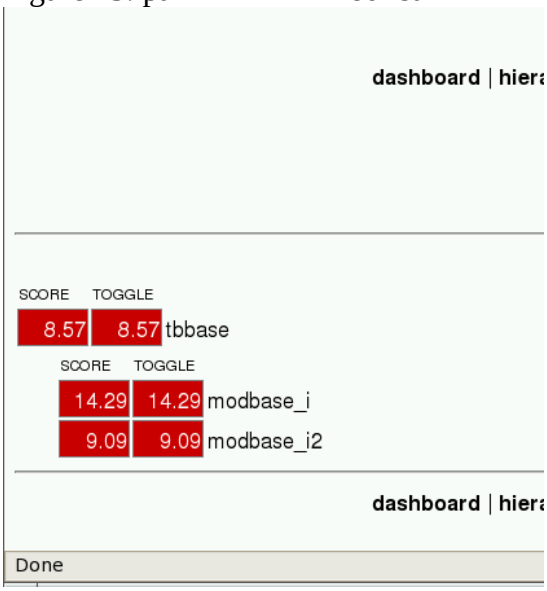


Figure 17: parm3_5 RTLD hierarchy list

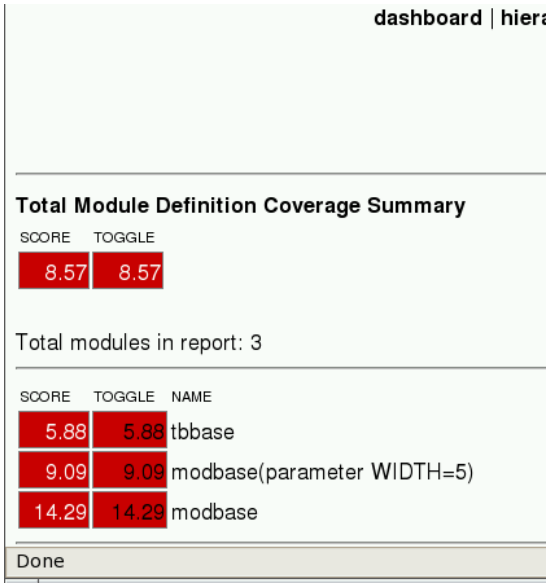


Figure 18: parm3_5 RTLD modlist

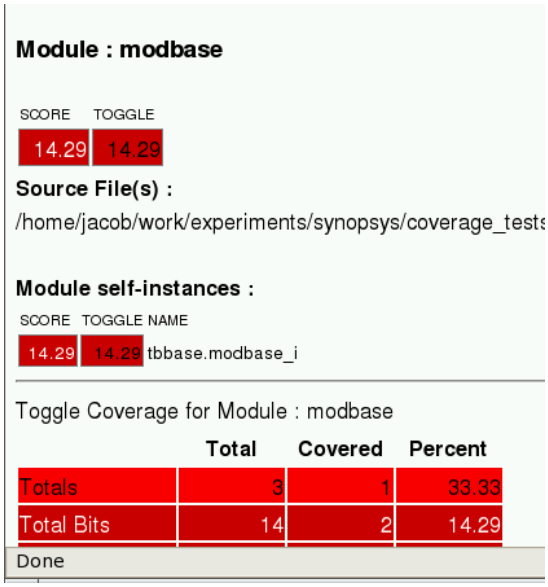


Figure 19: parm3_5 RTLD `modbase` toogle coverage (`WIDTH`=3)

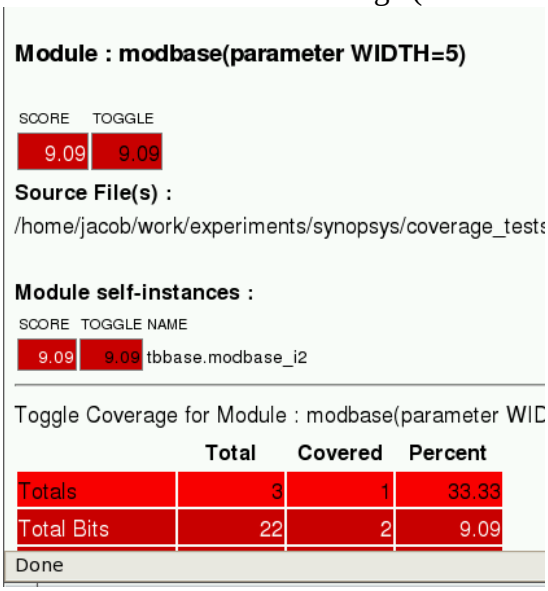


Figure 20: parm3_5 RTLD `modbase` toggle coverage (`WIDTH`=5)

Recommendation:

The experiments above show that things potentially can go wrong, when merging coverage databases with parameterizable modules. The problem is that the reported coverage depends on which coverage database is used as the base coverage database (listed first, right after the `-dir` option). Thus, the order of the directories used in the URG command actually matters. The produced result only contains the bits that match, leading to unexpected coverage results. The recommendation is to simulate the different values for the parameters in separate simulation runs, and then only merge those with have the same value if possible. Additionally, the coverage reports and output from URG should be studied carefully, so incorrect merging is caught early on. To aid this process, it would be beneficial if the underlying verification environment supported control of the order of coverage merging.

8.3. `defines and Parameters

Description:
Mixing `defines and parameters will just make things worse. The two previous sections revealed that one have to be careful, when the design contains `defines or parameters. Having a design with a mixture of those will just multiply the problems. Hence, even more attention to validation of the coverage reports should be paid.

Problem:
See **Fejl! Henvisningskilde ikke fundet.** and 8.2.

Recommendation:
See **Fejl! Henvisningskilde ikke fundet.** and 8.2.

9 Coverage Exclusion

Description:
Coverage exclusion can be done in two different ways:

1. Via the -cm_hier and other VCS® options. This allows the verification engineer to exclude modules or specific instances etc. This method only works for structural coverage and it has existed for quite a while. See section 5 for more information about this.
2. Via a generated coverage exclusion file. The file is either written directly in a text editor or generated by using DVE. The generated exclusion file can then be used to exclude coverage in coverage reports generated with URG, by using the -elfile option for URG. This method is fairly new and it works for both structural and functional coverage.

Problem:
Coverage exclusion using method (1) above is fairly trouble free. However, method (2) has a few caveats. The following snippet shows a coverage exclusion file generated from DVE by another user than the author for RTLA:

```
//================================================
// This file contains the Excluded objects
// Generated By User: benny
// Date: Fri Mar 12 12:57:59 2010
// ExclMode: default
//================================================

FILE:
/home/benny/esnug/experiments/synopsys/coverage_tests/hitchhiker_test/hdl/designs/
modbase.sv
INSTANCE: tbbase.modbase_i
Line 10
Line 16
```

Simulating RTLA and generating the coverage report with the exclude file, results in:

```
urg -lca -elfile $PROJECT/exclude/some.el -format both -report urgReport.elfile -
dir ./simv.cm
```

```
Note-[URG-RDG] Report directory generated
  Report written to directory urgReport.noelfile

URG Version D-2009.12-1 Copyright (c) 1990-2009 by Synopsys, Inc.

…

Excluded item does not exist:
Warning-[UCAPI-EL-INVLINE] Invalid line in exclude file
  Invalid line number '10.0' found in the exclude file for Line coverage.
  Please rewrite/regenerate the exclude file.


Note-[URG-RDG] Report directory generated
  Report written to directory urgReport.elfile
```

even though line 10 exists in RTLA. The problem is the hard reference to the file name, since when the author ran the command with the exclude file, the path is not correct. The path can be left out, but this turns off synchronization checks.

Recommendation:
The coverage exclude files are a nice new feature, to fine tune coverage numbers, if e.g. the RTL has features, which are not used in the current design. Then coverage exclude files, provides a mechanism for specifying precise reports that reflects what actually have to be verified. However, experience shows that verification engineers will spent a lot of time redoing these files, when generated with DVE, due to the timestap/file reference problem. Thus, the recommendation is to use this feature late in the project, when the RTL is fairly stable.

10 Combining All Coverage Features

The previous sections 5 to 9 have described different coverage related problems, based on experiences gained from verifying a real ASIC.

The sections depict how many issues the verification engineer faces, when the RTL is generic etc. Furthermore, in the real project, the A0/1 block was not 100% verified in its own testbench, but it relied on verification of sub-block C and the top level verification as well. Thus, coverage reports, merged from data from all testbenches, were crucial for the project. Many hours where spent on the underlying coverage flow, so that is could setup the correct order of merging between coverage databases generated by different testbenches etc.

Coverage exclusion by means of the `-cm_hier` file was used and briefly the newer coverage exclusion mechanism was used as well. This was then dropped later in the project, since this feature was still in beta at that moment.

The overall recommendation is to allocate some resources to analyze the different blocks in the DUT with respect to the coverage, so a flexible flow can be build, so for instance the merging order can be controlled. Additionally, coverage merging strategy for each block is a good idea, so `` `defines ``, parameters etc. are handled correctly.

11 Coverage Reporting

Description:
This section will briefly describe the different ways of generating coverage reports. Moreover, it will provide general flow diagrams for these coverage report flows.

Coverage reporting can be done in three different ways:

1. Using CoverageMetrics: cmview.
2. Using URG.
3. Using `dve -cov`.

CoverageMetrics mentioned in item (1) above is considered deprecated by the author. The functionality should be removed from the VCS installation, since it confuses people (well, at least the author), but it is probably kept there for backward compatibility.

URG is to be considered as the primary tool for merging and generating coverage reports (item 2 above). This is the recommended way for batch processing, potentially producing trend graphs based on VMM planner input. A general flow based on URG is shown in Figure 21. Such a flow should most users of VCS already have in place. The flow handles multiple VCS® simulations and coverage report generation with URG potentially using exclusion files generated with DVE.

DVE (item 3 above) does not provide means of generating reports like URG does. However, it provides a clickable gui, where coverage databases are visualized and inspected. Additionally, this is used to generate coverage exclude file and VMM planner HVP files.

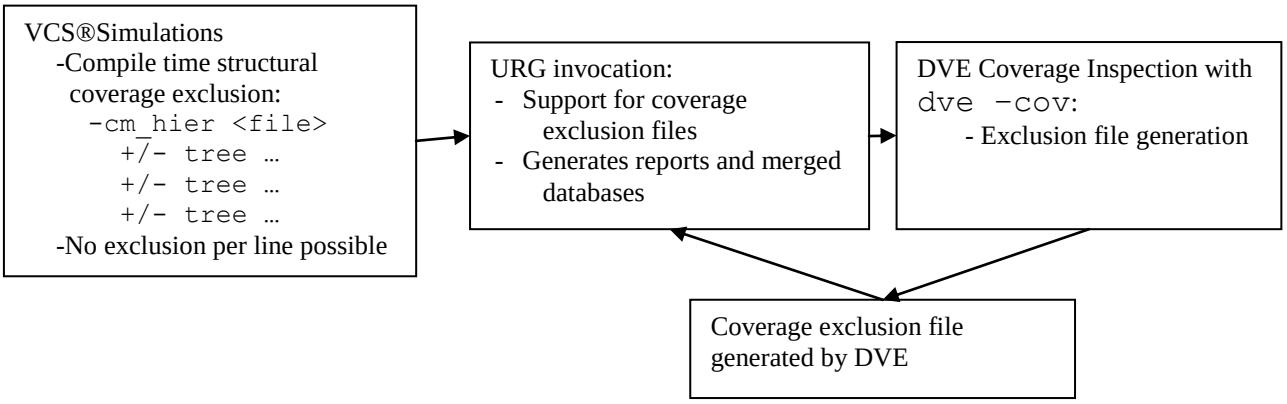


Figure 21: General batch coverage flow with URG

With VMM Planner, the coverage framework can be extended. VMM Planner can be used to highlight the most important coverage numbers and separate them into a separate report which only contains those numbers. These numbers can then be tracked over time, so trend graphs etc. can be generated. This provides an effective and powerful tool for the verification engineer since he/she can now generate reports that contain only what the manager needs instead of the full URG html reports, which tends to a bit overwhelming. Figure 22 provides the extended flow, where VMM Planner has been included.

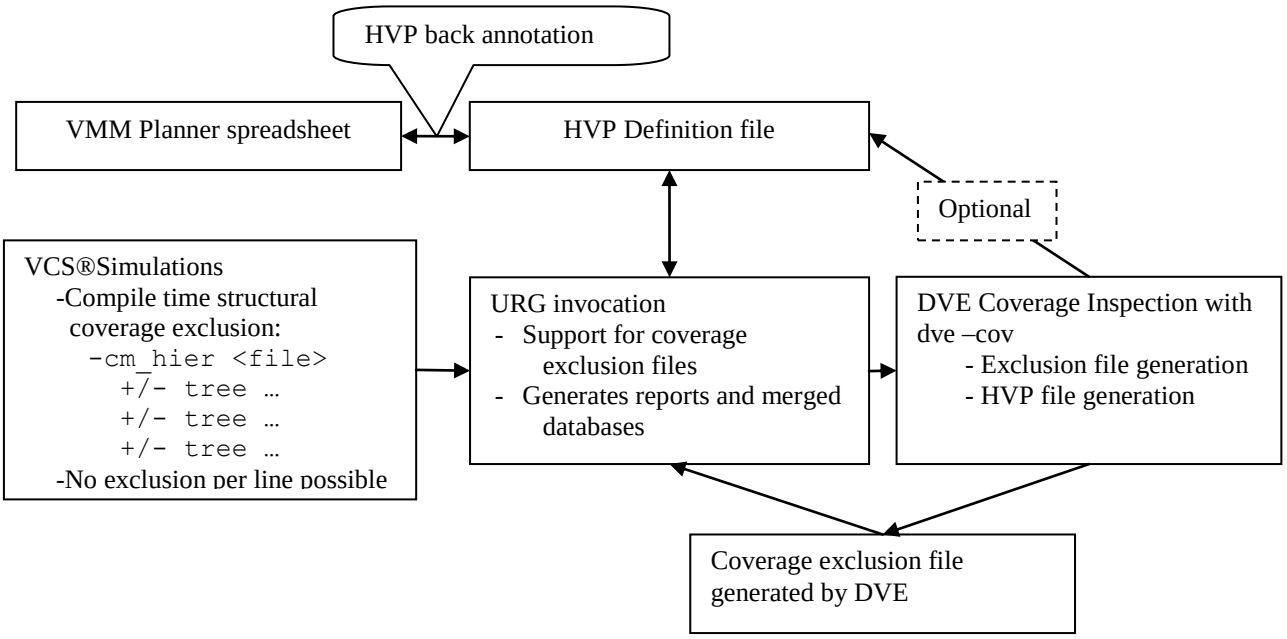


Figure 22: General batch coverage flow with URG and VMM Planner

It is optional whether the HVP file is generated from a spreadsheet or with DVE, but back annotation is only available, if it is generated from a spreadsheet.

Problem:

Even though the flows reveal that the VCS® simulation platform provides many advanced coverage features, then these options are provided by stand alone tools. The arrows in Figure 21 and Figure 22 show how to connect these tools. Hence, the arrows shows where manual scripting work is needed to bind all of it together into a usable verification environment with proper coverage support. This is the coverage reporting problem in a nutshell – knowing what tool to use when and how to connect them.

Recommendation:

Use the tools where they excel! There is some redundancy, e.g. the reports used for coverage inspection, which are generated by URG are either in text or in html format, but DVE can also be used for coverage inspection.

The recommended usage for VCS, URG, DVE, etc. is:

- VCS/simv:
  - Compile with only structural coverage for the RTL enabled.
  - Use –cm_name for each test run.
  - Reuse the simv as much as possible. Thus, coverage between tests are automatically merged.
- URG:
  - Use for merging between testbenches.
  - Use for coverage exclusion.
  - Use for HVP handling: back annotation, trend graphs etc.

- DVE:
    - Use for coverage inspection. This is preferred over reading the coverage html reports generated by URG.
    - Use for coverage exclusion.

- Spreadsheet:
    - Use for HVP generation/definition.

Additionally, it is recommended to carefully look into the generated reports and URG output to validate that the reported numbers are as expected and not suffering from any caveats (as explained in section 6 to 9).

12 Conclusion

The tools VCS, DVE and URG from the VCS® tool chain do a good job. Together they provide a feature rich coverage platform. In general they produce expected coverage result, even when the input RTL is tricky, when you actually analyze the scenario to be covered.

Thus, the conclusion is quite clear. The tool chain is working, but when moving away from simple coverage merging and reporting, then more skilled resources are needed, if the verification should be successfully based on coverage metrics. The paper has given some recommendations for handling various issues, which coverage based verification will run into during the lifetime of a project. In general, the verification engineers should be very careful and spent some time and effort on getting a proper flow implemented, which provides enough flexibility, so that different coverage strategies for different modules can be supported. If such a flexible flow is in place and skilled people have analyzed and handled all the subsequent coverage related problems in each testbench, then managers etc. should be able to rely on the generated coverage reports. Furthermore, section 11shows that extending the frame work to utilizing VMM planner completely requires substantial effort on the flow side, but if generically implemented, then the probability of reuse is quite high.

13 References

[1] Synopsys VCS® 2009.12 Documentation.