Taming the beast: A smart generation of Design Attributes (Parameters) for Verification Closure Using Specman

Meirav Nitzan, Xilinx Inc.
meiravn@xilinx.com
Yael Kinderman, Cadence Design Systems Inc.
yaelk@cadence.com
Efrat Gavish, Cadence Design Systems Inc.
efratg@cadence.com

Abstract

Design IPs are characterized by a set of ***design attributes***, or ***parameters***, which affect the way the design behaves. Moreover, it often results in a parameterized Test Bench. In order to reach a verification closure, the design needs to be tested with all relevant combinations of parameter values, sometimes exhaustively. Since parameters values are provided to the simulation at design elaboration time, figuring out the proper permutation sets of parameters needs to be done prior to the simulation run.

This paper will present an innovative solution to the parameters generation problem, which is based on two principals:

1. Smart planning of the parameters sets required to accomplish thorough regression testing.
2. A unique technology by Specman Elite which technically enables specifying the characteristics of "interesting" parameters sets, and generates parameter sets accordingly in an efficient manner with no redundancies

The paper will then describe a case study, showing the superiority of the proposed solution over simple SystemVerilog randomization, for achieving coverage closure on the parameter sets.

The paper will conclude with a summary of the proposed solution and its advantages for parameterized design verification.

Keywords

OVM, UVM, SystemVerilog, parameters, coverage, closure, verification methodology, testbench, Specman Elite

Introduction and Problem Definition

Parameterized Design IPs are very common in the FPGA world, and may be used in ASIC designs as well. These attributes, or design parameters, affect the way the design behaves. For example: if the design IP is that of a RAM, then the WRITE_MODE could determine how data is stored in the RAM, and whether or not the data output of the RAM reflects the input data. In order to reach a verification closure, the design needs to be tested with all relevant combination of parameter values, sometimes exhaustively. For example – all major design modes need to be tested with all possible data-width values.

When using OVM [1] or UVM [2] based verification environments, often times the environments themselves become parameterized as well [3]. For example – if the designs data bus width is varying, it will cause the design interface to become parameterized, and hence the monitor, driver and other such components to become parameterized as well.

The simple solution to providing high quality IP to the verification challenge above would be to create an exhaustive permutations set of all parameters, and fully verify the design with each permutation. In other words – run a full regression with all possible combinations of all possible values of the design parameters. The challenges with this solution are:

1. Not scalable.

   i. For example - for a design with 20-30 parameters, each having 2 to 10 values, the

exhaustive permutation set could reach hundreds of thousands in size, or even more.

ii. Extensive simulation time - poses a heavy burden on resources.

iii. Covering all permutations for more complex designs may not be feasible

2. Efficiency - turnaround time for running a regression is too long. If a bug fix takes a week to fully verify, designer productivity is severely impacted.

3. Functionally - there is no functional justification to cross all values of all parameters with each other.

It is clear, therefore, that simply calculating all permutations and applying the parameters sets to the regression test suite is not a good, efficient solution for parameters generation.

Solution Requirements

The requirements for covering the design parameters space are, therefore, the following:

1. **A pre simulation step** – the required solution needs to run before simulation, efficiently generating the parameters sets and providing them to the simulator test runs.
2. **Automated random generation** of parameters sets, considering all parameters legal values and constraints, as well as the dependencies between parameters
3. **Control over the distribution** of the generated parameters sets. For example: Avoiding repetitions of parameters sets, since each parameter set implies a new design configuration, and hence require the whole test suite to be simulated with it.
4. **Flexibility** to generate either full parameter sets generation, as well as smaller subsets of the parameters sets, based on the verification needs (complete design verification vs. nightly regression etc.)
5. **Proper coverage** of the parameters space needs to be verified.

Why isn't a simple SystemVerilog randomization good enough

Since all verification environments in Xilinx are OVM/UVM based, all the design parameters and their constraints can be defined in one configuration class. The parameters are defined as **rand** fields, with constraints and implied constraints, describing valid parameter values and how they affect each other.

Potentially, it should have been enough to simply randomize this configuration class separately as a pre-simulation step, and thus create legal parameters sets. Coverage group defined in the class could tell us how much of the parameter space we have covered so far [4]. This solution would satisfy quite a few requirements: it is a pre-simulation step and an automated random generation of the parameters set. The coverage group defined in the class satisfies the last requirement, for proper coverage of the parameters space.

Experience shows, however, that this solution does not satisfy the remaining two requirements: control over distribution, and flexibility to define which parameter combinations to exhaustively generate. A **case study** shown later in this paper will demonstrate those problems. Hence, a different solution is called for.

A basic capability in the *e* language the Specman Elite engine, which does not exist in SystemVerilog, was the basis for the solution presented in this paper. This basic capability was enhanced by Specman R&D, and is presented in the next section.

The solution outline

The Xilinx Global Verification Team along with Cadence Specman R&D came up with an innovative solution to the parameter generation as a pre-simulation step. The full solution required a **methodological approach** to the design parameters analysis, done by the Xilinx team, and an **enhancement to existing Generation Engine** in Specman, done by Specman R&D.

The methodological approach to solving the problem:

1. Analyze your design and identify the groups of parameters which need to have all their

permutations tested. For example: test all data width values with all write modes, test all date width values with all read modes. It is NOT necessary to test all read modes with all write modes, though.

2. Identify sub-groups of the complete list of "interesting" parameters groups specified in (1) for various purposes: a sub-group that validates a certain feature; a sub-group for nightly regression, a sub-group for sanity testing or for code check-in gating etc.

Once these parameters relationship have been established, a technical solution that generates random/exhaustive sets of values is needed.

The technical Description of the Parameter Sets Generator

The *e* language, as it turns out, does provide a good solution for the problem. Besides having the capability to generate random constrained values, it has the capability to **exhaustively** generate all valid values of certain fields in a class, while randomizing the others.

So, for example, let's look at the following *e* struct

```
struct foo{
  width: uint [1, 2, 4, 8, 16];
  mode: uint (bits:2);
  init_val: uint(bits:16);
 keep (mode == 0|| width ==16) => (init_val ==0);
}
```

Using the *e* function *all_iterations()*[5], the user can ask for a set of *foo* objects that include all valid combinations of {*width, mode}* (set size would then be (4*5) =20 objects), and in each one of these objects, the *init_val* field will have a random value that applies to the constraint *keep (mode == 0|| width ==16) => (init_val ==0);*

The *e* features described above almost fully satisfy the solution requirements. However, in a real-life case of design parameters sets, there is more than one group of parameters (such as {width, mode}) that needs to be exhaustively generated. Creating different lists for different parameter groups can solve that requirement, but then it would result in lots of duplications in those parameters between the lists.

Hence, Specman R&D came up with an enhancement to the generation engine, by providing a generation construct which allows the user to define **distribution traits** of an "interesting" parameters set.

The new generation construct enables defining the following:

- Exhaustive rules:
  The user can specify a set of attributes (x_1,..x_k) that needs to be exhaustively exercised. i.e. any generated parameters set must include all possible permutations of (x_1,..x_k)
- Non-Exhaustive rules:
  The user can specify a set of attributes (y_1,..y_m) that needs to be non exhaustively exercised. i.e. any generated parameters set should try to avoid generating 2 solutions that repeat same value of y_i (for each i in [1..m]).
- Repetition rule:
  The user can specify a repetition counter that defines how many times each permutation of the exhaustive set of parameters should appear in the generated parameters set (by default , the repetition counter is 1).
  Alternatively, the user can use it to force the size of the generated parameters set.

When the new generation construct (a macro) is used, Specman generation engine will generate exhaustive/random parameters sets, complying with the rules above in an efficient manner (In O(size of generated set)).

The solution components

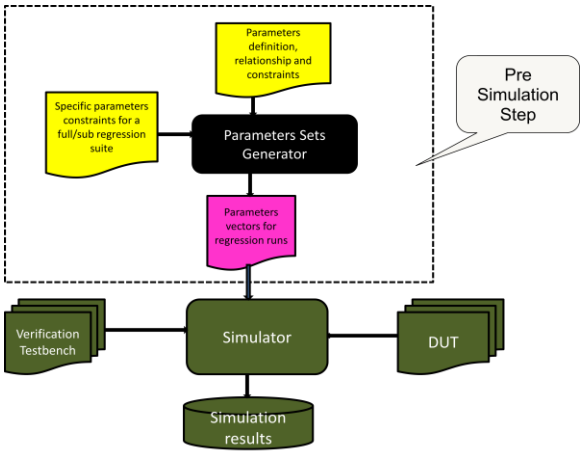

Figure 1 – parameter generation solution outline

The **inputs** to the parameters sets generation engine are, the following (as shown in figure 1):

1. A class containing design parameters definition along with their constraints and relationship – needs to be done once for a design.
2. An *ini* format file which defines more constraints and sub-groups of parameters which need to be exhaustively exercised with respect to each other. There may be several sub regressions defined in this input file (e.g. – nightly, full, feature X testing etc.).

The input files described above are processed by the Parameter Sets Generator in the following way: The *ini* file input is translated to further constraints expanding the original *e struct*, relying on *e*'s Aspect Oriented Programming capability – extensions of structs with added constraints (similar to SystemVerilog class inheritance with added constraints); The specification of the parameters groups to be exhaustively generated is translated to a Specman macro call.

The Specman based generation engine receives the original parameters struct, along with the information above. The result is sets of parameters which are written to output files.

Case study – verifying a Dual Port RAM

Verification Challenge Presentation

The Dual Port RAM is a two port RAM device. Each one of its ports can independently serve read or write requests. Hence, its OVM test bench is comprised of two independent agents, and a virtual sequencer coordinating their stimulus generation, as shown in figure 2:

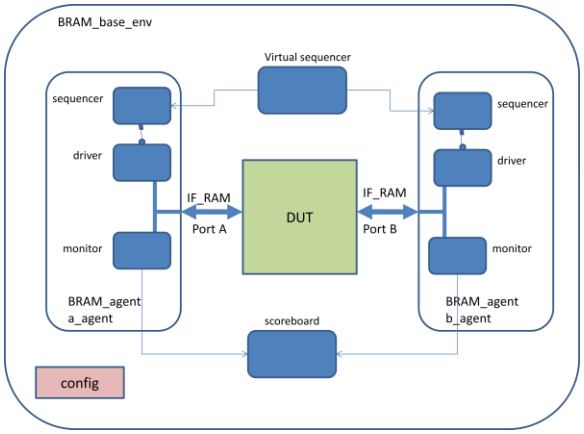

Figure 2 – BRAM: DUT and Test Bench

The design IP has 12 major parameters which define its logical behavior, and a few other parameters such as initial memory value or reset value. The total number of possible permutations of the major parameters reaches 180,000, which means running a full test suite on each set, or a week worth of regression time.

Applying the Smart Parameter Generation Solution

The first principal of the parameter solution calls for careful analysis of the parameter space, and defining the necessary exhaustive combinations of parameters sets. After applying this principal we ended up with a revised verification plan which specified a much smaller sets of exhaustive permutations of parameters sets, and thus reduced the total number of resulting parameters sets.

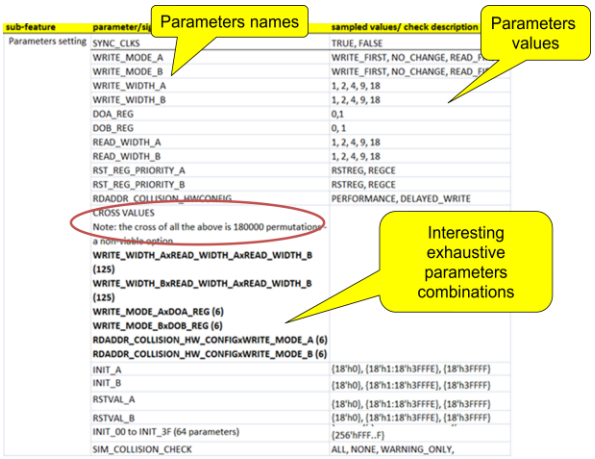

Figure 3 – a partial verification plan view for design parameters

Since the BRAM test bench needs to be aware of those parameters and their values, a special configuration class is defined. The parameters are represented as random fields and their legal values, as well as the dependencies between parameters values, are represented by constraints. The resulting SystemVerilog configuration class is shown in figure 4.

In order to enable the Specman based solution, a one-time effort of translating this SystemVerilog class to an *e* struct was required.

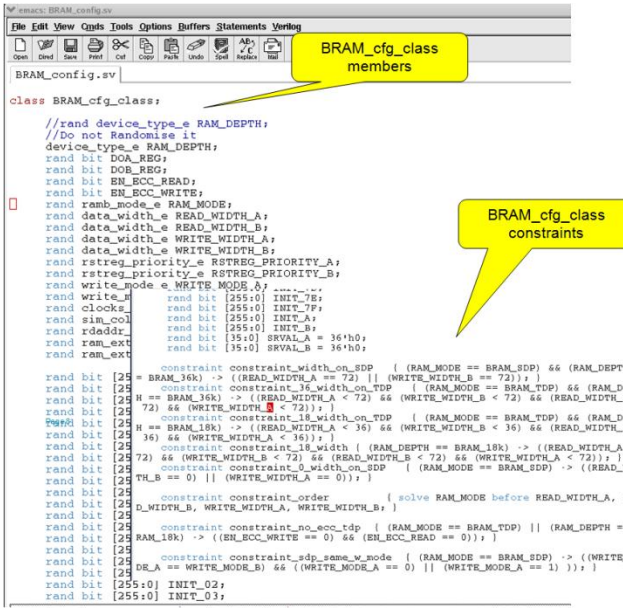

Figure 4 – config parameters SystemVerilog class

Defining a simple sub-regression

We defined a simple sub-regression, with further constraints on some parameters, and specified which parameters combinations need to be exhaustively exercised:

Port A: READ_WIDTH is 18 or 36, WRITE_WIDTH is 18 or 36

Port B: READ_WIDTH is 18 or 36, WRITE_WIDTH is 18 or 36

Exercise ALL combinations of:

*READ_WIDTH_A x READ_WIDTH_B (2*2=4),*
*WRITE_WIDTH_A x WRITE_WIDTH_B (2*2=4)*

The above requirements are presented in the following way in the parameter generation *ini* file:

[lite]

CROSS param.WRITE_WIDTH_A = Ta[18, 36]

CROSS param.WRITE_WIDTH_B = Ta[18, 36]

CROSS param.READ_WIDTH_A = Tb[18, 36]

CROSS param.READ_WIDTH_B = Tb[18, 36]

TB_OVM_TESTNAME = simple_test

The Parameter Sets Generator takes the sub regressions definition above and effectively adds it as added constraints to the relevant parameter fields. It then generates a set of values in which the following is satisfied:

- All combinations of READ_WIDTH_AxREAD_WIDTH_B and all combinations of WRITE_WIDTH_AxWRITE_WIDTH_B exist, with minimal repetition of each cross (x) value set.
- The rest of the parameters have random values

Comparing the Specman Based solution with SystemVerilog random generation

In order to run a comparison between the solution proposed in this paper, and a simple SystemVerilog randomization of a constrained Configuration class. We built a dummy top module, which only instantiates the configuration class, and prints out the fields values.

```
`include "BRAM_config.sv"
class lite_BRAM_cfg_class extends BRAM_cfg_class;
  constraint c3 { READ_WIDTH_B inside {18,36};}
  constraint c4 { READ_WIDTH_A inside {18,36};}
  constraint c5 { WRITE_WIDTH_B inside {18,36};}
  constraint c6 { WRITE_WIDTH_A inside {18,36};}
endclass: lite_BRAM_cfg_class


class cfg_cov;
   lite_BRAM_cfg_class bram_cfg;
  covergroup cg;
    READ_WIDTH_A: coverpoint bram_cfg.READ_WIDTH_A{
   bins ra[] = {18, 36};
   illegal_bins others = default; }
    READ_WIDTH_B: coverpoint bram_cfg.READ_WIDTH_B{
  bins rb[] = {18, 36};
  illegal_bins others = default; }
    WRITE_WIDTH_A: coverpoint bram_cfg.WRITE_WIDTH_A{
  bins wa[] = {18, 36};
  illegal_bins others = default; }
    WRITE_WIDTH_B: coverpoint bram_cfg.WRITE_WIDTH_B{
  bins wb[] = {18, 36};
  illegal_bins others = default;  }
    Ta: cross WRITE_WIDTH_A, WRITE_WIDTH_B;
    Tb: cross READ_WIDTH_A, READ_WIDTH_B;
  endgroup: cg
    function new();
  cg = new;
   endfunction // new


  function void sample(lite_BRAM_cfg_class cfg);
    bram_cfg = cfg;
    cg.sample();
  endfunction // sample


  function bit is_coverage_done();
    return cg.get_coverage() ==100.0;
  endfunction
endclass: cfg_cov
```

The top module instantiates an object of type lite_BRAM_cfg_class, randomizes it and stop randomization when a 100% coverage is achieved.

```
module top;
 lite_BRAM_cfg_class bram_cfg;
  cfg_cov cov;
  int unsigned count;

  initial begin
    bram_cfg  =new;
    cov  =new;
    count = 0;

    while (count < 1000) begin
     assert(bram_cfg.randomize());
     $display("WRITE_WIDTH_A = %d, WRITE_WIDTH_B =
%d\n",  bram_cfg.WRITE_WIDTH_A,
bram_cfg.WRITE_WIDTH_B);
      $display("READ_WIDTH_A = %d, READ_WIDTH_B =
%d\n",  bram_cfg.WRITE_WIDTH_A,
bram_cfg.WRITE_WIDTH_B);

     cov.sample(bram_cfg);   count ++;

     if(cov.is_coverage_done()== 1) begin
      $display("***needed %0d cycles to complete coverage",
count);
        $finish;
      end
endmodule // top
```

Several runs on the code shown above shown that the number of runs required between 7 and 26 runs:

```
 ***needed 7 cycles to complete coverage
Simulation complete via $finish(1) at time 0 FS + 0
./sv/BRAM_top_test.sv:78          $finish;
```

Vs.

```
***needed 26 cycles to complete coverage
Simulation complete via $finish(1) at time 0 FS + 0
./sv/BRAM_top_test.sv:78          $finish;
```

In contrast, running the proposed Specman-based solution proposed in this paper yielded **between 4 and 6** permutations of the parameters sets. The code generated by the Parameter Sets Generator script for Specman was:

```
< '
extend CONFIG_NAME_T : [lite];

extend lite config_s {
  keep RAM_DEPTH in [36];
  keep RAM_MODE in [TDP];
  keep READ_WIDTH_B in [18,36];
  keep READ_WIDTH_A in [18,36];
  keep WRITE_WIDTH_B in [18,36];
  keep WRITE_WIDTH_A in [18,36];
};
```

```
ADVANCED_ALL_ITERATIONS
sys.config_gen.config_list_lite_full type=lite config_s
exhaustives=((READ_WIDTH_B,READ_WIDTH_A),
(WRITE_WIDTH_B,WRITE_WIDTH_A),
(RAM_DEPTH,RAM_MODE)) repetition=1;

extend config_gen_unit {
  !config_list_lite_full: list of lite config_s;
  fill_config_list() is first {
    for each (conf) in config_list_lite_full {
      config_list.add(conf);
    };
  };
};

'>
```

The reasons for the differences in results

The Specman-based solution is first exhaustively generating the first specified group, READ_WIDTH_A x READ_WIDTH_B, while simply randomizing the other parameters, and hence creates 4 parameter sets for the entire parameters set. Then, it looks at the other specified exhaustive group, WRITE_WIDTH_A x WRITE_WIDTH_B. Now, since some combinations of the second exhaustive group were already exercised when handling the first group, the engine only had to add the non-exercised permutations of that group.

The SystemVerilog solution, on the other hand, randomizes all parameters in the BRAM_config_class with no special attention to the combinations we would like to see exhaustively exercised, since randomization does not consider coverage description. Hence, reaching the desired combinations could take a greatly varying number of times.

Summary and Conclusions

In this paper we presented a new and efficient solution to the problem of generating parameters for parameterized design and test bench regression runs. We have shown through a case study that this solution generated a minimal number of parameter sets, compared to a simple SystemVerilog randomization, which results in a number of parameter sets that greatly varies, and in the worst case is 4 times larger than the worst case of our solution. Since each set of parameter values needs to have a regression test suite run with it, each such redundant parameter set is a significant waste of simulation time and company resources.

References

[1] Open Verification Methodology (OVM), http://ovmworld.org

[2] Universal Verification Methodology, http://uvmworld.org

[3] B. Ramirez, M. Horn, OVM & Parameters: Why Can't They Just Get Along?, http://go.mentor.com/parameters_and_ovm

[4] IEEE Standard for SystemVerilog: Unified Hardware Design, Specification and Verification Language, IEEE Std. 1800-2009

[5] Incisive® Enterprise Specman Elite® Testbench **Specman e Language Reference**