2019 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION INDIA

Goal Driven Stimulus Solution

Get yourself out of the redundancy trap

Rohit Bansal, Samsung Semiconductor India R&D, Bangalore (rohit.bl@samsung.com)

Abstract - **Functional coverage is a key metric for verification closure and constrained random simulations have become the industry practice for achieving that goal; however it does not ensure with complete certainty meeting the coverage goals, even after running multiple random seeds or increasing volume of data traffic. This paper proposes a novel method for writing goal driven tests by use of smart constraint modelling, governed by feedback for reaching the coverage goals with certainty. This approach has a number of benefits including: faster automated coverage closure, saving regression resources and associated costs, random stability concerns for derivative projects. This paper discusses practical examples of problems faced, the proposed solution and demonstrates application on live project which resulted in significant savings.**

I. Introduction

Functional coverage is a key sign off criteria for verification closure. Constrained Random simulation and coverage collection today relies on large numbers of tests and merging coverage across those tests. Empirical data pertaining to effort spent in coverage closure suggests significant verification cycle time and resource utilization in terms of LSF usage, VIP and Simulator license requirement and human effort. The size and complexity of modern IP and SOCs result in huge run times which can add critical amount to the resource budget if the redundancies that come with random regressions are left uncared for. Coverage hole analysis and filling the gaps is usually time consuming and laborious. Bugs found during this activity; usually kept in last stage, further impact the design cycle.

It has become a general practice to rank regression tests in order to be able to reproduce coverage results for design derivatives with minimal effort. However, randomization results are highly sensitive to change in stimulus generation code due to which test ranking works only for limited use cases and significant resources are needed again where designs are scaled. Cumulated over various product versions, this becomes a sizable and often repetitive effort. More efficient processes are needed to reach coverage goals faster with less engineering effort.

The proposed approach introduces the concept of efficient goal driven test case coding using smart constraint modelling. The solution presented here shows how it can be made possible to avoid repetitions and generate simulations with unique configurations and data. By using feedback from earlier randomization calls, the solution space for constraint solver can be changed to exclude already randomized values. It is possible to model constraints that span across various variables to target all possible variable combinations. Exploiting the MSIE flow supported in all major simulators, the feedback from one run can be incorporated into subsequent re-runs. Simple scripts were implemented for waiting on randomize call of previous run to complete for ensuring parallel runs during the time consuming simulation phase.

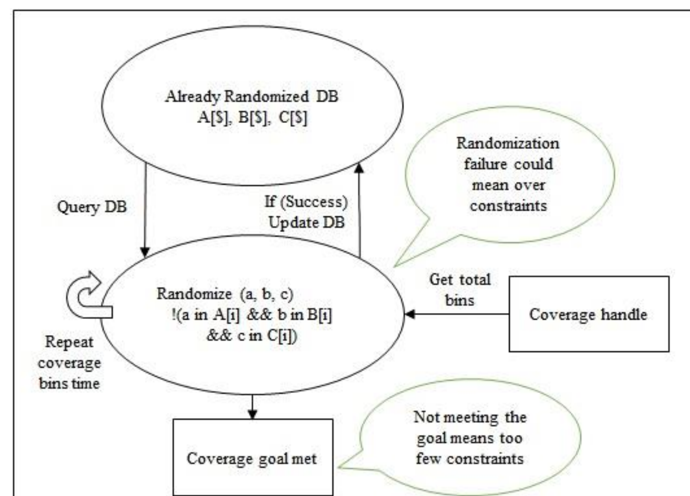

Figure 1. Stimulus Generation Methodology

The implementation of this method has been done keeping in mind various aspects - reusability, ease of addition in existing verification environments, not to break multiple run parallelism in regression and cover practical functional coverage cases like crosses, value ranges, etc. This paper discusses the implementation of this approach in detail and the numerous benefits over existing test modelling.

II. Related Work

Feedback based coverage closure verification methodologies have started cropping up in recent times. Inspirations for this thought and effort include papers on coverage closure methodologies proposed by various authors some of which are mentioned below.

- Improving Constrained Random Testing by Achieving Simulation Verification Goals through Objective Functions, Rewinding and Dynamic Seed Manipulation (https://dvcon.org/sites/dvcon.org/files/files/2017/07_1.pdf)
- Coverage Closure – Is it a "Game of Dice" or "Top 10 Tests" or "Automated Closure"? (https://dvcon-india.org/sites/dvcon-india.org/files/archive/2015/proceedings/111_Coverage_Closure.pdf)

These approaches achieve promising results over conventional practices. However these have shortfalls which add challenges for their use in practice. The major limiting factors being their re-usability across variety of test bench coding styles, ease of adoption. The solution discussed here tries to overcome these shortfalls through its simple idea of using the simulation tool's constraint solver smartly making it easier for adoption in real life across various applications.

III. Implementation

The demonstration for constraint modelling is done using a simple test randomizing two sequences with the first one targeting length and size for a packet; the second one targeting different values for some configuration variable config_a. For simplicity purpose the test is coded with coverage model built inside the test and coverage collected directly from sequence variables. This example will help in understanding how to apply this method for guiding the constraint solver; first targeting individual variable coverage and then cross coverage, ranged bins, etc.

```
import uvm_pkg::*;
`include "uvm_macros.svh"
`include "optimize_macros.svh"

class traffic_seq extends uvm_sequence;
  `uvm_object_utils(traffic_seq)
  rand int length;
  rand int Size;
  constraint length_c {length inside {[1:10]};};
  constraint Size_c {Size inside {[1:20]};};
  function new(string name = "traffic_seq");
    super.new(name);
  endfunction
endclass

class config_seq extends uvm_sequence;
  `uvm_object_utils(config_seq)
  rand int config_a;
  constraint config_a_c {config_a inside {[1:5]};};
  function new(string name = "traffic_seq");
    super.new(name);
  endfunction
endclass
```

Figure 2. Source Code of Base sequences used for example

```
//-----Test-----
class my_test extends uvm_test;
  `uvm_component_utils(my_test)
  rand traffic_seq a1;
  rand config_seq b1;
  `uvm_optimization_utils(length,int,a1.length)
  `uvm_optimization_utils(Size,int,a1.Size)
  `uvm_optimization_utils(config_a, int,b1.config_a)

 covergroup cg;
   c1 : coverpoint a1.length { bins a[] = {[1:20]};}
   c2 : coverpoint a1.Size { bins a[] = {[1:10]};}
   c3 : coverpoint b1.config_a  { bins a[] = {[1:5]};}
   c1xc2Xc3 : cross c1,c2,c3;
 endgroup

 function new (string name = "my_test", uvm_component parent = null);
   super.new(name, parent);
   cg  = new;
 endfunction : new

 function post_randomize();
   `ADD_ALREADY_RANDOMIZED(length,a1.length)
   `ADD_ALREADY_RANDOMIZED(Size,a1.Size)
   `ADD_ALREADY_RANDOMIZED(config_a,b1.config_a)
 endfunction

 function pre_randomize();
   int act_cov_c1,act_cov_c2,act_cov_c3;
   int total_cov_c1,total_cov_c2,total_cov_c3;
   cg.c1.get_coverage(act_cov_c1,total_cov_c1);
   cg.c2.get_coverage(act_cov_c2,total_cov_c2);
   cg.c3.get_coverage(act_cov_c3,total_cov_c3);
   `CLEAR_PARAM_QUEUE_COV(length,total_cov_c1)
   `CLEAR_PARAM_QUEUE_COV(Size,total_cov_c2)
   `CLEAR_PARAM_QUEUE_COV(config_a,total_cov_c3)
 endfunction

 function void build_phase (uvm_phase phase);
   super.build_phase(phase);
 endfunction : build_phase

 task run_phase(uvm_phase phase);
   a1 = traffic_seq::type_id::create("a1");
   b1 = config_seq::type_id::create("b1");
   phase.raise_objection(this);
   this.randomize();
   cg.sample();
   phase.drop_objection(this);
 endtask : run_phase
endclass : my_test
```

Figure 2. Source Code of Example Test

A number of re-usable macros have been defined which enable easy utilisation of the solution being discussed. Three user defined macros are used in last example. They help in constraint modeling and defining functions for creating randomized database, updating and clearing the database. The same are discussed below.

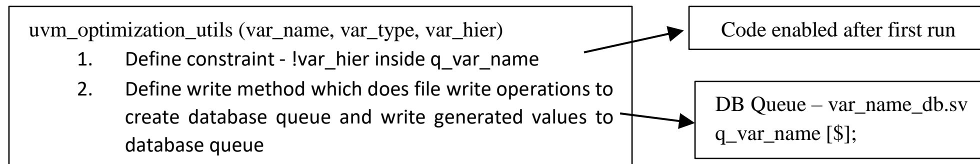

Figure 3. uvm_optimization_utils macro

The above macro has two purpose. First it defines a constraint whose goal is to remove already randomized values from the constraint solver space. This code is to be enabled for subsequent re-runs as the database queue does not exist yet. Second it defines a write method for updating the database. The database consists of a list of files per variable with the already generated values information in the form of queue. These files can be used when re-running test along with define that enables first part of the macro.

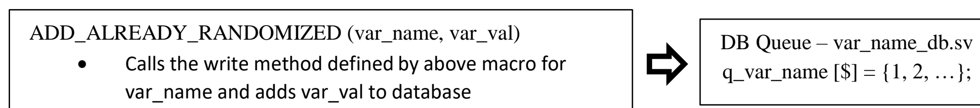

Figure 4. ADD_ALREADY_RANDOMIZED macro

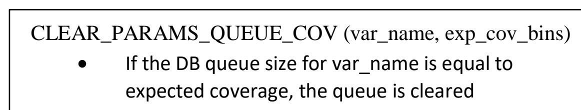

Figure 5. CLEAR_PARAMS_QUEUE_COV macro

The database clear can be controlled using feedback from coverage model . Clearing the database is important so that the modeled constraints do not cause randomization failure when constraint solver space is exhausted for a particular variable while other variables still have more coverage scope.

The above example is not modeled for cross of the variables but meeting individual coverage. The cross coverage example is discussed ahead.

```
class my_test1 extends uvm_test;
  `uvm_component_utils(my_test1)

  `REGISTER_OPTIMIZE_VAR(length,int)
  `REGISTER_OPTIMIZE_VAR(Size,int)
  `REGISTER_OPTIMIZE_VAR(config_a,int)

  `ADD_OPTIMIZE_CROSS_CONSTRAINT_BEGIN(length_Size,length,a1.length)
    `ADD_OPTIMIZE_CROSS_CONSTRAINT_FIELD(Size,a1.Size)
    `ADD_OPTIMIZE_CROSS_CONSTRAINT_FIELD(config_a,b1.config_a)
  `ADD_OPTIMIZE_CROSS_CONSTRAINT_END

  `REGISTER_OPTIMIZATION_RANGES_BEGIN(length,int)
    `ADD_OPTIMIZATION_RANGE(2,5)
    `ADD_OPTIMIZATION_RANGE(6,9)
  `REGISTER_OPTIMIZATION_RANGES_END

  `REGISTER_OPTIMIZATION_RANGES_BEGIN(Size,int)
    `ADD_OPTIMIZATION_RANGE(2,5)
    `ADD_OPTIMIZATION_RANGE(6,15)
    `ADD_OPTIMIZATION_RANGE(16,19)
  `REGISTER_OPTIMIZATION_RANGES_END

 function new (string name = "my_test1", uvm_component parent = null);
   super.new(name, parent);
 endfunction : new
endclass
```

Figure 6. Source code for cross coverage example

The uvm_optimization_utils macro is broken into variable registration and constraint definition for the purpose of targeting unique cross combination of variables. The REGISTER_OPTIMIZE_VAR macro is same as the utils macro except the constraint modeling part is removed. The constraint is structured in a way as shown in above example for flexibility of adding as many variables as the user wants to cross. The pre_randomize (not shown in example) method needs to be changed to input total number of cross bins as queue clear function argument rather than individual targets. All the other code from previous example can be re-used and is excluded in above test for simplicity.

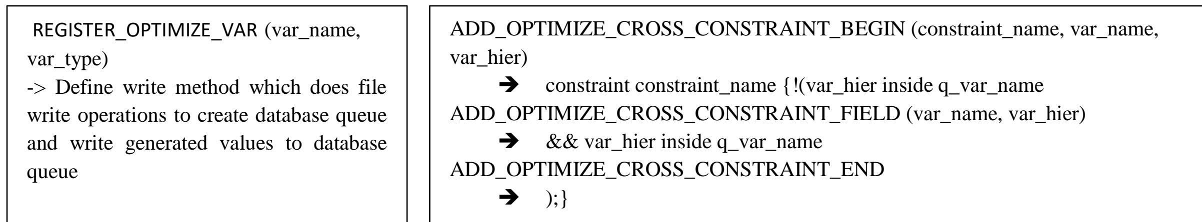

Figure 7. Cross combination example macros

It is very common to have coverage bins modelled as ranges rather than individual values. Having a method to use that information to guide constraint solver is important for an efficient goal driven stimulus solution. This is attained by introducing a new set of macros for registering such ranges and re-modelling the macros discussed thus far to use that information. The register_optimization_ranges* macro help in registering planned coverage ranges for different variables to per variable ranges database. The macros discussed earlier are modified so that if a value is generated in that range, the whole range can be excluded from the constraint solver space. To achieve this, the database queue is implemented as a queue of queues. The constraint definition is updated as per the new queue data type. The write method is updated to check if the value being added to database falls under a registered range. If true, the range is added to database, else the unique value.

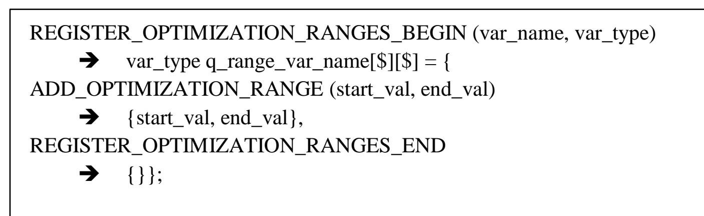

Figure 8. Range registration macros

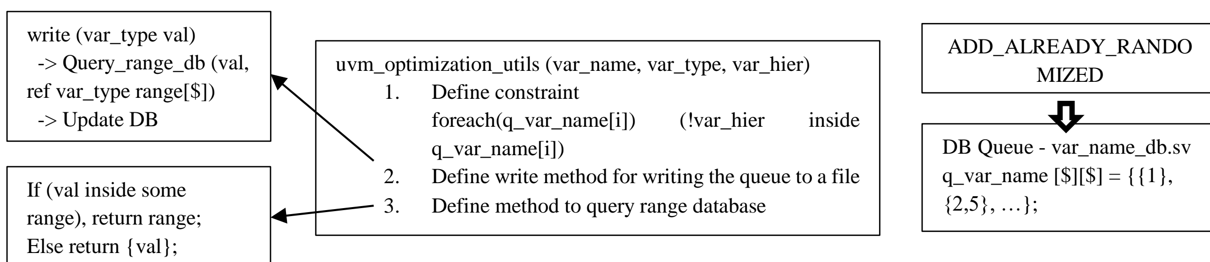

Figure 9. uvm_optimization_utils macro updated

The example discussed above targets use cases where test is regressed multiple times to achieve coverage goals and a single run captures a particular configuration. In case the use case is to generate all possible traffic in same test run with multiple randomize calls rather than regressing the test, the same flow can be slightly modified to create local queue during variable registration and modify write function to update the local queue after each randomization instead of file based operation. A practical use case model is to regress different configurations with separate runs while the variety of data generated in the same run. Implementation for the same can be accomplished by mixing the methods described above and is left up to the user to explore.

An automated flow needs to be created to take advantage of the previously generated values in regressions which enables different iterations to run in parallel without waiting for the previous test run to finish; else it will result in huge regression run time penalty. Most modern regression tools have pre run and post run options. A

blocking script is added to the pre run phase which waits for database creation and size of some variable's queue in the database to reach expected value. The basis for using this flow is that configure phase runs in the beginning of any test case and the more time consuming run phase happens after that. In order to make better utilization of LSF resources, pre run script is run on local machine and only the run script is submitted to LSF. Another dependency is introduced such that pre run script of next iteration is executed only after previous iteration pre run script exits. This prevent too many scripts running on local machine. Although the overall time for regression for same number of runs will be more, the advantages come from being able to reduce the count significantly as redundancy is removed along with negligible coverage analysis effort.

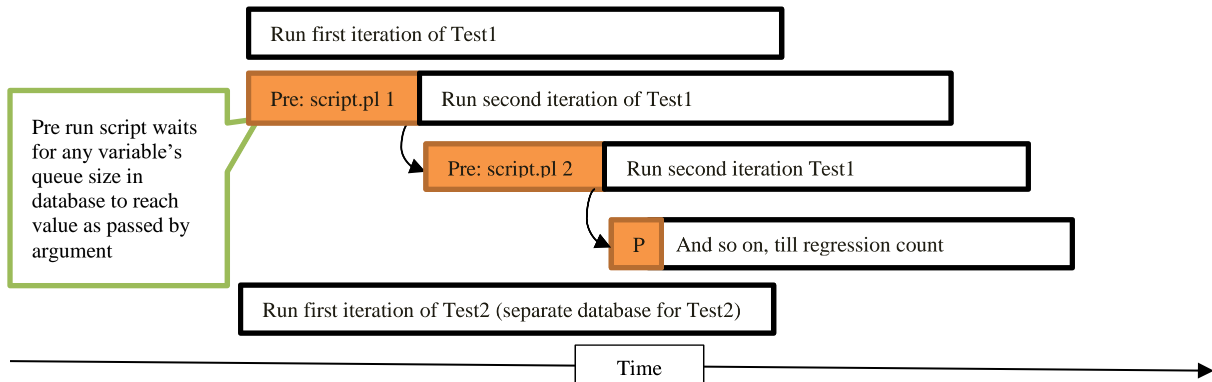

Figure 10. Regression flow

Another key aspect to be taken care of for a sound regression solution is being able to reproduce results for analysis and failure debug. To achieve that, a local copy of already randomized database is kept with each test run. The test re-run can use that copy instead of the global database to reproduce results.

IV. Results

The performance benefits were first studied after implementing the solution in an example test case and the proposed solution was then used for improving the time to reach coverage goals for interconnect verification. The example test case discussed above was used to cover a simple cross of three variables with 1000 possible combinations. Using the traditional approach, the test was run more than 5000 times which resulted in less than 90% coverage. The discussed approach helped in reaching 100% coverage with 1000 reruns in one fourth time given same number of LSF and simulator resources excluding the effort that is needed for improving coverage from 90 to 100% with existing methods.

The solution was then used on a live project for reducing time to coverage closure for an interconnect supporting around 100 masters and 200 slaves. The goal was to test different transaction types, size of data, burst length combinations for all master slave routes supported. Due to the reusability and simplicity of modelled constraints in the aforementioned example, it took a few days to implement the method and reach the coverage goal against the original plan of spending multiple week's effort on achieving the same, given same number of regression and human resources.

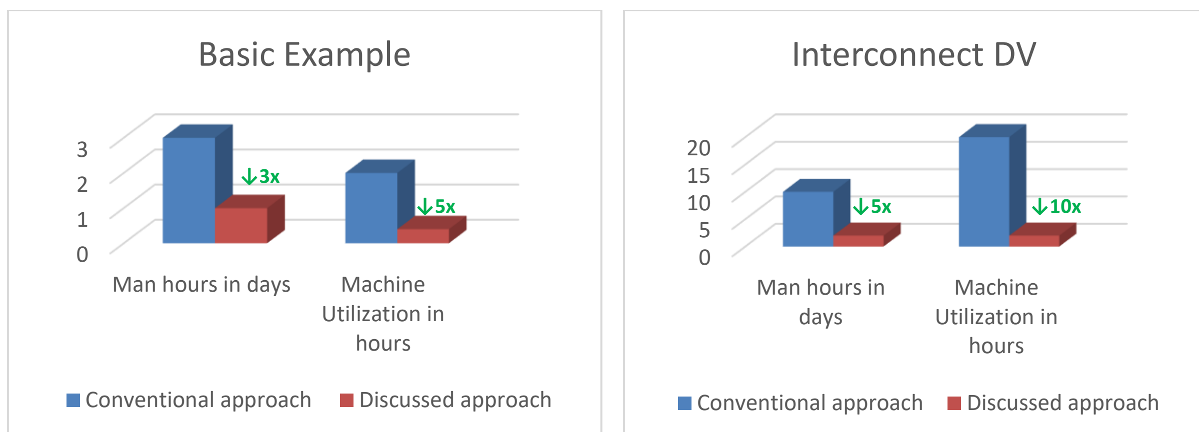

Figure 11. Returns Observed

V. Limitations

This method has some limitations which can result into lower than expected results. For example, the regression run time with limited count could be similar or more than that of bigger count due to synchronization overheads, if the design size and coverage goal is small. In case of bigger designs with complex constraints and coverage models, the additional set of constraints can increase constraint solving time and modelling effort resulting in overall diminishing results.

The central idea is to generate unique combination of values which prevents its use for creating scenarios where repeating and mixing similar packets is important. This makes it difficult to use for transition cover points. One way to overcome this limitation could be to define same variable two times with similar set of constraints and register cross combination of these variables. An additional constraint which limits value of newly added variable to last value of actual variable can be added. This might be sufficient for some use cases, but the possibility of discovering corner cases with more interesting transitions is not guaranteed.

It's possible that the randomization calls occur late in simulation environment and cannot be modeled to happen earlier due to some reasons. The synchronization process overhead between different iterations will limit the advantages obtained from this method in such cases. Also there might be redundant randomization calls in test case which might not necessarily lead to coverage hit. This can produce unexpected results. These limitations can sometimes result in deployment of the method more challenging than intended.

VI. Conclusion

To summarize, the preliminary results prove the strength and promise that the approach holds over conventional methods. This paper introduces a possible solution using existing tool and SV language features that can help avoid redundant use of engineering resources resulting in faster automated coverage closure and significant cost savings. The simplicity of the basic principal used in the solution makes it disposable for targeting completely or partially coverage goals in a variety of SV-UVM based TB environments. Results and benefits from initial example and one real application were discussed and compared against common practices.

VII. Future Scope

To extend and improve the current work, it needs to be deployed for more complex use cases and results analyzed. It would be interesting to see the kind of designs where it proves most useful and innovative ways in which some of the limitations can be tackled. One key area to look into is to develop automations and a standard process which can make adoption and evaluation of this method faster for different use cases. Although the fundamental constraints modelled are re-usable, it requires manual effort in terms of modelling their use for specific goals. Further research and effort needs to be expended for automating the modelling process as per desired coverage. The gains observed in terms of time saved reduce sharply if the randomization call happens late in simulation because of synchronization overhead. In case, moving randomization call earlier is not possible for some reason, simulation checkpoint save before randomization and rerun from checkpoint approach can be adopted to overcome this hindrance. Transition coverage support can be added and tested.

Acknowledgement

I thank my employer SSIR for sponsoring my attendance at DVCON to share this paper. I would also like to thank my team members who adopted the methodology, shared the results and gave me valuable feedback.

References

[1] "IEEE Standard for SystemVerilog—Unified Hardware Design, Specification, and Verification Language", IEEE standard 1800-2012, IEEE-SA Standards Board, New York, 2012.

[2] Chris Spear, System Verilog for Verification -A Guide to Learning the Testbench Language Features, New York: Springer, 2012.

[3] Eldon Nelson, Improving Constrained Random Testing by Achieving Simulation Verification Goals through Objective Functions, Rewinding and Dynamic Seed Manipulation [Online]. Available: https://dvcon.org/sites/dvcon.org/files/files/2017/07_1.pdf

[4] Coverage Closure – Is it a "Game of Dice" or "Top 10 Tests" or "Automated Closure"? [Online]. Available: https://dvcon-india.org/sites/dvcon-india.org/files/archive/2015/proceedings/111_Coverage_Closure.pdf

[5] Cadence, "Xcelium version 18.03" [Online]. Available: http://www.cadence.com

[6] Synopsys, "VCS version VN-2017.12" [Online]. Available: http://www.synopsys.com