Combining Simulation with Formal Techniques to Reduce the Overall Verification Cycle

Aneet Agarwal
Texas Instruments
Bangalore, India
+91-9900210780
aneet@ti.com

Gaurav Gupta
Synopsys (India)
Bangalore, India
+91-9972033995
gauravg@synopsys.com

ABSTRACT

With design complexity growing by the day, the need for verification technologies that can complement simulation based verification is also gaining momentum. Formal verification has clearly emerged as one of the strong candidates. In a typical simulation based verification cycle, the number of bugs reported grows exponentially in the beginning, but this number shrinks rapidly thereafter in the cycle and what remains is a few difficult to find corner case issues.

Obviously, there is a need to start verification early in the design cycle. The additional requirement is to uncover the corner case bugs earlier to reduce the overall verification cycle. Overall, a simulation environment also takes a considerable amount of time to setup. Early adoption of formal verification in the design cycle enables quick RTL bring-up before the simulation environment is ready. It also exposes hard-to-find corner-case bugs that may not be uncovered in simulation. On the downside, it suffers from capacity issues as formal analysis is exhaustive. However, hybrid formal verification provides a way to counter capacity issues by combining simulation with static formal verification to explore deeper design states. It greatly enhances the capability to find bugs hidden deep in the design space. Hybrid formal verification has also evolved and it now allows combining non-synthesizable simulation environment elements while verifying a design formally.

Texas Instruments (TI) image processing sub-chips, for the imaging sub-system are used in application processors for mobile handsets. These are algorithmic IPs, and also contain lot of control logic (e.g. complex arbitration logic) that makes them good candidates to be exhaustively verified through formal verification. Availability of pre-verified assertion IP packages for industry standard-interfaces like OCP[1] enables a formal verification environment to be setup quickly.

This paper covers the TI experience of employing formal verification (Model Checking) on OCP2.2 compliant designs and the methodology adopted. Furthermore, it covers the exploration of hybrid technologies to re-use non-synthesizable simulation environment elements and attempts closing verification loopholes by using different aspects like code coverage. This paper would also cover the methodology adopted to find dead code in RTL blocks generated by high-level synthesis tools (e.g. C-to-RTL generation tools). Finding dead code was important to achieve targeted code coverage. Formal technology allows finding out the unreachable lines of code with 'guarantees' that it cannot be hit in simulation. It also enables us to feed back the results obtained through formal verification and exclude unreachable lines of code in simulation to compute the required code coverage number.

Keywords

Magellan, AEP, AIP, DW, VIP, RAL, OCP, VCS, HLS, VMM, TI

1. INTRODUCTION

Coverage closure is an area where verification engineers usually spend lot of their time writing additional tests to target uncovered coverage points despite a feeling that no combination of vectors would ever exercise it. Usually a conclusion is met through visual code inspection, but it proves to be a lengthy and expensive process. We made use of Formal Analysis to reduce our time to conclude that a particular coverage target (structural coverage: line, condition etc.) could not be covered. Formal Analysis can prove a target as reachable or unreachable. This approach helped us to quickly find dead code in our design block (an algorithmic IP), RTL of which has been derived from a C-RTL conversion (HLS[2]) tool.

Writing a constraints model for performing formal verification is perhaps the most critical phase of the whole formal verification cycle. Validity of our results highly depends on the way constraints have been written at input of our design block. An under constrained environment may trigger false failures and could allow illegal stimulus to be driven at input of our design block during simulation phase (as we are using a hybrid tool Magellan[3]). On the other hand, an over constrained environment may lead to false proofs which may prove to be fatal as it could hide bugs in the design block. We explored ways to detect over constraining and found out that one way to detect over constraining is to use "AEP code coverage" feature of Magellan. Magellan can automatically extract structural coverage metrics (line, statement, etc) from the design's source code. Magellan then can prove whether a corresponding line is reachable or not using its formal engines. For reachable lines, it would generate a counterexample by running simulation using inbuilt VCS[4] simulator. If it finds unintended unreachable lines, it indicates that constraints model is over constrained.

Reuse of passive testbench components such as monitors and scoreboard with Formal Verification Environment made an interesting combination of both the worlds (Simulation & Formal) by

[1] OCP stands for Open Core Protocol

[2] HLS stands for High Level Synthesis
[3] Magellan is Synopsys Hybrid Formal Model Checking Tool
[4] VCS is Synopsys Mixed Language Simulator

validating Formal constraint model against passive testbench components[5]. This method allows exhaustive constraint random simulation, generated during random simulation phase (of Hybrid Formal Tool) and during replay of Formal traces into the Simulator to be passed to Monitors and Scoreboard. It should be noted that Active testbench components[6] have to be shut down. Non-synthesizable "Constraint Model" used in simulation that typically had been coded using HVLs like "System Verilog testbench" also could not be used.

Following sections outline our experience using Formal techniques in our verification flow and explore ways to bring together Formal techniques and Simulation during Verification process.

2. DEPLOYING FORMAL TECHNIQUES TO IDENTIFY DEAD CODE

This section discusses about the challenges faced during the verification closure of an algorithmic IP design, the RTL of which is extracted through C-RTL conversion (HLS) tool. The Structural (code) coverage closure was very difficult and painful job as the design contained a lot of dead RTL code/logic introduced by the HLS tool. Therefore we decided to use Formal Analysis to quickly identify dead code as there was no reason to measure coverage on unreachable lines.

Setting up Formal Verification environment was very easy as it didn't require any constraint or testbench. RTL is directly fed to a Formal Tool, Magellan, without giving any constraints and Magellan's line coverage AEP[7] was enabled. Figure-1 shows a snapshot of the simple setup file required to setup the Formal Verification environment in Magellan.

```
## Design

add_design_info -vlogan {-work WORK +v2k <*.v> ...}
add_design_info -vhdlan { <*.vhd> ... }
set_design_info -topModule SIMCOP

## Environment

add_env_port -name pi_clk    -clock 40
add_env_port -name pi_clk_en -clock 80 -waveform {20
60}
add_env_port -name pi_rst_n  -reset 0
add_env_port -name pi_rst_n  -constant 1

add_env_port -name PI_SIMCOP_DFT_LCG_CTRL_EN_N    -
constant 7'b0
add_env_port -name PI_SIMCOP_DFT_LCG_TE           -
constant 7'b0

## AEP Module(s)
set_aep_info -line -maxGoals 25000
```

Figure 1. Magellan Project File

[5] Components that doesn't drive the DUT like Scoreboard
[6] Components that drives the DUT like Generator
[7] AEP stands for Automatic Extracted Properties

Magellan automatically extracted line coverage points based on the structural analysis of the design. Magellan's Formal engines then discovered certain lines that were unreachable. The unreachable coverage points represented the dead or redundant code of the design, which could not be covered with any combination of stimulus at all (no constraint was applied on the inputs). The unreachable coverage report is further used to prepare an exclusion file, which is a key input to guide simulator to exclude these coverage targets from simulation coverage evaluation. Table-1 shows the comparison of coverage statistics before and after removing dead code logic.

Table 1. Coverage Statistics

Module	Module Version	Line Coverage (%)	Comments
SIMCOP	SIMCOP--0.8	53	Initial Coverage
SIMCOP	SIMCOP--0.9	73	After Dead Code Removal

Using Formal Techniques, we were quickly able to identify the unreachable lines of code and it helped us to achieve targeted coverage number for our design.

3. DETECTING OVER-CONSTRAINED FORMAL CONSTRAINT MODEL USING AEP COVERAGE

It is important to detect over constraint environment as it could hide bugs. Detecting over-constraining is not easy as in normal property checking flow virtually no signs are given to the user to identify over constraining. Moreover over constraint environment may lead to proofs for those properties, which otherwise might be falsified in the correct environment. We realized that it is important for us to make sure that our constraint model is correct and we decided to make use of "AEP Code Coverage" analysis with Magellan to validate the same. Magellan's Formal engines can prove whether a line or condition is reachable or not. If it finds unintended unreachable lines, it indicates that constraints model might be over constrained. We were able to deploy this strategy successfully across our designs to successfully validate our constraint models. Figure-2 shows an example how we were able to detect over constrained Formal Environment using this approach.

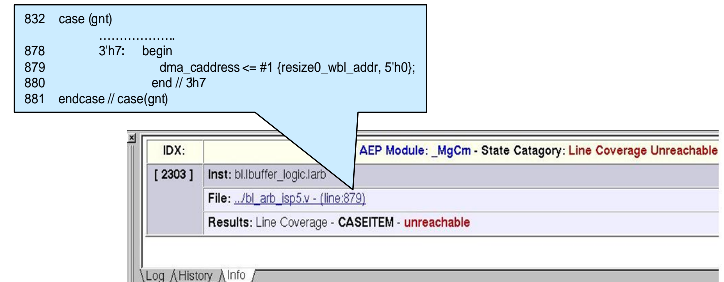

Figure 2. Magellan GUI, Showing Unreachable Line Information

As shown in Figure-2, Magellan reported line no. 879 as unreachable. We back traced the cause and found that the signal “gnt” never reached the value 7 because the request corresponding to that grant was never generated due to the following constraint (Figure-3).

```
Property mtc_cons0b ;
    disable iff (!vpss_rst_n)
    @(posedge vpss_clk) ( !(|resize0_sdrc_req));
endproperty
```

Figure 3. Input Constraint

Further analysis showed that this constraint was added to mask an earlier bug which no longer existed, but the constraint was not removed as it should have been.

4. VALIDATING FORMAL CONSTRAINTS MODEL AGAINST PASSIVE TESTBENCH’S COMPONENTS

As discussed earlier, a correct formal constraint model is essential for effective verification. Availability of Simulation based verification environment provides a novel way to reuse some of the passive testbench components to validate the formal constraint model. It would also facilitate to identify discrepancies between reference model and DUT. For data intensive designs, this approach would provide a way to monitor data transfer behavior which otherwise is difficult to do in traditional a Formal Environment.

In one of our recent design, we reused monitors and scoreboard developed for simulation testbench in our formal verification environment. Simulation testbench is being developed using VMM[8] methodology and has an integrated Synopsys DW[9] OCP Slave VIP and RAL[10] package. Reference Model integrated into scoreboard is a non cycle accurate model mimicking complex DUT behavior. We explored this approach with a set of objectives in our mind:

- Validate monitors and scoreboard against Formal verification environment.
- Validate and reuse non-formal compliant checks.
- A more regressed testing for data intensive path with less turn - around time.
- Portable Valid Coverage data across Simulation and Formal Verification environment.

4.1 Design under Verification

Design under verification is a multi-channel DMA. One of its ports is an OCP Master with a tag management unit. On the other side there are five DMA channels. Four of the channels can read Data from the DMA buffers and one channel can write into it. DMA requests data from the OCP slave port for read intensive channels and sends data to OCP slave for write intensive channels. This design is complex and hugely data intensive (approximately half a million gates).

4.2 Steps to Integrate Testbench Components with Formal Verification Environments

1. We setup a Formal Verification environment for DUT using Synopsys OCP AIP[11] and some custom assertions.
2. Created a Verilog file (tb_inst_inline.v) that contains non synthesizable testbench code. Following is a snippet for the same (Figure-4).

```
`include "vmm.sv"
`include "dma_top.sv"
`include "ral_env.svh"

//vip interface instantiation
  `add_ocp_vip_if(dma)

 module_env env;

initial begin
  env = new(channel_intf);
  env.run();
end

assign    dma_if.MReset_n  = channel_if.RST_N;
assign    dma_if.SReset_n  = 1;
assign    dma_if.Clk       = top.CLK;
 .
 .
 .
```

Figure 4. Snippet From tb_inst_inline.v

3. Added testbench components in Magellan Project File. Also added “tb_inst_inline.v” file with *add_env_inline* command. a*dd_env_inline* command prevents the Magellan formal engines from seeing the testbench code that was encapsulated within Verilog file *tb_inst_inline.v*. Figure-5 shows a snippet of a project file.

```
set_design_info –vcs { … apps/synopsys/designware/vip
                                  /ocp_vrt/dw/vip/ocp_vrt/1.50a
                                  /ocp_slave_svt/vera/src
                               … }
add_env_inline –testbench –file tb_inst_inline.v
```

Figure 5. Project File with Testbench Components

4. Disconnected active testbench components (like RAL model, OCP slave VIP[12] generator) that drive the DUT and disabled Driver BFMs, since input constraints are already taken care by Formal environment. We decided not to completely remove these components from our environment as it would have required a lot of effort to

---

[8] VMM stands for Verification Methodology Manual
[9] DW stands for Designware
[10] RAL stands for Register Abstraction Layer
[11] AIP stands for Assertion IPs
[12] VIP stands for Verification IPs

extract only the monitors and scoreboard from the testbench. Instead, we decided to disable them in our testbench environment which was quick and effective as these components were lying dormant now. Figure-6 shows a snippet of disabling these components within our testbench.

```
// Commenting RAL backdoor call as DUT has been configured from
// Formal Verification Environment
  //update_ral_model (status,this.ral.default_path);

// Commenting OCP Slave VIP model as OCP AIP Slave
// assertions are configured as constraints in Formal Verification
// Environment
    // svt_ocp_slave                    slave;
       .
       .
    // slave.start_xactor();

  // DW OCP Monitor
      svt_ocp_monitor                   mon;
       .
       .
      mon.start_xactor();
```

Figure 6. Disabling Active Testbench Components

5. The next step was to send data collected from DUT interfaces to testbench monitors. In the testbench environment, these interfaces were directly connected to DUT ports. As DUT is instantiated in Formal Verification Environment now, we needed a way to pass data from DUT ports to monitors. For doing so, we made a continuous assignment between DUT ports to Monitor's ports. Figure-7 shows an example.

```
assign dma_if.MCmd   = top.MCmd;
assign dma_if.MAddr  = top.MAddr;
```

Figure 7. Connection to Monitor's Ports

6. We then enabled Coverage AEPs (line, condition, FSM, and toggle) because stimulus would be applied to DUT interface only when Simulator is active and testbench components would be able to see those stimuli. Enabling coverage AEPs would make sure that a good set of stimuli was applied by simulator as it would exercise counterexamples corresponding to each reachable coverage target. This would generate high-coverage stimuli for the design.

7. Configured the DUT.

8. Started testbench environment (env.run).

9. Ran formal verification environment using Magellan.

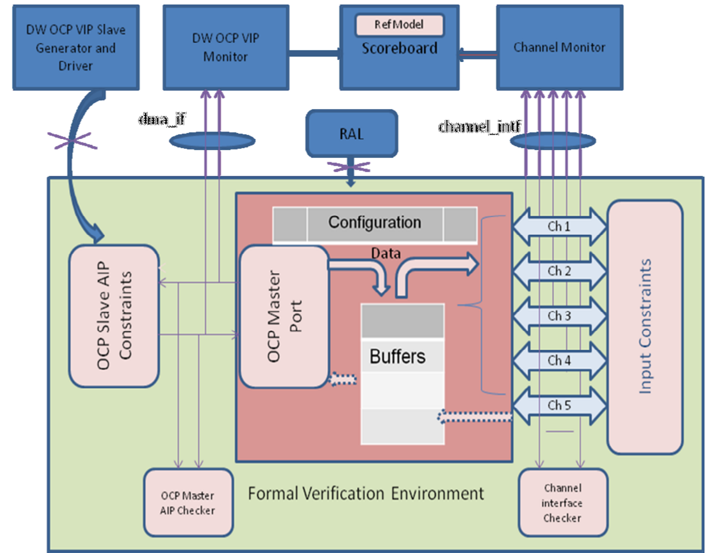

Figure 8. Environment Setup

Figure-8 illustrates our complete Verification environment. At this stage, our setup was ready and we started running our Formal Verification environment

4.3 Results

We seamlessly integrated simulation environment components (monitors and scoreboard) into Formal Verification Environment with all the comparison and checks being performed as in a simulation environment. Figure-9 displays a snapshot of a non-formal compliant check executing in an OCP VIP Monitor along with FSM coverage AEPs being exercised.

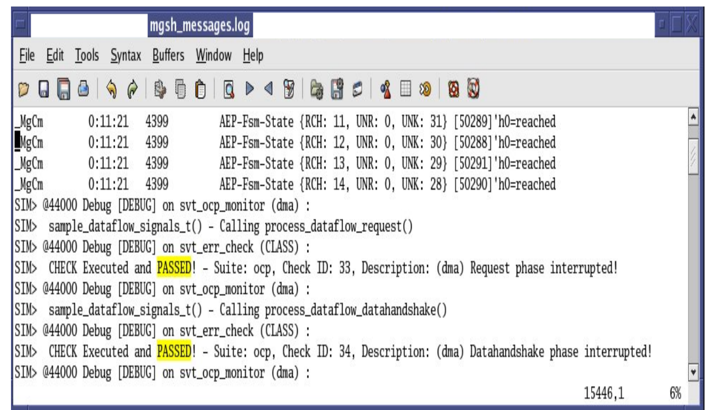

Figure 9. Magellan Messages Log File

It took us approximately a week to setup our initial environment, though further refinements to the environment are going on as per the DUT requirement. Our initial runs have already shown us good promise as we were already able to identify a mismatch with reference model for the OCP Master Commands.

5. CONCLUSIONS

Formal verification provides huge value addition in discovering bugs earlier at block level before integrating it to the subsystem. Additionally, corner-case bugs can be found by means of Guided Coverage Convergence. A chosen coverage metric, such as "cover" coverage or formal "state reachability" intelligently drives the built-in simulator to exercise all the reachable coverage targets, while the reporting of unreachable coverage targets guarantees the conclusiveness of the verification results for a given block-level verification task.

Formal Verification also can easily be deployed for “RTL bring up” and AEPs could be of great help for doing that as the designer can perform sanity test of his/her design without worrying about testbench or checks.

Dead Code analysis provides a good way to quickly discover dead code especially for the RTL generated by using HLS tool since they are prone to generating a lot of Dead code.

Finally, reusing passive testbench components with Formal Verification Environment opens up a whole new dimension to verification. It creates umpteen avenues to validate traces generated during Formal verification through Monitors and Scoreboard of simulation environment, to identify discrepancies in the two environments that might be hiding errors. Comparing stimulus generated during Formal verification in testbench’s scoreboard helped to validate that there is no difference in the constraint model that we used in testbench (Written in SVTB[13]) and the constraint model that we used in Formal verification environment. (Written as a set of Assertions).

6. ACKNOWLEDGMENTS

Our sincere thanks go to Ashish Chandra of Texas Instruments for providing details about his Simulation Verification Environment and helping us to derive strategy for integrating passive simulation components into Formal Verification Environment.

7. REFERENCES

[1] Jayanta Bhadra, Magdy S. Abadir, Li-C. Wang, Sandip Ray, "A Survey of Hybrid Techniques for Functional Verification," IEEE Design and Test of Computers, vol. 24, no. 2, pp. 112-122, June 2007.

[2] http://www.synopsys.com/cgi-bin/fv/webinar/reg1.cgi

[3] https://event.on24.com/eventRegistration/EventLobbyServlet?target=registration.jsp&eventid=174132&sessionid=1&key=99DAA6FD4F58B8097AEF90654999A77B&sourcepage=register

[13] SVTB means System Verilog Testbench