From Specification to Complete RTL Verification Using Formal Methodology

Suckheui Chung

AMD
Boxborough, MA

www.amd.com

ABSTRACT

Formal verification is challenging and its results vary due to many factors. However, in many applications the results justify the investment. This paper presents one such approach that led to extensive formal verification on an IP thereby achieving 100% functional coverage, a goal nearly impossible to reach through traditional functional simulation, random or direct. This approach involved writing behavioral models using System Verilog Assertions on a design that is highly complex but has well defined end to end points. This daunting complexity was simplified and resolved using formal techniques. This paper discusses: trade-offs, mapping from specification to behavioral modeling with SVA and debug to meet the coverage goal. This approach found some bugs which could not be found easily with traditional functional simulations.

Table of Contents

Table of Figures

Table of Tables

1. Introduction

As RTL becomes more complex, the effort to verify its functionality is increasing dramatically. The verification community has explored many ways to achieve higher percentage of functional coverage. Direct or random simulations have been mainly used. SVA (System Verilog Assertions) have been used to supplement the simulation based verification. Recently, more interest has been developed in formal verification.

Formal verification is promising, but challenging too. There have been some success stories, but it is still regarded as a secondary tool.

This paper shows that formal verification can be used as the main functional verification methodology and achieve 100% functional coverage. However, two things need to be emphasized, before discussing how formal verification methodology was applied.

First, the formal verification on this IP (Intellectual Property) achieves 100% functionality coverage given that it verifies the "Specification" against RTL implementation. If the spec is modified, the formal verification needs to be updated too. It is not 100% covering the design "intention". A verification fail hole(s) can occur if the spec is not defined for that specific fail case(s).

Second, SVA is used to make behavioral models of the spec. It is not used for "assertion based verification methodology". This is shown in Figure 1.

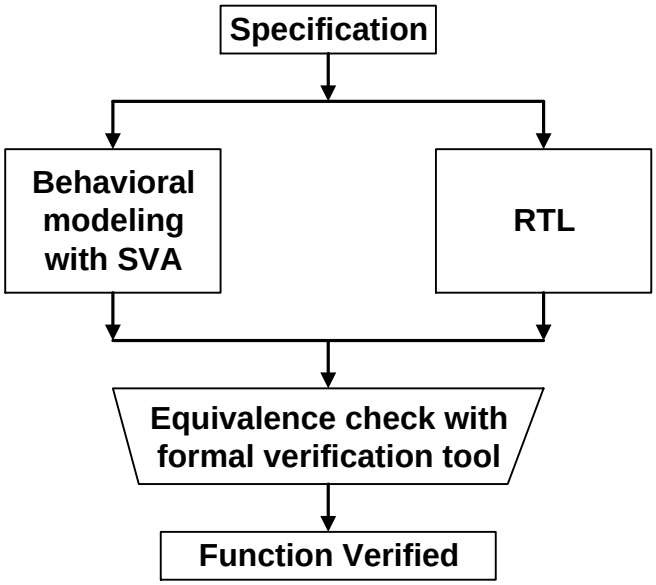

Figure 1: Formal Verification Flow Diagram

2. Verification Challenge

MCA (Machine Check Architecture) is a cross functional IP over any applicable processors. Its purpose is to collect the processor's error information and generate relevant request for OS to react accordingly.

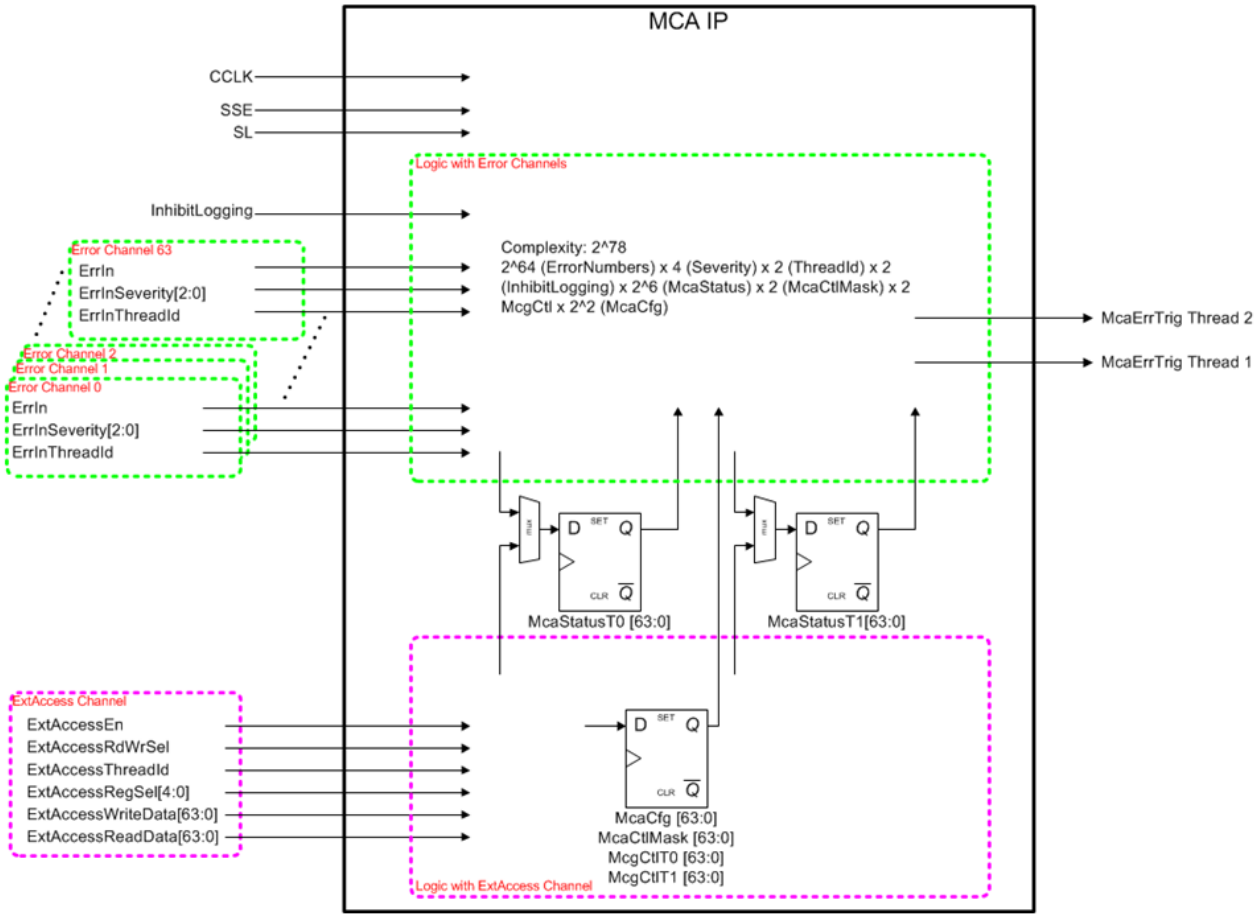

Figure 2: IP block diagram

It has two main input paths, error and external access (extaccess) channels. The error channel receives error info from the outside IP, and extaccess is the control connection to IP. Every time an error occurs, IP puts the relevant info in registers inside IP accordingly. Based on the error type, it may generate relevant interrupts via McaErrTrig.

The main verification challenge of this IP lies in its high complexity and combination of error inputs, multiple register bits, and associated logic that make difficult to verify its functionality of outputs. One approach to solve this is running exhaustive simulations with random errors until high functional coverage is achieved as done traditionally. But, the complexity level could go to 2^{78} combinations as shown in Figure 2. Therefore, this can be an expensive solution.

The advantage of this IP design is that it has well defined end-to-end (source-to-destination) points. All error inputs (source) contribute to internal registers and interrupt outputs (destination). Logical relations between "error inputs and registers' outputs" and "IP outputs" are clearly defined in the spec. This means that formal verification can be used effectively from one

end to another straightforwardly, since the logical relation is the same as in the "behavioral model". Formal verification can check if "input to output of RTL" is identical to the "behavioral model of the spec".

There are three main verification paths in IP:

1. From error inputs to registers
2. From error inputs and registers outputs to IP's outputs
3. From extaccess inputs to registers

This paper shows how to implement "behavioral models of the spec" using SVA (System Verilog Assertion) as language step-by-step. SVA is used not for RTL functional assertions, but for the language to make models of the spec.

3. How to implement formal verification with SVA and debug

All implementations were done with Synopsys vc-static [2]. But, main techniques are applicable for any available tools.

3.1 Reset

The first step for formal verification is to define the reset. IP has two inputs, SSE and SL. The two inputs generate three different modes as shown in Table 1.

Mode	SSE	SL
Cold reset (reset for all registers)	1	1
Warm reset (reset for some registers)	1	0
Functional	0	0
Illegal	0	1

Table 1: Reset and functional mode with SSE and SL

The reset is defined as follows:

```
create_clock CCLK -period 100
sim_force SSE -apply 1'b1
sim_force SL  -apply 1'b1
sim_run 2
sim_save_reset
fvassume nc -expr {{SSE, SL} != 2'b01}
```

This ensures that the cold reset is asserted for 2 cycles before formal verification, and the illegal state is excluded from input combinations. The proper reset condition before formal verification is important. Otherwise, formal verification can produce too many mismatches to debug.

Some constraints can be added to reduce the complexity. For example, the condition below can reduce the effort to make "behavioral models using SVA". These 2 inputs can always be high during normal function.

```
set_constant TstCcgEn -value 1'b1
set_constant CoarseEn -value 1'b1
```

Fewer constraints are recommended for higher coverage. For this IP formal verification, no constraints were used to reach 100% functional coverage.

3.2 Behavioral modeling using SVA assertion and coverage

Here is an example of how SVA assertion and coverage are used [1]. The spec is as follows:

```
After cold reset, McaDefAddr register should have all 0's for bit[63:0]
```

First, a property can be defined as follows:

```
property hardreset (logic SSE, logic coldreset, logic [63:0] regs);
  @(posedge clk) disable iff (SSE)  $fell(coldreset) |-> ##0 (regs[63:0] == 0);
endproperty
```

The hardreset property checks the negedge of coldreset, and then checks if the register bit[63:0] are all 0's. This property for assertion and coverage are written as follows:

```
aDef: assert property (hardreset(SSE, coldreset, McaDefAddr)) else $fatal ($stime,,,"%m assert FAIL");
cDef: cover  property (hardreset(SSE, coldreset, McaDefAddr)) $display ("%m cover PASS @%0d", $time);
```

The hardreset property is a "behavioral model using SVA". aDef: assert lets the formal verification tool to check the hardreset property matches RTL. This can be used for simulation based verification as well. The main difference is that simulation based verification needs the exact stimulus written by users to cover this assert case, but the formal verification tool checks all possible cases for this assertion automatically. aDef:cover checks if the case is covered.

3.3 Reset status check

Registers' reset status can be verified the same way as shown in 3.2. The case for warm reset is shown as follows:

```
property Warmreset_diff (logic coldreset, logic warmreset, logic [63:0] regs, logic [63:0] regs_warmreset);
  @(posedge clk) disable iff (coldreset)  $rose(warmreset) |-> ##1 (regs == regs_warmreset);
endproperty

assign McaCfg_warmreset  = {32'h0000_0002, 28'h0000_003, 1'b0, 1'b1, 1'b0, 1'b1};

aWarmreset_diff_McaCfg: assert property (Warmreset_diff(coldreset, warmreset, McaCfg, McaCfg_warmreset)) else $fatal ($stime,,,"%m assert FAIL");
cWarmreset_diff_McaCfg: cover property (Warmreset_diff(coldreset, warmreset, McaCfg, McaCfg_warmreset )) $display ("%m cover PASS @%0d", $time);
```

In order to do the same verification in simulation, users need to write direct test vectors to read back registers via the extaccess channel, and match the read values via checker.

3.4 Functional check on paths from error inputs to register output

The spec is defined as follows:

The status register logs a transparent error only when the error is transparent and the configuration bit[33] is high.

To pinpoint this check, the spec can be interpreted this way.

The status register is stable when the error is transparent and the configuration bit[33] is low.

The reason for this is that if the spec is interpreted as it is, multiple bits need to be checked. A simpler property can be written as follows:

```
property Status_stable_disable_cfg (logic allreset, logic error_valid, logic cfg, logic [63:0] regs);
  @(posedge clk) disable iff (allreset) (error_valid & ~cfg) |-> ##0 $stable(regs);
endproperty

aStatus_stable_no_trans_T0: assert property (Status_stable_disable_cfg(allreset, error_trans_T0,
McaCfg[33], McaStatusT0)) else $fatal ($stime,,,"%m assert FAIL");

cStatus_stable_no_trans_T0: cover  property (Status_stable_disable_cfg(allreset, error_trans_T0,
McaCfg[33], McaStatusT0)) $display ("%m cover PASS @%0d", $time);
```

Allreset is cold or warm reset. However, it would result in failure in formal verification, because the reset or extaccess channel can be accessed while the error channel is being used. The spec is defined under the assumption that (1) reset was asserted before IP under the functional mode, and (2) the extaccess channel does not alter the value of the configuration register in the middle of the active error channel. Therefore, two more conditions are added that use additional block_reset and block_ext signals as follows:

```
property Status_stable_disable_cfg (logic allreset, logic block_reset, logic block_ext, logic
error_valid, logic cfg, logic [63:0] regs);
  @(posedge clk) disable iff (allreset) (block_reset & block_ext & error_valid & ~cfg) |-> ##0
$stable(regs);
endproperty

aStatus_stable_no_trans_T0: assert property (Status_stable_disable_cfg(allreset, block_reset,
block_ext, error_trans_T0, McaCfg[33], McaStatusT0)) else $fatal ($stime,,,"%m assert FAIL");

cStatus_stable_no_trans_T0: cover  property (Status_stable_disable_cfg(allreset, block_reset,
block_ext, error_trans_T0, McaCfg[33], McaStatusT0)) $display ("%m cover PASS @%0d",
$time);
```

Block_reset is modeled as follows:

```
always @(posedge clk) begin
  block_reset_pos <= ~SSE && ~($past(SSE)) && ~($past(SSE,2)) && ~SL  && ~($past(SL)) && ~($past(SL,2));
end

always @(negedge clk) begin
  block_reset_neg <= ~SSE && ~($past(SSE)) && ~($past(SSE,2)) && ~SL && ~($past(SL)) && ~($past(SL,2));
end

assign block_reset = block_reset_pos & block_reset_neg;
```

This forces formal verification not to check reset during aStatus_stable_no_trans_T0 is being checked. The extaccess channel could be disabled as follows:

```
always @(posedge clk) begin
  block_ext <= ~ExtAccessEn && ~($past(ExtAccessEn)) && ~($past(ExtAccessEn,2)) && ~($past(ExtAccessEn,3));
end
```

ExtAccessEn is the enable signal input for the extaccess channel.

With these two restrictions, the spec is successfully verified. However, how can users debug for each case?

The GUI windows captured from the formal verification tool below show that the case block_ext is not defined in the property. The formal tool detects there is a “falsified” case, and generates the red color error via GUI. With several clicks, the formal verification tool opens Verdi with the text window describing the falsified case, and also showing the waveform. Afterwards, it is the same as with the normal debugging process of RTL. This is useful to debug a property that does not perfectly cover all cases. Users may need to repeat this debug, and can complete property and reach 100% functional coverage.

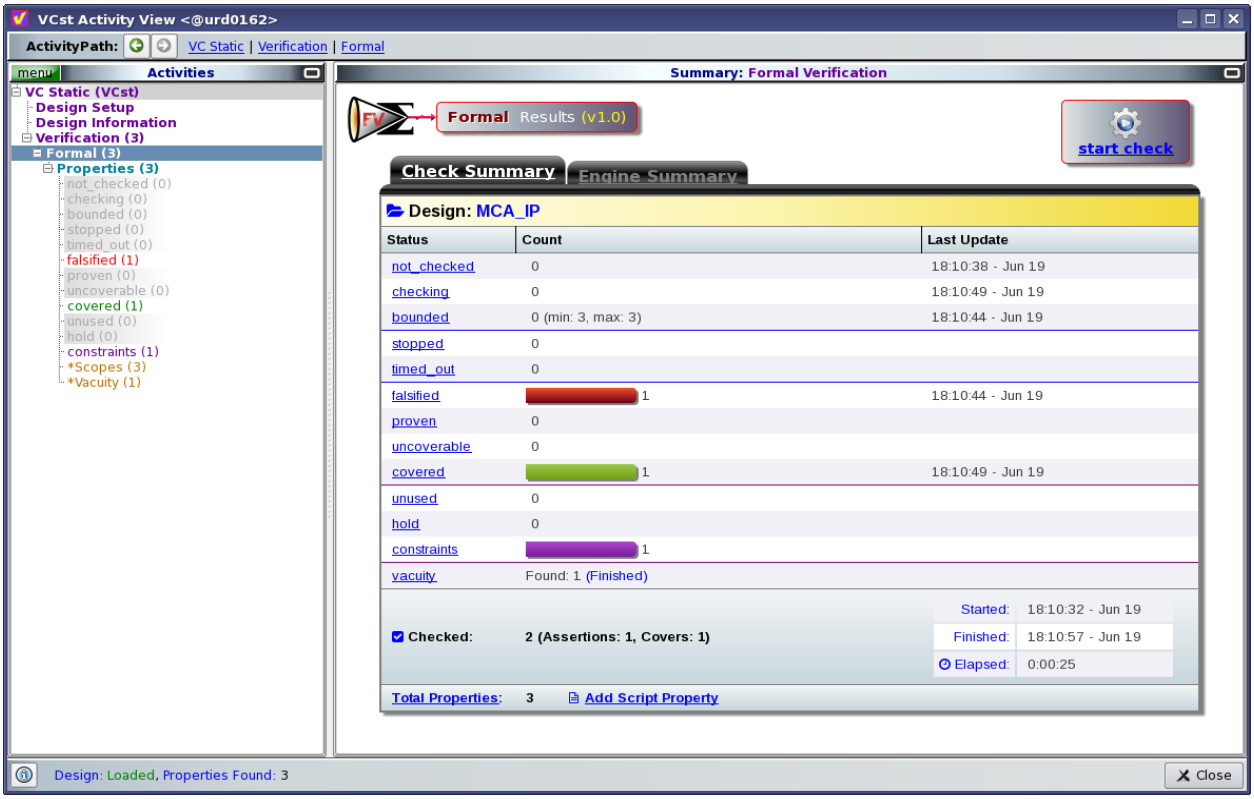

Figure 3: Falsified screen shot 1

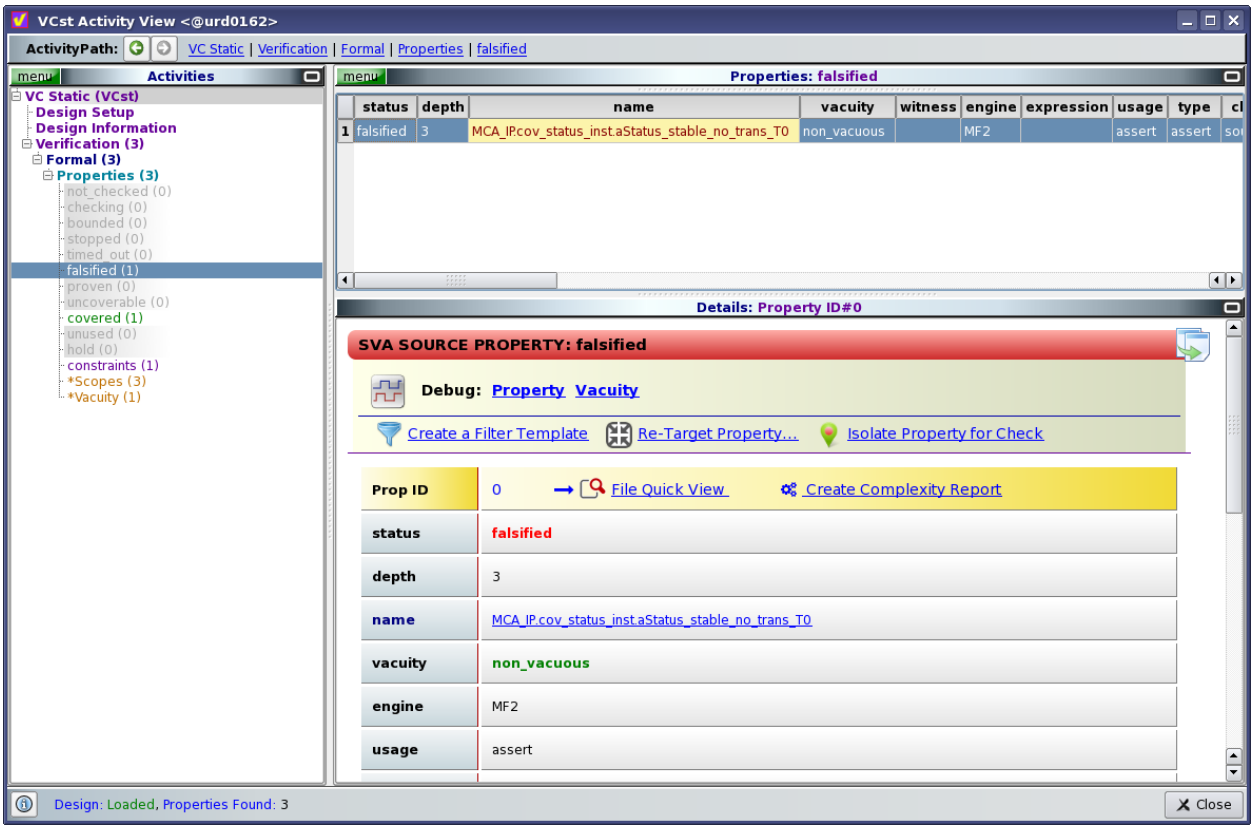

Figure 4: Falsified screen shot 2

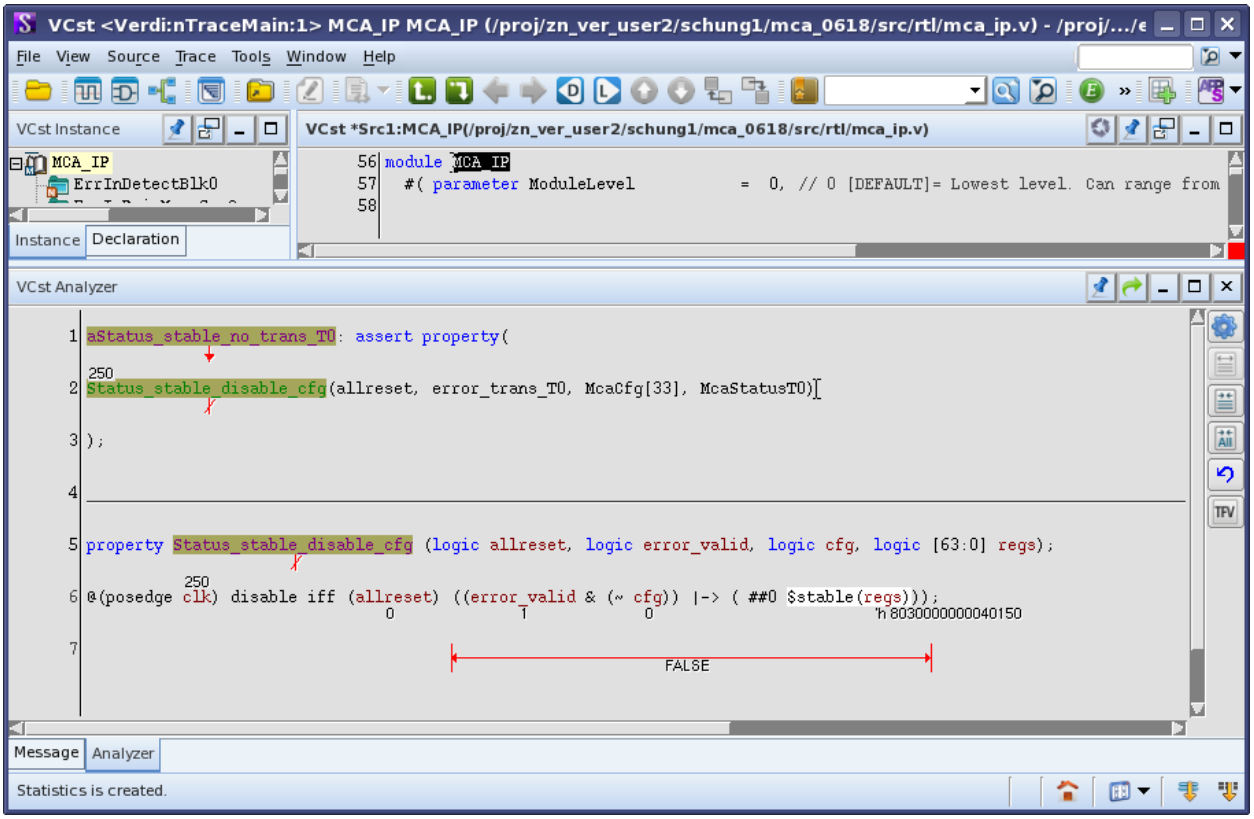

Figure 5: Verdi showing falsified property

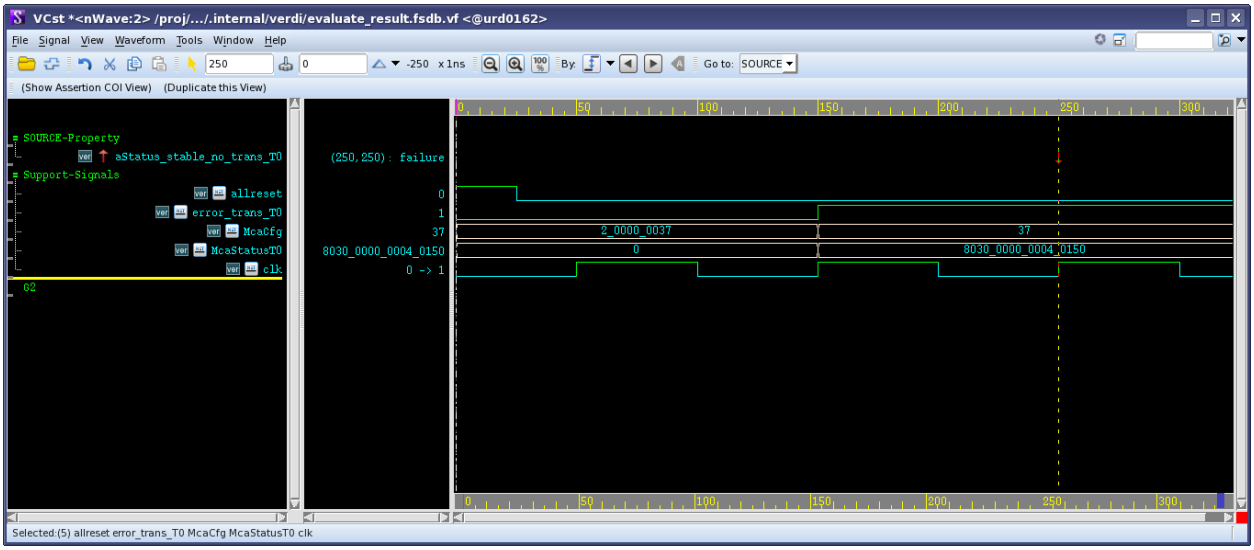

Figure 6: Verdi showing falsified property waveform

3.4 Functional check on paths from error inputs to IP outputs

The path from error inputs to IP outputs is more complex, because it uses the previous registers output values. The spec is defined as follows:

McaErrTrig (Interrupt) happens when a fatal, uncorrected, corrected or transparent error is received and McaStatus register overflow bit[63] is low.

A property can be defined as follows:

```
assign DetectMachineCheckExceptionT0 = int_fatal_T0 | int_uncor_T0 | int_corre_T0 |
int_trans_T0;

property Intreq3 (logic allreset, logic DetectMachineCheckException, logic block_ext, logic ovf,
logic intreq3);
  @(posedge clk) disable iff (allreset)  (~$past(intreq3) & DetectMachineCheckException &
~$past(ovf) & block_ext) |-> ##0 intreq3;
endproperty

aIntreq3_T0: assert property (Intreq3(allreset, DetectMachineCheckExceptionT0, block_ext,
McaStatusT0[63], IntReqInternal[3])) else $fatal ($stime,,,"%m assert FAIL");

cIntreq3_T0: cover  property (Intreq3(allreset, DetectMachineCheckExceptionT0, block_ext,
McaStatusT0[63], IntReqInternal[3])) $display ("%m cover PASS @%0d", $time);
```

Note that, as condition, the past value of the overflow(ovf) register should be written as ~$past(ovf). What happens if users write the current ovf instead?

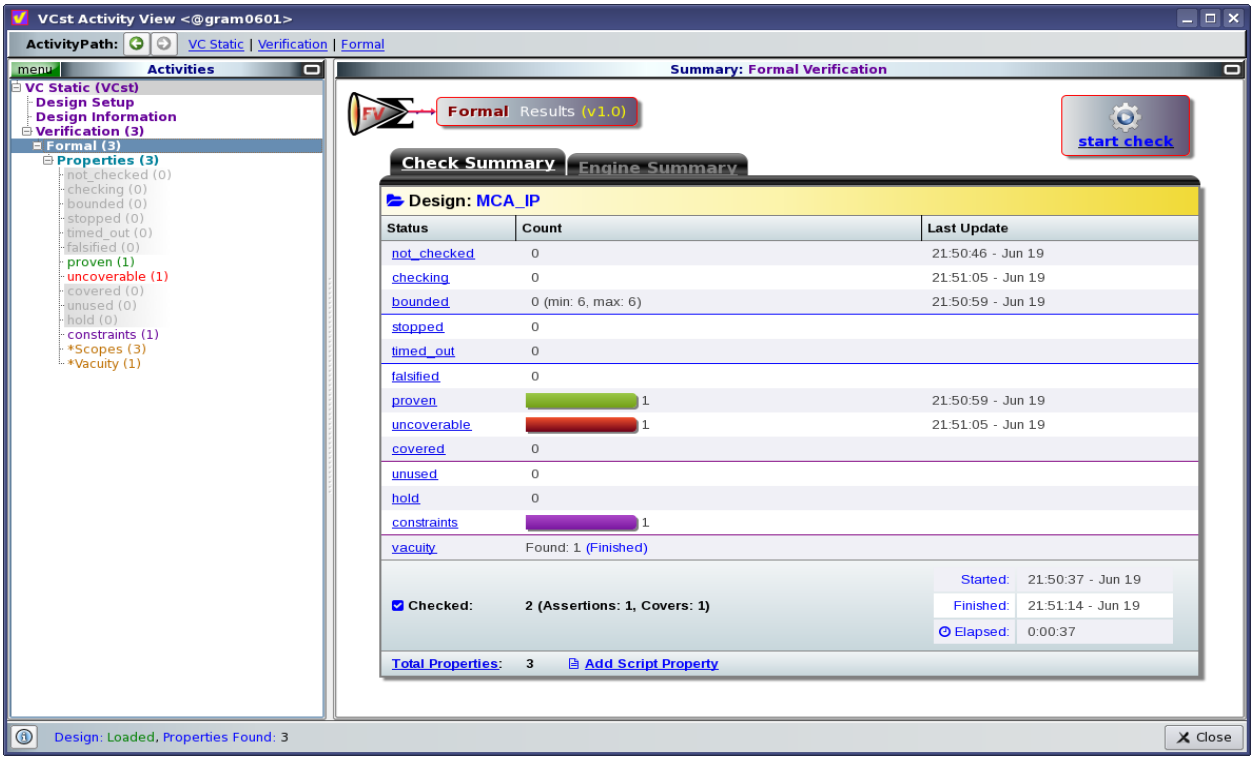

Figure 7: Uncoverable screen shot

Given the screenshot in Figure 7, the assert property appears to be proved, but the cover property is uncoverable. The next screenshot, Figure 8, shows that it is vacuous meaning that the tool has difficulty to determine if the function is logically sound. The property expression of the example above is vacuous, since it has a potential race condition.

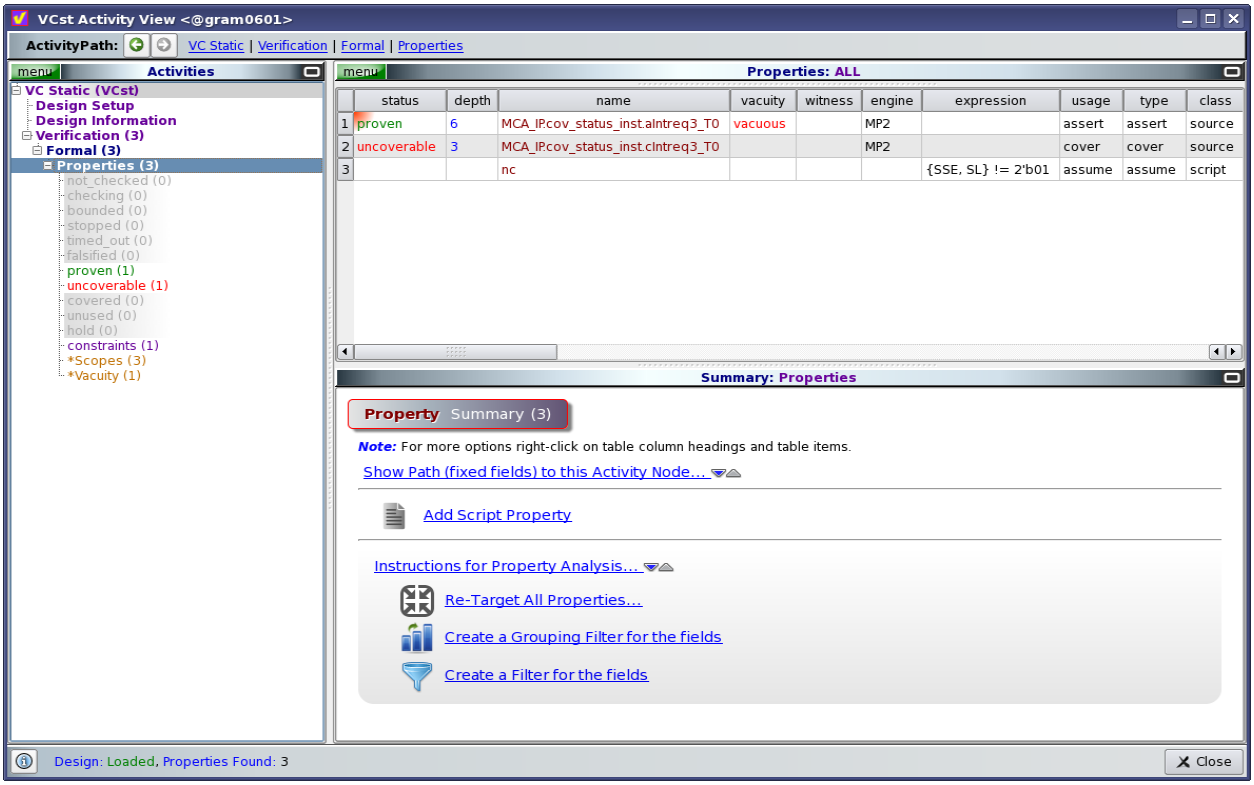

Figure 8: Vacuous screen shot

4. Conclusions

This paper shows how to apply formal verification given the spec. The spec is interpreted to make behavioral models with SVA, and the formal verification tool compares these models to RTL. The formal verification tool supports all GUI and text debugging capability for equivalence fail cases.

It is verification engineers' role to make behavioral models from the spec. It should be reminded that there could be some functional verification failure if the spec is not defined completely. However, given that this spec is complete the formal verification reached 100% functional coverage.

There were some cases where the spec was not defined, and traditional random simulations caught those fail ones. The spec was updated, and so were the formal verification behavioral models. This proves that verification engineers still need simulation based verification. This paper shows that formal verification can be used as the main verification methodology, and simulation based verification as supplementary.

Although not every design can be verified 100% with formal verification, the formal verification of this specific IP achieved that goal because of two reasons:

- All functions of the IP are end-to-end (source-to-destination). Each function can be modeled with SVA.
- The spec is well defined and documented before or in the middle of the project. In rare cases the spec is not defined clearly.

5. References

[1] SystemVerilog Assertions and Functional Coverage, Ashok B. Mehta, Springer, 2014

[2] VC Formal Verification User Guide, Version J-2014.12, Synopsys, December 2014