FSM Minesweeper – Scalable Formal Verification Methodology for Detecting Hangs in Interacting FSMs

Anshul Jain, Achutha KiranKumar V M, Harbaksh Gupta, Shashwat Singh
Intel Corporation
{anshul.jain, achutha.kirankumar.v.m, harbaksh.gupta, shashwat.singh}@intel.com

***Abstract*- Hangs have traditionally evaded the dynamic simulations and cause serious schedule risks for SOCs' time-to-market. Majority of hangs attributes to deadlocks and livelocks in interacting finite state machines. Modern micro-architectures use interacting FSMs for modular implementation of complex flows which are vulnerable to synchronization issues causing the sub-system to hang. FSM minesweeper is a novel methodology which leverages the exhaustive analysis of formal technology to identify all the cases of deadlocks and livelocks in a system of interacting FSMs by using a straightforward solution which is eminently scalable to large IPs.**

I. Introduction

Traditionally, IP teams have relied upon IP-level dynamic simulations to verify requirements such as absence of hang, coherence, security etc. Due to the complexity of feature-rich modern designs, coverage of corner-case scenarios in dynamic simulations is predictably low which makes the product schedule vulnerable to the possibility of showstopper, late-stage sub-system level bugs escaping the verification process. Hangs are those notorious bugs which have historically escaped dynamic simulations and caused panic in the later stages of design signoff. Hangs are mostly rooted in control logic of the design which malfunctions in a very specifically timed, highly unlikely sequence of events. Formal verification, equivalent to an exhaustive sequence stimuli verification, can help in finding issues in these complex timed interactions, thus avoiding late bug escapes.

Major formal verification tools implement model checking [1] for finding bugs which suffers from the exponential computational complexity. Thus, proving absence of hangs on RTL implementation of an entire IP is unrealistic. Therefore, we focus on the most worrisome subcategory of hangs which attributes to interacting finite state machines (FSMs) [2].

FSM minesweeper is a novel methodology that leverages the exhaustive breadth-first search analysis of formal technology to identify all the cases of deadlocks and livelocks in a system of interacting FSMs, uses a straightforward solution of mining bugs in a system of interacting FSMs and manages formal complexity barriers effectively. These qualities make the methodology scalable to large IPs. Objective of the famous “minesweeper” video game is to clear a board containing hidden “mines” or “bombs” without detonating any of them. However, the objective of FSM minesweeper is the exact opposite – to expose all the hidden “mines” or “bugs” by detonating them.

Novelty of our methodology lies in the bottom-up strategy as compared to top-down strategy of traditional formal verification methodology [3]. Traditional methods demand comprehensive knowledge of “specific” requirements of the sub-system to implement an end-to-end forward progress checker. Our methodology relies upon “general” axiomatic requirements of an FSM. Therefore, it can be deployed without acquiring detailed design understanding before starting the execution. And the nature of counterexamples guide verifiers in acquiring relevant knowledge about the design saving extraneous ramp-up effort.

We have applied our methodology in server and client memory sub-systems, and it has yielded results that showcase its effectiveness in finding hang-related bugs which are (1) located within individual FSMs, (2) located in glue-logic between FSMs and (3) architectural in nature. Currently, our methodology is semi-automated. However, we envision complete automation with further investments.

II. METHODOLOGY – FSM MINESWEEPER

We propose the following stepwise approach to mine hang-related bugs in a sub-system comprising of multiple interacting FSMs.

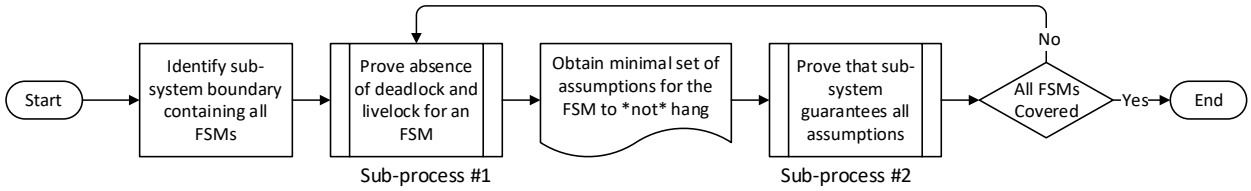

Figure 1: FSM Minesweeper Flow Chart

A. Identify design boundary containing interacting FSMs of interest i.e., main Device Under Test DUT

Since we require only the FSM implementations and the glue logic facilitating interactions between FSMs, therefore entire RTL implementation of an IP is irrelevant – targeting entire IP may unnecessarily increase the formal complexity. We recommend to identify all the FSMs in the IP (major industrial formal property verification tools are capable of listing FSMs in a given design) and pick that module in the design hierarchy which instantiates all the FSMs.

B. Prove absence of deadlock and livelock assertions for each FSM

Create standalone DUTs – one for each FSM and implement deadlock and livelock assertions for individual states of the FSM using liveness [4] properties. Deadlock assertion means that "*FSM should be able to transition out of a given non-reset state eventually*". Livelock [5] assertion means that "*FSM should be able to reach reset state from a given non-reset state eventually*". Some industrial formal property verification tools are capable of generating such deadlock and livelock assertions automatically. Hence, the process of generating deadlock and livelock assertions can be automated. Figure 2 shows an FSM implemented for scheduling transactions and example of deadlock, livelock assertions.

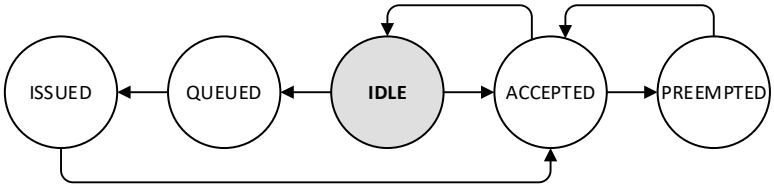

Figure 2: Issue Queue FSM

Deadlock Assertion Example

```
(iq_state == ACCEPTED) |-> s_eventually (iq_state != ACCEPTED)
```

Livelock Assertion Example

```
(iq_state == ACCEPTED) |-> s_eventually (iq_state == IDLE)
```

Run deadlock and livelock assertions to find failures/cases where the FSM is either stuck in a particular state or stuck in a loop among a set of states. Deadlock assertion failure means that either FSM is waiting for an external dependency to be resolved or FSM does not have an exit arc even after all the external dependencies stand resolved. Livelock assertion failure means either FSM is relying on a loop-breaker mechanism ensured by sub-system or FSM is stuck in an incessant loop. Former cases attributes to fairness assumption(s) to be guaranteed by the sub-system and later cases attributes to an RTL bug. Figure 2 shows one such loop between ACCEPTED and PREEMPTED state. Once all deadlock and livelock assertions failures are debugged and proven as shown in sub-process 1 in figure 2, we get a minimal set of assumptions that the FSM is expecting sub-system to guarantee. Though liveness properties are comparatively complex to prove than safety properties, however smaller Cone Of Influence (COI) of individual FSM in standalone mode makes convergence achievable.

C. Prove that sub-system guarantees assumptions made by each FSM

Previous step of the methodology essentially provides us with a minimal set of rules or requirements in executable form (SVAs). We run the assumptions made by each individual FSM as checkers on entire sub-system i.e., our main DUT and prove all assumptions in the minimal set obtained from step 2. Though these assumptions are checked on relatively larger DUT, the total COI is still much smaller and more manageable than that of the entire sub-system.

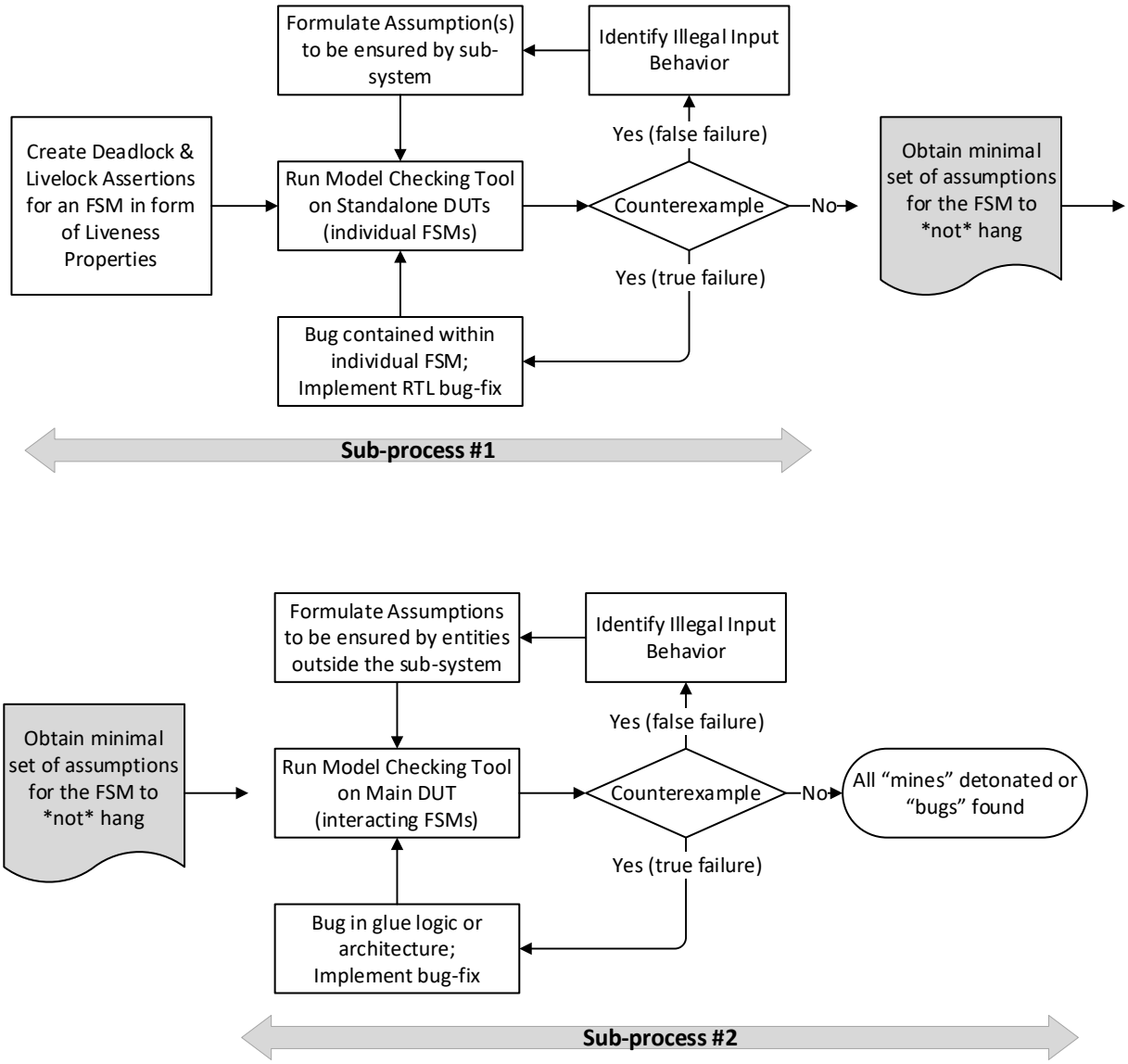

Figure 3: Sub-processes Flow Chart

Step 2 and 3 of the methodology should be iterated for each FSM to eventually prove the absence of hangs in interacting set of FSMs. The entire process explained so far can be visualized as minesweeper game where each click on a cell (FSM) uncovers mines (bugs) or exposes more cells (interaction requirements).

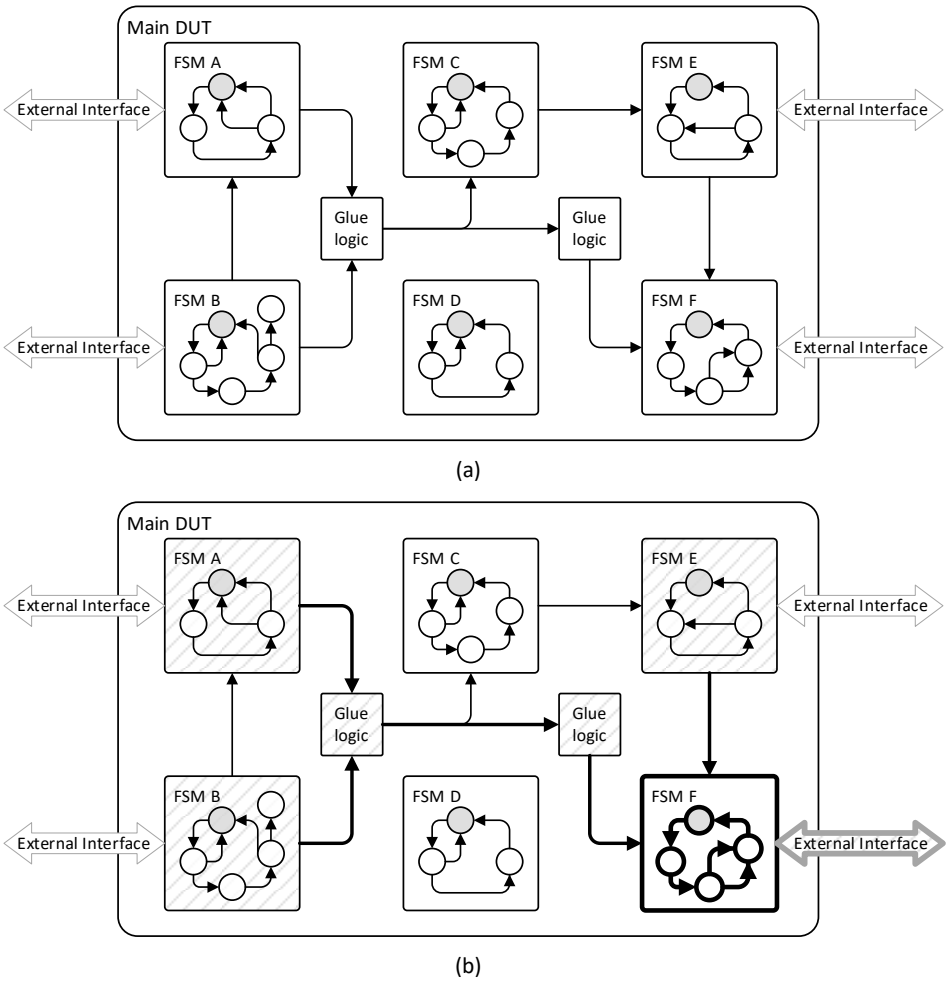

Figure 4: FSM Minesweeper Example

Figure 4 (a) represents an example of sub-system containing multiple FSMs of which A, B, C, E and F form a set of interacting FSMs. Figure 4 (b) depicts how step 2 on FSM F will expose deadlock and livelocks contained inside FSM F as well as help make executable requirements of interactions between FSM F and FSM A, B, E and external interface.

III. Results

We applied the methodology on two representative designs in memory sub-system – (1) Power Management Agent (PMA) of a client CPU. (2) Memory Controller (MC) of a server CPU. We have found 4 late stage hangs in PMA and 5 critical hangs in MC with FSM Minesweeper methodology.

A. Power Management Agent (PMA)

PMA integrates all memory sub-system IPs to provide a single modular interface between the memory sub-system and the SOC. It handles interactions with memory sub-system IPs (e.g., memory controller, inband ECC, fabric interface) for reset and power management with firmware. PMA houses 14 interacting FSMs. We found a bug where an FSM becomes unresponsive in case when back-to-back instructions received in short span of time. RTL implementation did not take care of overwrite scenario and did not implement a necessary arc from a particular state causing a deadlock/hang in memory sub-system.

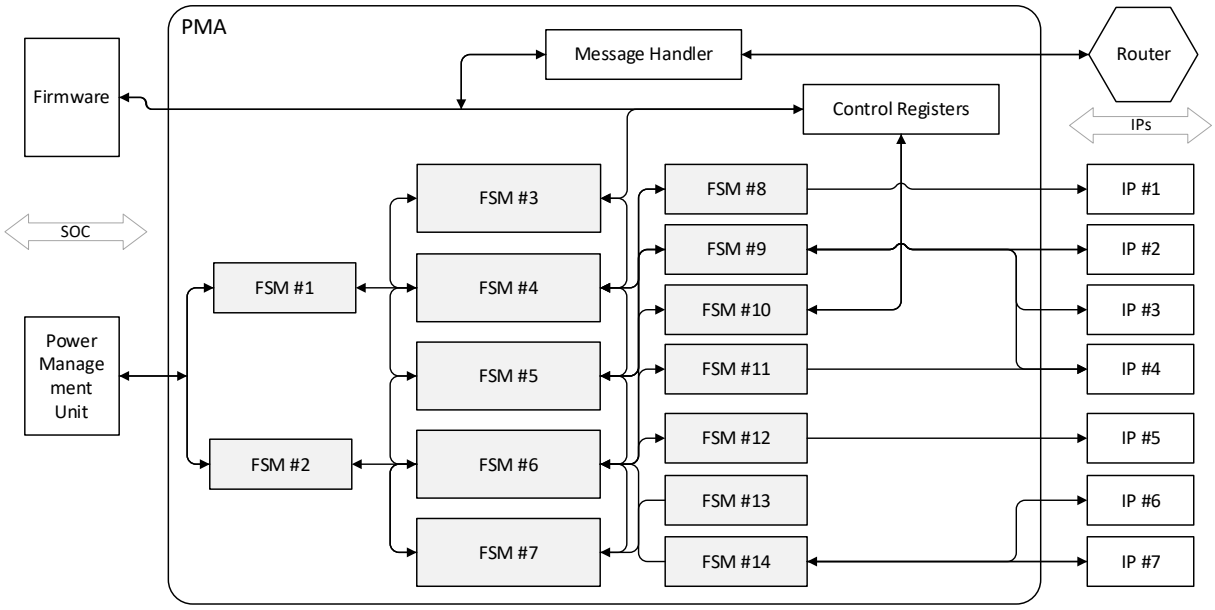

Figure 5: PMA High-level Block Diagram

Example of hang found through sub-process #1 – "FSM #4 deadlocks in CALC state when Power Management Unit (PMU) overwrites download message". Typically, FSM #4 expects download request message from PMU in IDLE state. FSM #4 then triggers index calculation and sends an ack message to PMU. FSM #4 uses the ack message to transition to next state. Sometimes, PMU can send another download request to overwrite the first download request. In such cases, FSM #4 transitions back to CALC state for index calculation and needs to send another ack to PMU to proceed. RTL implementation did not take care of overwrite scenario causing a deadlock in FSM #4 and hang in memory sub-system. Please refer to figure 6 and 7 for details.

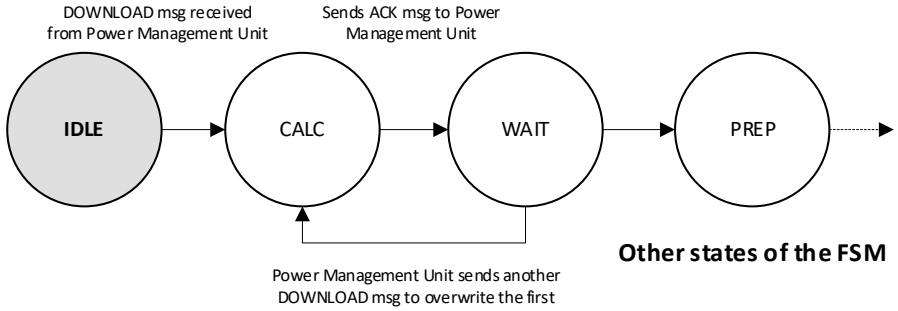

Figure 6: State Transition Diagram

Buggy RTL Implementation

```
arc_IDLE_to_CALC = (state == IDLE) & msg_req_rise;
arc_WAIT_to_CALC = (state == WAIT) & msg_req_rise;
msg_ack_set = (arc_IDLE_to_CALC);

always_ff @(posedge clk) begin
   if (msg_ack_clr) msg_ack <= '0;
   else if (msg_ack_set) msg_ack <= '1;
end
```

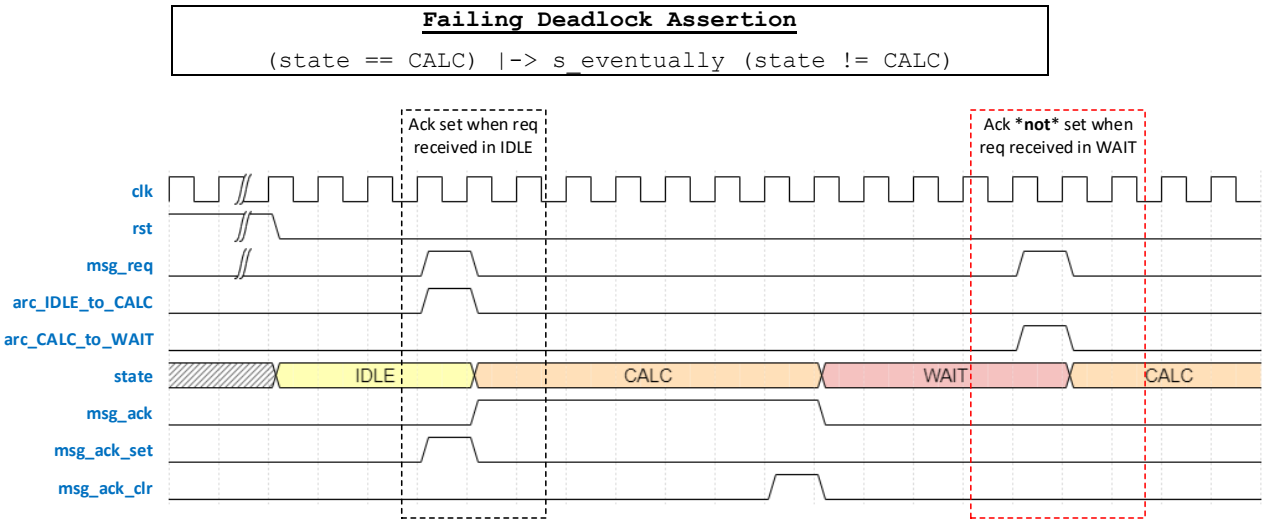

Figure 7: PMA Bug Waveform

Table 1: Overall Deadlock/Livelock Bugs Found in PMA

Bug Category	**Located within individual FSMs**	**Located in glue-logic between FSMs**	**Exists in Architecture of FSMs**
No. of Bugs Found	2	1	1

B. Memory Controller (MC)

Memory controller sub-system implements multiple error flows using interacting FSMs. These FSMs work in tandem in various modes for different types of errors. A late-stage architectural hang was found in SOC validation where error retry flow created a hang in a particular mode. We applied our methodology to reproduce this architectural hang for proof-of-concept (POC). We were not only able to reproduce the bug but also prove the robustness of the bug fix. Successful results from the POC were extended to next generation of the MC IP and 4 critical hangs were found using our methodology.

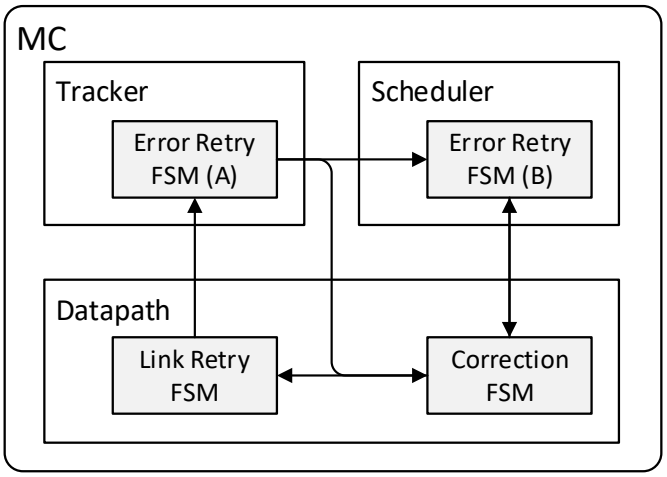

Figure 7: MC High-level Block Diagram

Example of hang found through sub-process #2 – "A late-stage architectural hang was found in SOC validation where error retry FSM (B) and correction FSM became out-of-sync in Persistent Fault Detection (PFD) mode when an uncorrectable error is encountered". When we applied our methodology to root-cause the bug, it was found in

Correction FSM which decodes the error to be uncorrectable but fails to stop the error retry. This particular functional issue in Correction FSM stalls the Error Retry FSM (B) in CORR state indefinitely.

To root-cause this issue, we ran sub-process #1 on Error Retry FSM (B) by implementing deadlock and livelock checker on each of its state. At the completion of sub-process #1, we identified a set of fairness assumption required by Error Retry FSM (B) to *not* hang. One of the fairness assumptions that emerged from sub-process #1 was "**Error Retry FSM (B) should receive stop indication within finite duration of receiving start uncorrectable error is detected in PFD mode**".

In sub-process #2, we ran the fairness assumptions on main DUT (system of interacting FSMs) as checkers and the aforementioned assumption failed for the buggy RTL. The failure scenario is explained in figure 8. Error Retry FSM (B) receives the start and stop indication from two different FSMs. Error Retry FSM (B) is kicked-off by Error Retry FSM (A) and needs to be stopped by Correction FSM eventually. In most cases, Error Retry FSM (B) is stopped by Correction FSM when it moves to IDLE state after correcting the error. However, in case of uncorrectable errors, Correction FSM stops the correction as soon as it realizes that the error is uncorrectable. In failing scenario, when uncorrectable error is detected in PFD mode, Error Retry FSM (A) kicks-off Error Retry FSM (B), Error Retry FSM (B) transitions to CORR state waiting for stop indication. However, due to below RTL implementation of Correction FSM, it transitions to IDLE upon detecting an uncorrectable error without sending the stop indication to Error Retry FSM (B) – shown in red in figure 8. This causes Error Retry FSM (B) to stay stuck in CORR state even after the transaction was completed.

Buggy RTL Implementation:

```
stop_corr = corr_err_pfd_en & // Persistent fault detection enabled
   ~corr_err_req_uc &         // Do not correct uncorrectable error
   ~corr_err_skip;            // Skip correction when interrupted
stop_retry_set = corr_fsm_exit_dec & stop_corr
```

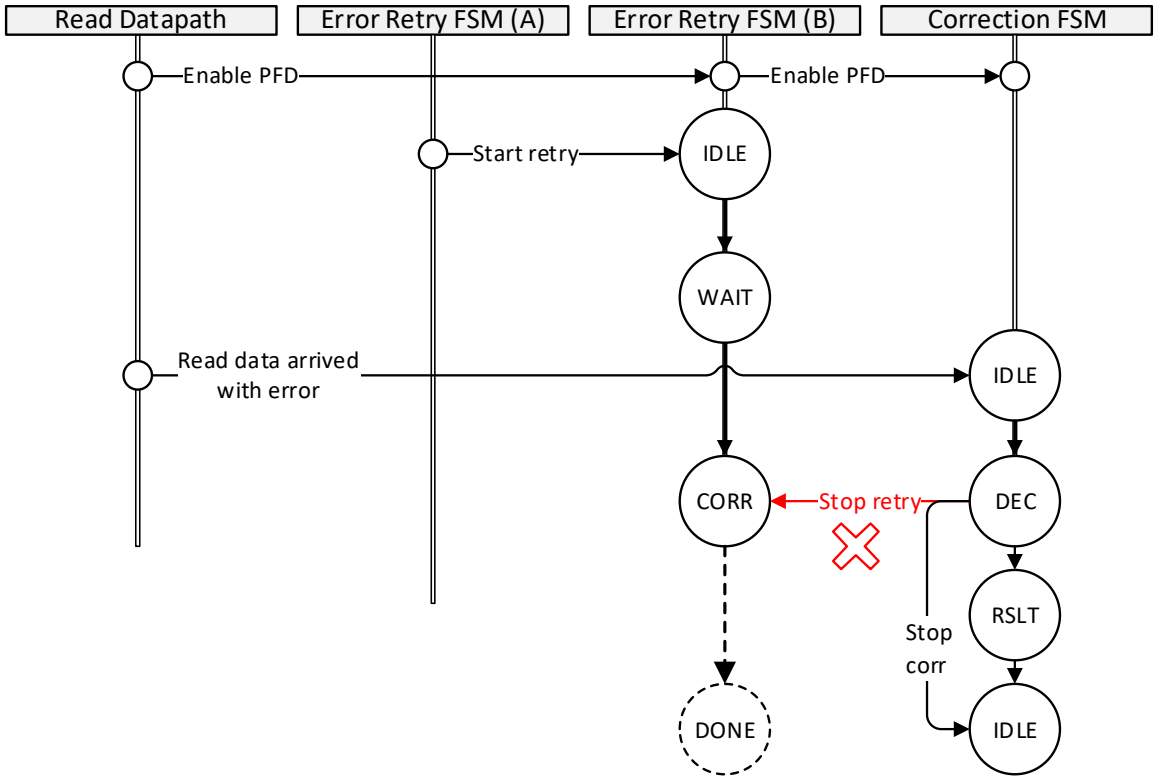

Figure 8: Error Retry Flow Description and Bug Details

Table 2: Overall Deadlock/Livelock Bugs Found in MC

Bug Category	Located within individual FSMs	Located in glue-logic between FSMs	Exists in Architecture of FSMs
No. of Bugs Found	2	1	2

IV. CONCLUSIONS

Proving the absence of hangs at IP-level is a critical challenge for today's design and verification teams. The problem is not well-addressed by traditional verification methods including end-to-end formal verification. In this paper, we have described a novel methodology which addresses a part of the problem effectively by focusing on hangs arising in interacting FSMs of a sub-system.

FSM minesweeper is an evolutionary application of formal verification and assume-guarantee technique which can be performed at IP-level RTL implementation to improve the quality of individual FSMs, identify issues in glue logic and find architectural flaws in the way multiple FSMs interact with each other. We have been seeing promising results from the proposed methodology, encouraging us to invest more in automating the methodology to spur wider adoption.

ACKNOWLEDGMENT

We would like to thank PESG, IPG & DDG management for their continued support for developing new methodologies, Vineesh V S from FVCTO for helping in formal debugs, Harish Mopidevi from Client MC IP team for supporting us in design understanding, and Server MC IP team for supporting us with design verification collateral such as failure waves etc.

REFERENCES

[1] Doron A. Peled, Edmund M. Clarke, and Orna Grumberg, Model Checking, Second Edition, 2018
[2] Adnan Aziz , Vigyan Singhal , Robert K. Brayton, Verifying Interacting Finite State Machines : Complexity Issues, 1993
[3] Book, Formal Verification – An Essential Toolkit for VLSI Design, 2015
[4] IEEE Std 1800™-2017, IEEE Standard of SystemVerilog – Unified Hardware Design, Specification, and Verification Language.
[5] Mark Eslinger, Jeremy Levitt, Joe Hupcey, "Deadlock Verification for Dummies – Easy Way of Using SVA & Formal", DVCon US 2020