Functional Safety Verification Challenges for Automotive ICs

by Mihajlo Katona, Veriest

INTRODUCTION

The ISO 26262 is an international standard for functional safety of electrical and electronic systems installed in serial production road vehicles. This standard recognizes two types of failures:

1. The first type is **Systematic Failures**
   These failures are induced in a deterministic way during the system's development, manufacturing, or maintenance. They are process-related issues and are sourced in pre-production activities, such as specification issues or manufacturing defects. The ISO 26262 standard targets preventing and avoiding these kinds of failures.
2. The second type is **Random Failures**
   Random failures are related to random defects and process or usage conditions such as radiation or silicone wear out. Random failures are separated into two groups: permanent faults and transient faults. ISO 26262 also targets controlling random failures.

At Veriest, as an ASIC services company, we handle many different automotive projects with varying safety level requirements. In this article, I'd like to share some of the "lessons learned" in these projects.

The verification problem we are targeting here is that state-of-the-art functional verification methodology is not directly supporting verification of random failures within ISO 26262 requirements for functional safety. We believe a verification methodology is required to distinguish functional safety verification from classical functional verification flow. For this, we need new tools and verification approaches, and stricter and well-documented verification procedures. And our particular focus is transient faults that can randomly upset the state of a system we are verifying.

In a semiconductor world, functional safety is all about data storage and data movement through the system. Electrical or magnetic interference inside hardware systems can cause a single bit to flip to the opposite state spontaneously. And this is a typical case for random failure, which we desperately need to analyze and see its effects on the functional behavior of the system we are verifying.

FUNCTIONAL SAFETY VERIFICATION CHALLENGE

In a usual ASIC implementation flow, we are always starting from a design specification. The design team interprets the specification and defines a microarchitecture for the safety features we are analyzing. On the other side, the verification team is doing an independent analysis and is building the functional verification environment.

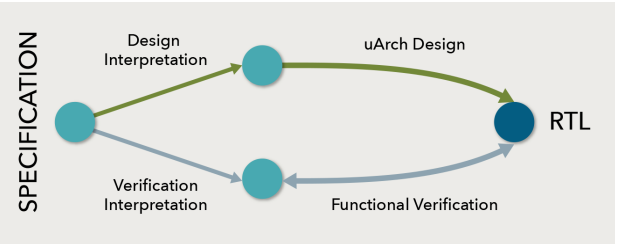


Figure 1

The goal of verification is to build a testbench and test plan for functional verification of the safety features in our design under test. By the methodology definition, a black-box verification approach is required to verify the system only through the available interfaces without any knowledge of the actual implementation of the design.

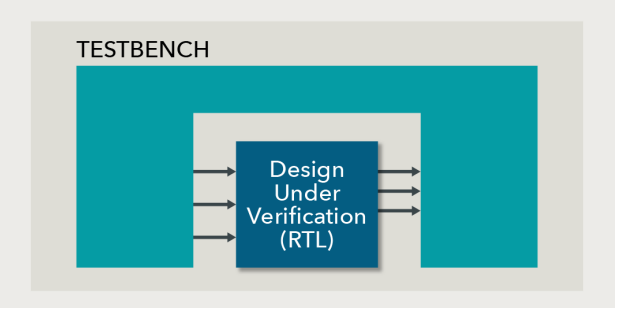


Figure 2

If we take our design example, an automotive AI processor, we can see typical data flow processing architecture receiving input data from some data buffer. This data is then distributed to the processing elements through the specified distribution network. In an AI processor, data processing is based on a multiply-and-accumulate architecture. This processing block is usually parallelized and has additional internal memory buffers supporting the required data transformation flow. Further, there can also be some post-processing mechanism in a data pipeline before sending the results to the output memory buffer.

And as usual, there are some configuration and status registers through which users can navigate the data processing flow according to their needs and monitor relevant status information.

As we can see in Figure 3, there are different memory blocks in this data processing pipeline. To ensure fault-less memory status, we need some data protection mechanisms implemented in the system.

If we start with one memory block, we need to have a block to calculate error correction codes before we write data into memory. This is typically based on a data word that enters the system.

When we read data from memory, we also read the ECC code stored during a write cycle. Each information is sent to the ECC check and correction module. At the output, we have the corrected read data and the ECC status check information.

In some advanced systems with stricter safety targets, we might have a situation where ECC calculation and check include the address information and data (Figure 4).

From the verification strategy perspective, we need to define hooks in the system to check that ECC mechanisms are implemented according to the functional specification and safety requirements. For this purpose, an ECC verification agent is required. This agent needs to connect the ECC model to the primary data source. Further, the ECC agent must check that the write cycle is correct. This write checker takes reference information from the ECC model and checks if the system behavior complies with the specification.

Next, the ECC agent also must investigate the read cycle. For this, we need two hooks in a system: The first one is the output from memory. The second hook is the output from the ECC check and correction module. Reference data is also taken from the ECC model.

And finally, the ECC agent must report the ECC status information to the system controller. From the functional verification perspective, these are all passive components and functional checkers.

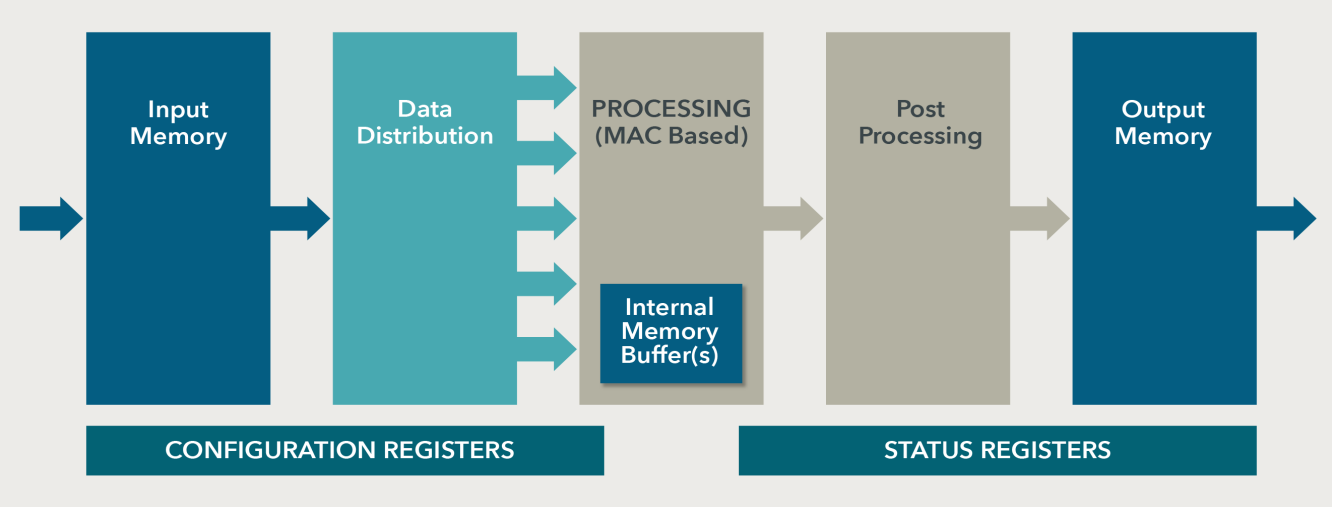


Figure 3

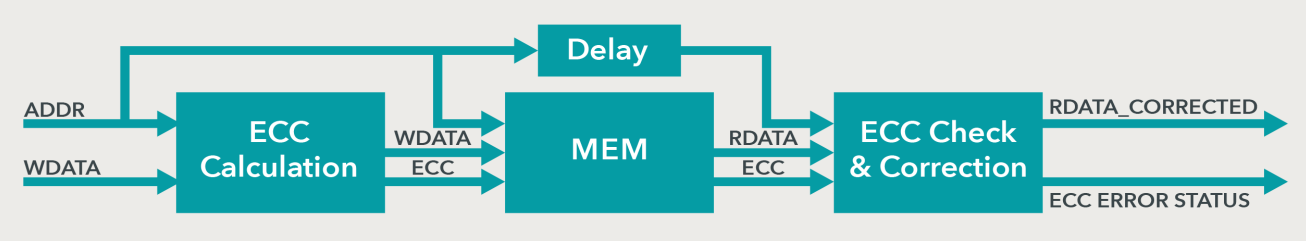


Figure 4

But our ECC agent also needs to support the active error injection to the system, emulating the permanent or transient faults that might happen during the system lifetime. This requires a constrained-random BFM and a driver to generate and drive required error types. Those errors can be of the permanent type, and they are typically connected to the memory models. For verifying transient faults, we are interested in data manipulations on the read data path. Here, the focus is on analyzing single bit upsets in the logic that uses memory data.

The ECC agent must have an ECC predictor, which will create expected results related to the ECC. These results will be later used to check the ECC status reported by the system (Figure 5).

This short analysis shows the challenge when it comes to the verification of a functional safety mechanism. Firstly, we need to have an error injection inside DUT. Since verification needs to have insights into DUT internal signals, the standardized black-box methodology is not applicable. We need a new approach – white-box verification.

In this case, error reports are desirable. Usually, error reports indicate something is not good, but we want to see errors reported from the system and the verification environment. This means the DUT functional state is disturbed, and

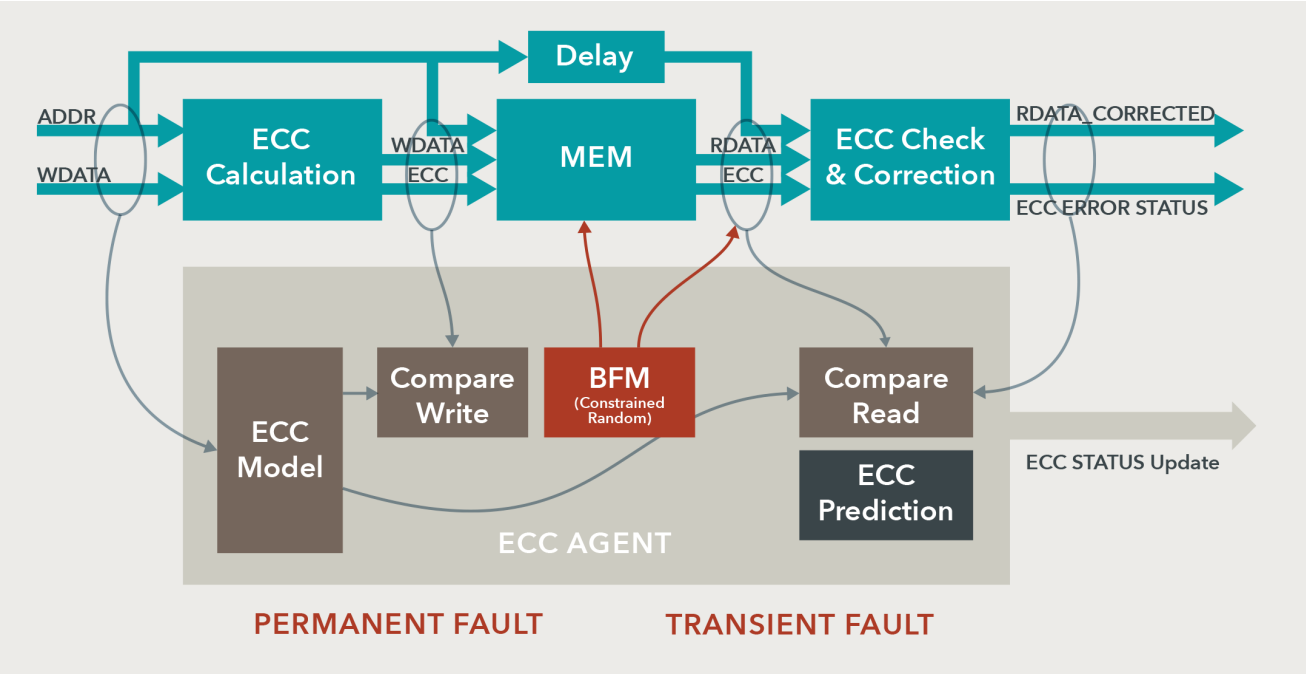


Figure 5:

some actions are expected. With this, the classic verification tree is growing a new branch to meet ISO 26262 requirements, which implies white-box error generation and injection.

ERROR INJECTION MECHANISMS

From the safety UVC architecture perspective, an ECC agent needs to have ECC modeling, ECC checkers, error injection, error monitoring and prediction, error reporting, and recovery flow implementation. From the verification perspective, all those requirements are aligned with typical safety agent architecture that contains sequencer, BFM, signal sampling, collector, and monitor with scoreboard and checker's block. The environment is also required to have specific configuration parameters and register model.

Note that for the error injection, we need to have a specified list of the signals through which faults are randomly injected. Error injection constraints are defined within the sequence driver implementation, and actual signal upset events are defined within the safety UVC bus functional model.

From the error injection perspective, we have two options: signal deposit or signal force. Whatever we choose, the error injection must be implemented by bit flipping on the design side. Signals corrupted from the verification environment must be agreed upon with the design team. This is a crucial requirement for achieving required safety targets.

But while defining this list, try to select registers and not wires or nets in a design. Verilog nets have a resolution function that might give an unexpected value if there are multiple drivers on that net. For example, suppose the verification environment is driving this net to inject a transient error, and the design is also driving that net to a different value in the same simulation time due to other functional requirements. In that case, we might get an unexpected value on that net due to the resolution function that the simulator will apply (Figure 6).

To summarize, signal deposit gives a value to a net or register that will propagate forward. The signal retains that value until its next scheduled change. A signal deposit gives the best performance when used for corrupting memory bits or simulating not-desired flip-flop toggles in the data processing pipeline.

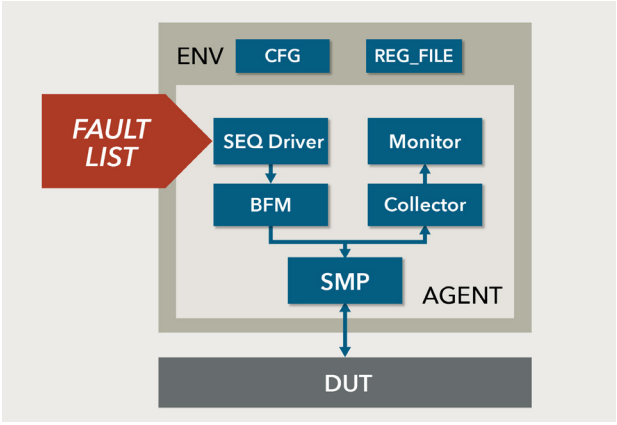


Figure 6

Signal force is a slightly different mechanism. It forces a value to continuous assignments that will propagate forward. It overrides all other drivers and stays in effect until replaced with another force or canceled with a release. This method should be used for injecting permanent faults. In addition, it can also be applied for injecting transient faults to the system, together with a signal release, as illustrated in the example below.

```
task force_error( ... ) ;
 @(negedge if.clk);
 if (a_error_type in [DATA_ERROR, DATA_AND_PARITY_ERROR])
   begin
   force p_smp.psl signal data[a sig_idx] = a_data_corrupted;
 end
 if (a_error_type in [PARITY ERRORS, DATA_AND_PARITY_ERROR])
   begin
   force p_smp.psl_signal_parity[a sig_idx] = a_parity_corrupted;
 end
endtask : force_error

task release_error( ... ) ;
 @(posedge if.clk);    // skip next rising edge, wait for signal force
 #(CLK_PERIOD/3);   // avoid race conditions with monitor,
                       // release signals after clock rising edge
 if (a_error_type in [DATA_ERROR, DATA_AND_PARITY_ERROR])
   begin
   release p_smp.psl_signal data[a_sig_idx];
 end
```

```
if (a error_type in [PARITY_ERRORS, DATA_AND_PARITY_ERROR])
  begin
  release p_smp.psl_signal_parity[a_sig_idx];
end
endtask : release_error
```

In this example, we have a SystemVerilog task forcing a corrupted value to a signal at the negative edge of the clock signal. Depending on the generated error type, this task will force a value to a data signal or a signal that holds parity bit information. If the system enables double error injection, the task might force a corrupted value to both the data and parity signals.

Then, the verification environment must also release the force to model transient fault injection. This task must be executed after the error is injected. In this example, error injection is on the negative clock edge so that the signal can be released with the next rising edge of the clock. To ensure no race conditions with the monitor or any other part of the simulated Verilog code, we might want to move the signal release slightly after the rising edge of the clock signal. And then, depending on the injected error type, we release data signal, parity signal, or both.

From the timing perspective, the error is generated on a negative clock edge. Then the force and release tasks are started with the next rising clock edge, and the signal force is applied on the first negative edge after that. Lastly, the force is released slightly after the following rising edge of a clock signal. This approach implements error injection for a transient fault that lasts longer than half a clock (Figure 7).

There are also specific challenges when injecting errors into the data processing pipeline. Corruption of some data bits might not influence the results and signature check if we look at a simple limiter that reduces the 16-bit value to an 8-bit result. If the error is injected outside the output value range, it will be ignored and will not influence the signature value (Figure 8).

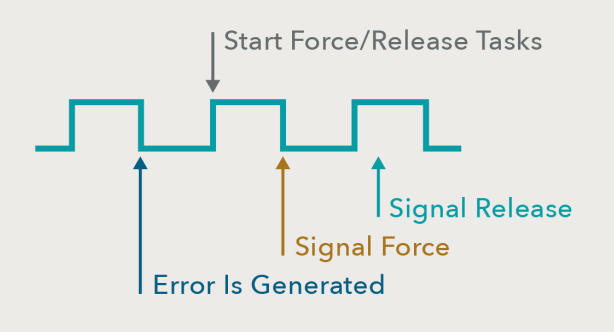


Figure 7

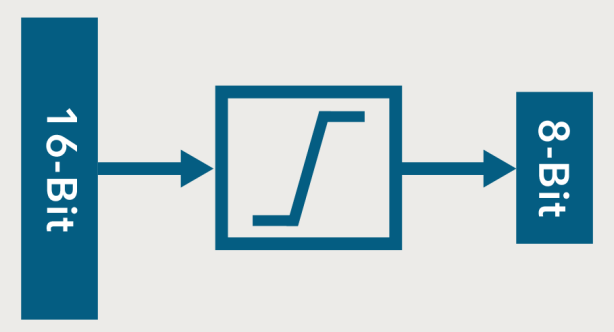


Figure 8

Also, note that the injected error might take several clock cycles to impact the result and signature due to multi-cycle paths and pipelined organization. Using a signal deposit for error injection in such data paths is more convenient, but please be mindful of the Verilog resolution function.

SAFETY MONITORING AND RECOVERY FLOW

Another critical aspect of the safety UVM implementation is the safety monitor. This block must observe system behavior when errors are being injected but also when errors are not injected in the system to ensure that false error reports are not detected with DUT. With this definition, the safety monitor must execute three threads in parallel:

- The signal sampling
- Idle monitoring during periods when the verification environment does not inject errors, and
- Error checking when transient faults are injected from the safety UVC BFM

For example, we can start those three threads from the UVC monitor after the reset period from the pre_main_phase (part of the UVM phasing mechanism).

```
class safety_monitor extends uvm_monitor #(my_transaction);
  `uvm_component_utils(safety_monitor)
  ...
  task pre_main_phase (uvm_phase phase);
    forever begin
      @(posedge dut.reset);
      fork
        sample_signals();
        idle_monitor();
        check_dut_error();
      join_none

      @(negedge dut.reset);
      disable fork;
    end
  endtask: pre_main_phase
endclass: safety_monitor
```

In addition to the monitoring, it is crucial to ensure that the error recovery flow works. It's essential to ensure that when the system detects an error, it is reported to a user through the specified mechanism, and the appropriate control action is taken. We call this action the error recovery mechanism, and its execution is required for returning the system to normal function.

For example, if the error is reported from the functional safety monitoring block, the user must check the error source by reading appropriate status register. Then, depending on the error source, some specific action must be taken. After this action is completed the verification environment must make sure that normal functionality is continued.

For realistic modeling, a recovery flow should use front door access to the status information, for example through the APB or AXI register access methods. Those transactions take certain time in clock cycles, particularly if clock division is applied to an interface that is used for access to status registers.

During this period, verification environment might randomly choose to inject new error into the system. This scenario puts a stress on verification environment with at least two errors injected but only one error recovery procedure taken. Due to this, the verification team must ensure that the ECC agents are unifying multiple errors into one recovery flow with an atomic approach to a ECC error recovery.

When critical event is detected, a specification might require that all data processing is stopped, and reset is executed. Such error recovery flow for critical events must be also verified. It's important to re-start the processing after the reset is executed to check that system is back to normal functional mode after the critical error recovery flow is executed.

VERIFICATION MINDSET FOR FUNCTIONAL SAFETY VERIFICATION

If we look at the required procedures for functional safety verification, they are much stricter and more formalized to comply with ISO 26262 requirements. Verification procedures must be enforced and documented. In addition, during functional verification of the safety mechanisms, it is required to have continuous review process in place, where all meetings must be documented until all follow up action items are not implemented. And to achieve this different mindset is required. The safety procedure is the main point, not document for internal use.

CONCLUSIONS

In one simple sentence, functional safety is related to a safe machinery that will not cause any risk to human life. Automotive ICs are the active systems with implemented safety mechanisms for mitigating failure effects before human life is endangered. This article presented some of the functional verification aspects that were successfully applied in several automotive projects. Hopefully, some of the presented ideas will help implementation of some other projects.

SIEMENS

Editor:
Tom Fitzpatrick

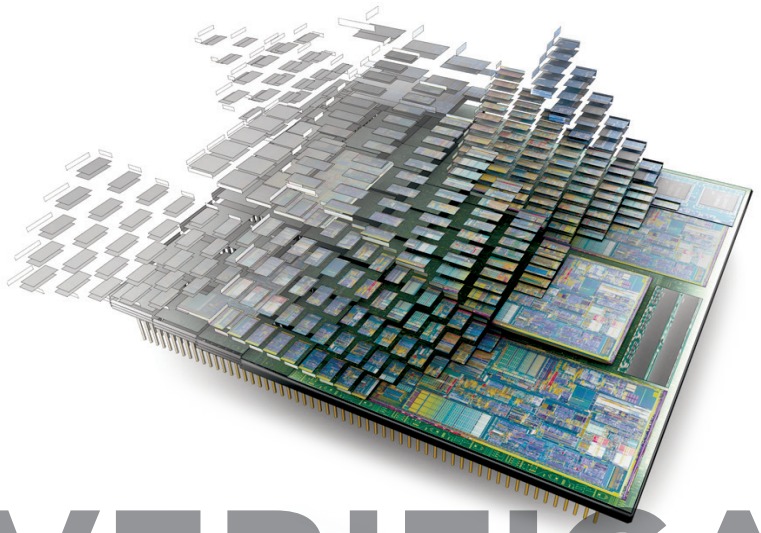
Program Manager:
John Carroll

Siemens EDA
8005 SW Boeckman Rd.
Wilsonville, OR 97070-7777

Phone: 800-547-3000

To view our blog visit:
VERIFICATIONHORIZONSBLOG.COM

Verification Horizons is
a publication of Siemens EDA



VERIFICATION HORIZONS

VH