# ISO 26262 Dependent Failure Analysis using PSS

Moonki Jang, Jiwoong Kim, Dongjoo Kim
Samsung Electronics, 1-1, Samsungjeonja-ro, Hwaseong-si, Gyeonggi-do 18448, Korea
moonki.jang@samsung.com, jiwoong7.kim@samsung.com, dj.num1.kim@samsung.com

Shai Fuss, Zeev Kirshenbaum
Cadence Design Systems, 18 Aharon Bert, Petah Tikva 4951448, Israel
shai@cadence.com, zeevk@cadence.com

**Abstract - Dependent Failure Analysis (DFA)[1] is a methodology defined in ISO-26262[2] to analyze the impact of errors in one shared resource on other elements and find safety vulnerabilities. Coherency is a good example for resource sharing and is thus critical for failure analysis. The reusability and constrained-random generation provided by the Portable Test and Stimulus Standard (PSS)[3] helps with generation of the Dependent Failure Initiator (DFI) and coupling factors for DFA.**

## I. Introduction

ISO 26262 is the adaptation of IEC 61508[4] to address the sector specific needs of electrical and/or electronic (E/E) system within road vehicles. This adaptation applies to all activities during the safety lifecycle of safety-related systems comprised of electrical, electronic and software components. Safety is one of the key issues in the development of road vehicles. Development and integration of automotive functionalities strengthen the need for functional safety and the need to provide evidence that functional safety objectives are satisfied. With the trend of increasing technological complexity, software content and mechatronic implementation, among others there are increasing risks from systematic failures and random hardware failures, these being considered within the scope of functional safety. ISO 26262 includes guidance to mitigate these risks by providing appropriate requirements and processes. [5]

Dependent Failure Analysis (DFA) methodology has been introduced as part of ISO 26262 part 11: Guideline on application of ISO 26262 to semiconductors [6] to analyze the impact of errors on one shared resource to other elements and find safety vulnerabilities. Many System-on-Chip (SoC) designs for automotive applications consist of multiple processor cores and IP subsystems that share SoC resources such as memory, system interconnect (IC), and IO subsystems. In this environment, a fault in a shared resource can affect other elements that share that resource, causing unexpected chain errors. When such an error occurs in a mobile phone, the problem can be solved simply by rebooting. However, in an automotive environment such an error can be fatal enough to threaten the driver's life. ISO 26262 defines these chain errors as dependent failures and classifies them as cascading failures and common cause failures.

- Cascading Failure: Failure of an element, resulting from a root cause, and then causing a dependent failure of another element or elements
- Common Cause Failure: Dependent failure of two or more elements, resulting directly from a single specific event or root cause

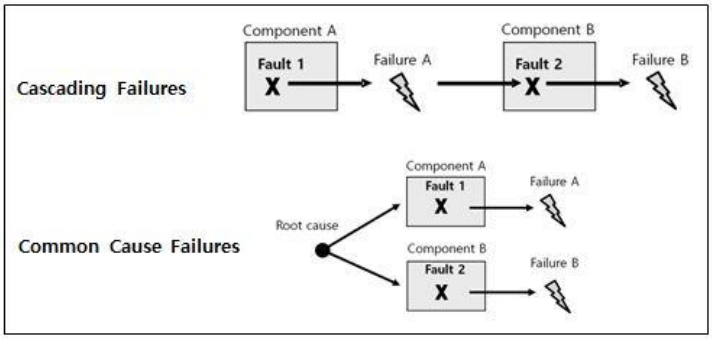



Figure 1. Type of dependent failures

The purpose of the DFA is to find the Dependent Failure Initiators (DFI) and coupling factors that can cause the above dependent failure, and to prevent or control them in advance. DFA generally proceeds in the following order:

- Identify the Dependent Failure Initiators (DFI)
- Finding possible coupling factors for each DFI
- Confirm effectiveness of safety mechanism and measures to prevent DFI and coupling, or to mitigate dependent failures

For DFA, it is necessary to generate DFIs of various conditions at a desired time point and generate numerous types of coupling factors to identify violations of independence or freedom of interference. This means that we need to create hundreds of scenarios that combine all of the functions that can be used as coupling factors for each DFI. We therefore needed a new, different kind of verification methodology to model and automatically generate these scenarios. We found that test generation using the modelling approach and constrained-random generation in PSS is suitable for generating numerous tests with different conditions as in the case of DFA. The use of constraint-based randomization in PSS allows us to efficiently create multiple randomized scenarios, in a fraction of the time it would take to manually create and maintain a large number of directed tests. The nature of randomization allows finding bugs even in unsuspected areas, as well as achieves a much better coverage. Functional coverage in PSS can be used to track the verification progress and point out areas that have not been verified yet, thus supporting a Metric-Driven Verification (MDV) approach. An additional advantage of PSS is the ability to generate executable code for various different platforms, including RTL simulation, Acceleration/emulation and even post-silicon.
We also collected the results of each test run and generated a new form of failure report - FTR, which was able to immediately reflect in DFA.

## II. IDENTIFYING THE DEPENDENT FAILURE INITIATOR

The Dependent Failure Initiator (DFI) represents the root cause of dependent failures in functional safety. In general, DFI is defined as an item that can threaten the independence required between elements. In this paper, a fault occurring in a shared memory area is defined as the DFI and implemented through fault injection as follows:

- Uncorrectable ECC error injection

When the Processor Element (PE) reads from memory, the ECC check logic can detect an error in the stored values and return an error response to the PE. In general, an ECC beat is calculated and updated when a data beat is written. The backdoor task allows you to directly access the data array inside the memory model to change the contents of a particular data beat. If there is a data beat that does not match the ECC beat, the ECC check logic will generate an ECC error when reading the data beat. Our testbench triggers such errors by changing memory contents through a backdoor interface. As shown in figure 2 below, ECC error injection can be performed on DRAM / L3cache / L2cache, which have different coherency targets.

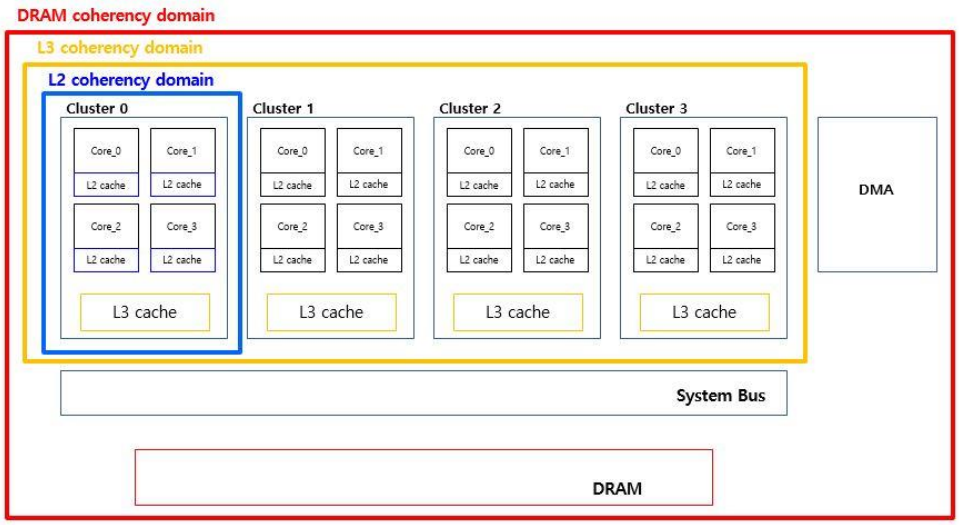



Figure 2. Coherency domain

- Memory Management Unit (MMU) translation fault generation

All PEs with an MMU look up the page table in the shared memory area and read the physical address information for address translation. If the contents of the page table are corrupted, serial translation faults can occur in the PE that references the page table. Our testbench manipulates the valid field of the page descriptor to generate an MMU translation fault at the desired time.

- RAS error injection for CPU, Interrupt controller, System MMU

The RAS (Reliability, Availability, and Serviceability) extension is ARM's error handling and recovery architecture that is applied from the ARMv8 architecture. [7] RAS has its own fault injection model extension that can inject faults into the L1 / L2 / L3 caches to generate error interrupts. However, this feature is mainly used for SW error handling and recovery process verification because it is a fake error rather than an actual error in cache memory. We reset the core and cluster during RAS recovery process after RAS error injection to verify the effect of this reset process on the coherency behavior of other processor elements.

Using PSS, the above fault injection options are modeled as reusable actions. A PSS tool can then generate various DFIs with the desired number of faults at any given time.

## III. FINDING THE POSSIBLE COUPLING FACTORS FOR EACH DFI

A coupling factor is a common characteristic or relationship of elements that leads to dependency in their failure. The following coherency interference stimulus for a shared memory region may be a coupling factor as it causes different PEs to continuously access a shared memory region. This results in a dependent failure from the previously defined DFI (fault injection into the shared memory region).

- False sharing access

The figure below shows the transaction flow when different coherent masters write their own data to a memory area allocated within one cache line (64 bytes). Each master uses a unique address-range within the same cache line. Each time a coherent master writes a value to a block allocated to it, a number of snoop transactions are generated between the coherent masters to clear the caches of all other masters, as shown in figure 3. If a fault is injected into the 64-byte cache-line, this coherency operation causes a failure in all coherent masters participating in the false sharing scenario.

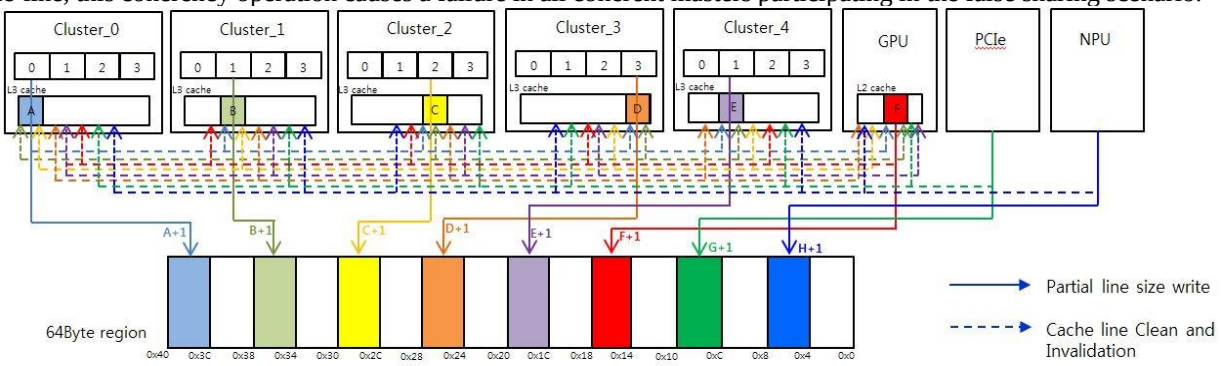



Figure 3. False sharing with write competition

- Distributed Virtual Memory (DVM) transaction broadcasting

DVM is a transaction generated by a processor for maintenance of a virtual memory system such as a MMU. As shown in figure 4, when Cluster_0 sends out a DVM request transaction, the DVM controller sends a request to the MMU elements of the system. When the response arrives from each MMU, the DVM controller sends a response to Cluster_0. DVM transactions are created during address remapping of the recovery process due to a fault in shared memory. At this time, if a specific element enters a reset state due to another failure, it may fail to return the DVM response and lead to recovery failure.

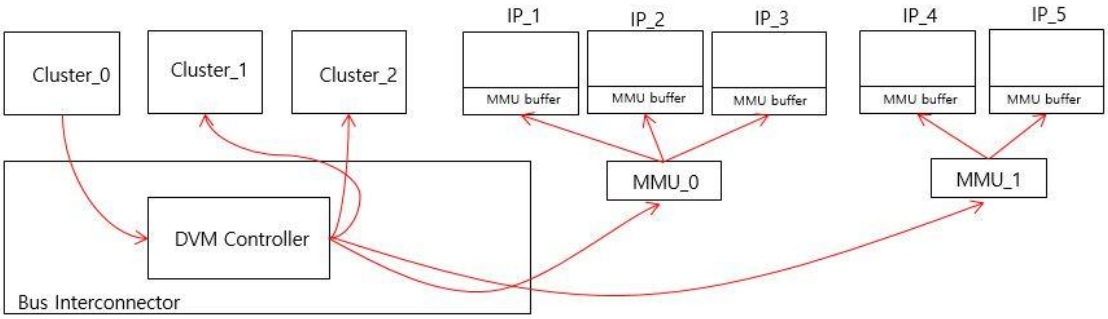



Figure 4. DVM transaction flow diagram

- Exclusive access

When a PE attempts to perform an exclusive access to an exclusive target region, the global exclusive monitor on the system bus allows only one PE (CPUCL0 in figure 5 below) to access the region. During that time, the exclusive target region is locked, and all other PEs that attempt to access it are blocked until the exclusive access is complete and the lock released. If an error occurs in the exclusive target area during exclusive access, the blocked PEs may wait indefinitely for the lock to be released.

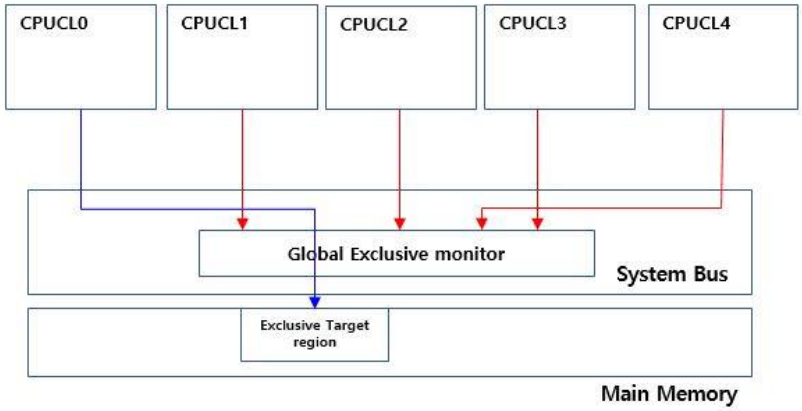



Figure 5. Exclusive access flow diagram

In addition, CPU cluster power down and dynamic CPU clock level changes are also used as coupling factors that can cause HW coherency protocol errors.

The PSS model allows us to create a coupling-factor scenario and inject an error per the selected DFI, while in parallel running interference stimulus which might disrupt the recovery process.

## IV. GENERATING THE SCENARIO USING PSS

PSS allows you to create all the required infrastructure below, but you can also save some time and use an existing library as we did. Among other topics, the library provides built-in actions to generate coherency scenarios for multi-core systems, such as false sharing, true sharing, RAS error injection, and more. Our project-specific tests leverage the built-in actions and add additional system-specific contents.

### *A. Traffic Generation*

Consider for example the following system-level scenario we would like to generate:

- Run false-sharing scenario with:
  - 85% of the cases do read-increment-write actions

- 15% of the cases do an error injection action, selecting from these kinds: RAS error, MMU translation fault, ECC memory error, or CPU internal exception.

The code example in figure 6 shows the declaration of a new PSS action, which selects either valid traffic, or injection of one of several error kinds. The PSS tool will randomize the selection of one of the choices, each time the action is traversed within a scenario. The action shows the use of both library contents, as well as project-specific actions:

```
action activity_selection {
  activity {
    // Randomly select one of the choices:
    select {
      // Valid coherency actions:
      [85]: do read_increment_write;

      // Error injection options:
      // - RAS error injection (library)
      [5]: do cdn_coherency_ops_c::ras_core_error_inject;

      // - MMU translation fault generation
      [5]: do core_remap_ttbr_error_inject;
      // - Uncorrectable ECC error injection
      [5]: do ecc_memory_error_inject;
    }
  }
}
```

Figure 6. Activity selection

The code example in figure 7 illustrates the use of the utility action shown previously, by embedding it within a false-sharing action provided in the library:

```
// Action: false_sharing_with_err_injection
//
// Invokes the library's false-sharing action, with the
// activity_selection as the embedded action to be executed
// by each core.

action false_sharing_with_err_injection {
  activity {
    do cdn_coherency_ops_c::false_sharing_random_operator
              < activity_selection >;  // (*)
  }
}
```

Figure 7. False sharing with error injection

(*) The code example above shows the use of template types, defined in PSS 1.1

Figure 8 shows a sample generated scenario, in the form of UML diagram, produced by the PSS tool. It illustrates the steps performed by two cores, each reaching a point where it injects an MMU page fault:

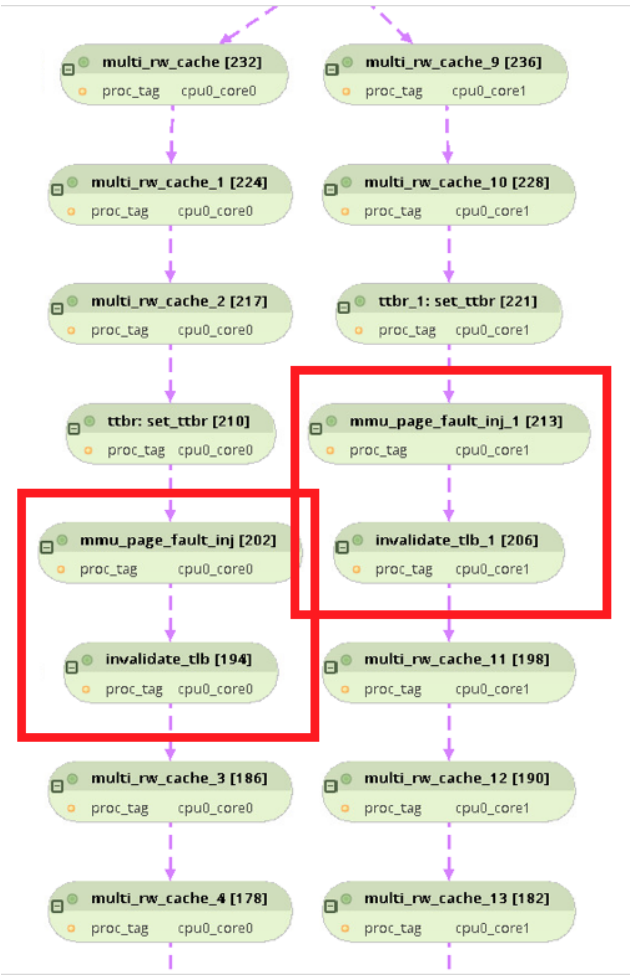


Figure 8. UML diagram of the generated scenario

Library actions such as the false-sharing shown above handle many aspects that are vital to generate a valid system-level test. These aspects include:

- Core selection, based on a system configuration table which lists existing cores and their attributes
- Synchronization points between the activity of multiple cores
- Memory management and memory allocation based on a system configuration table which lists existing memories and their attributes

Using similar combinations of library actions and our own project-specific contents, we define even more complex tests such as the following, shown in figure 9:

- Run the above false-sharing scenario with fault injection.
- Concurrently, run interference scenarios:
    - Clock frequency change
    - Powering cores down/up
    - Exclusive access

```
action false_sharing_with_err_injection_and_interference {
  activity {

    parallel {
      do false_sharing_with_err_injection;

      repeat (10) {
        select {
          do change_frequency;
          do cdn_coherency_ops::power_activity;
          do cdn_coherency_ops::exclusive_cache_access;
        }
      }
    }
}
```

Figure 9. False sharing with error injection and interference stimuli

This results in significant, randomized scenarios that exercise many aspects of the system, and thus improve coverage and the chances of uncovering bugs. Figure 10 shows a UML diagram of one such generated scenario:

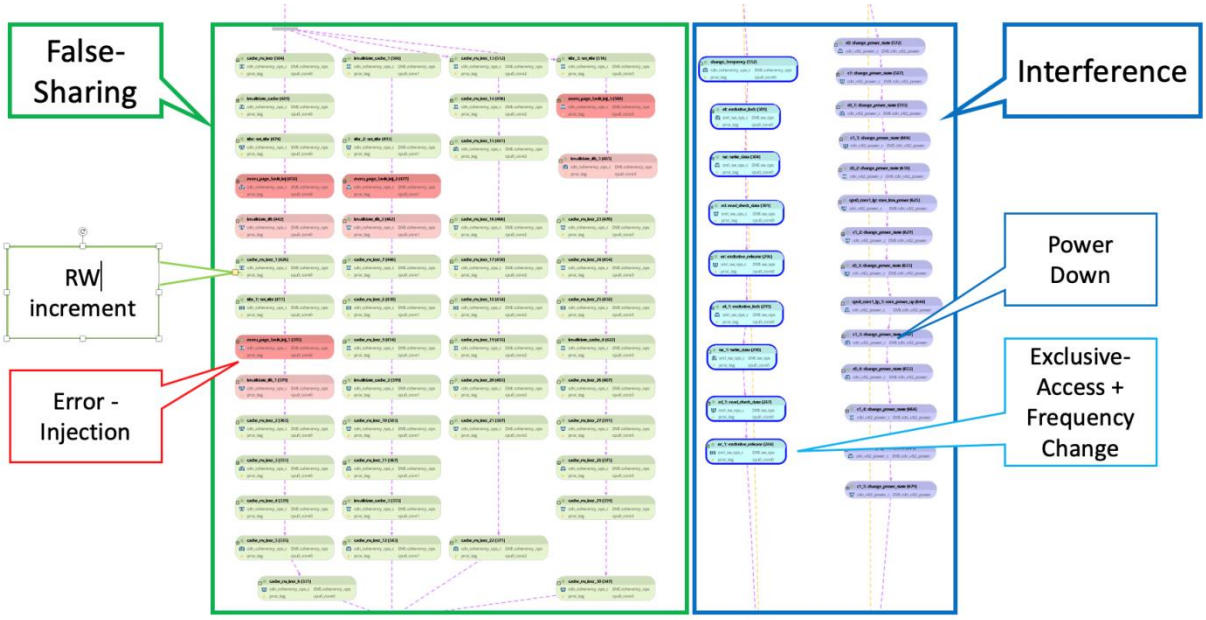



Figure 10. Generated dependent failure scenario

*B. Self-Checking and Coverage*

When using a constrained-random approach, two additional elements are key:

1. Self-checking to allow the testbench itself to determine whether the test passed given the randomized scenario
2. Coverage collections to record what combinations/flows were actually exercised, and which are still pending (uncovered) in the verification plan.

The self-checking aspect is built into our PSS model and the library. For example, all data that is written to memory is also read later during the test to make sure it is valid.

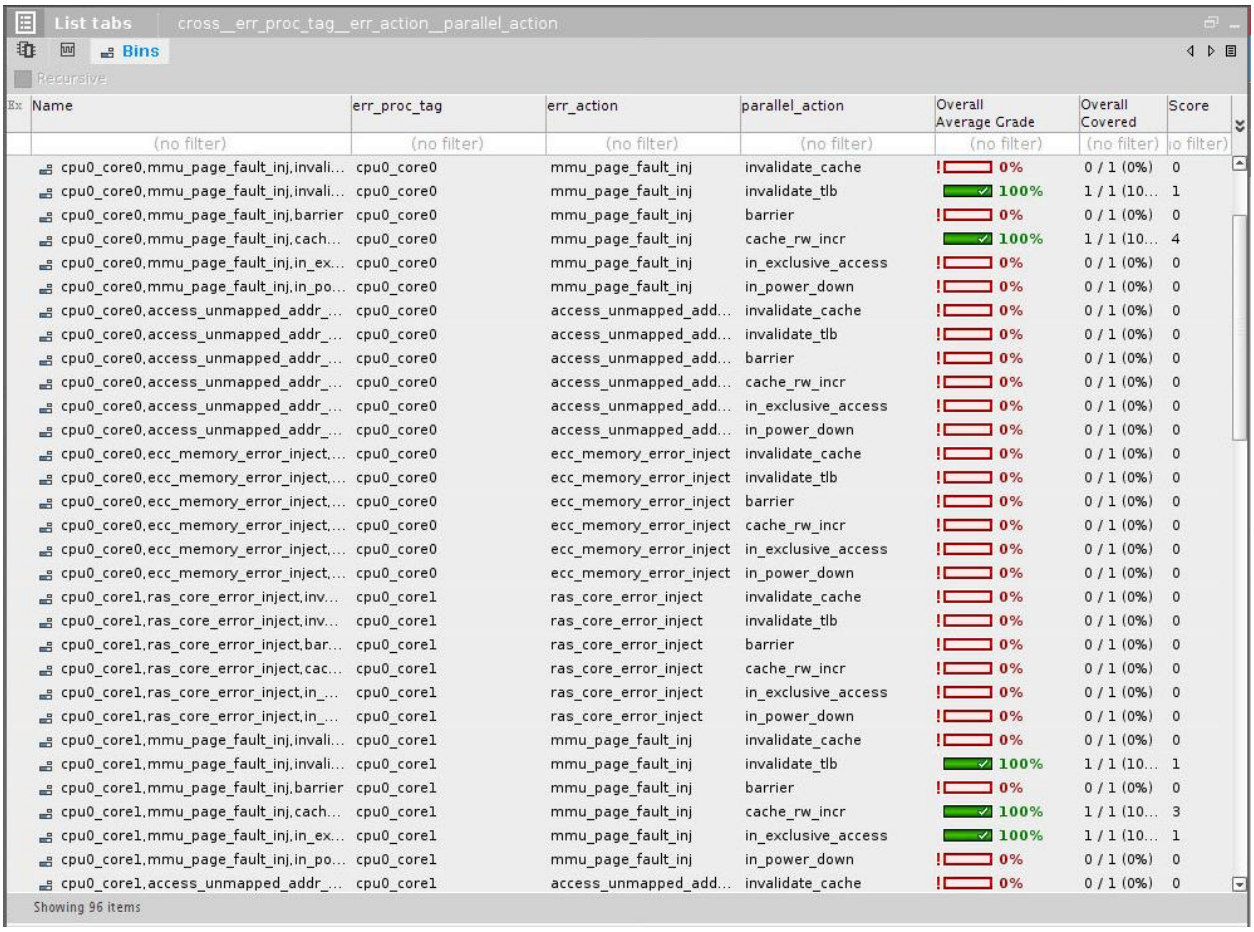
Similarly, PSS functional coverage is used to record the relevant traffic attributes, along with information such as which combinations of error kinds were tested during which types of valid activity scenarios.

The following example shows a PSS coverage group definition:

```
// Sample functional-coverage declaration to track error
// injection and handling

action monitor_error_handling {
  covergroup {
        // processor injecting the error
    coverpoint err_proc_tag;
        // type of error
        coverpoint err_action;
        // type of valid activity during that time
    coverpoint parallel_action;

    // cross coverage of the above
    proc_x_err_x_traffic:
            cross err_proc_tag, err_action, parallel_action;

 } cov_error_handling
```

Figure 11. PSS coverage definition

A sample coverage report is shown below, illustrating the cross-coverage item with some of its bins. Only few of the bins are shown as covered, based on the data collected in a single run:




| Name | err_proc_tag | err_action | parallel_action | Overall Average Grade | Overall Covered | Score |
|---|---|---|---|---|---|---|
| (no filter) | (no filter) | (no filter) | (no filter) | (no filter) | (no filter) | (no filter) |
| cpu0_core0,mmu_page_fault_inj,invali... | cpu0_core0 | mmu_page_fault_inj | invalidate_cache | 0% | 0 / 1 (0%) | 0 |
| cpu0_core0,mmu_page_fault_inj,invali... | cpu0_core0 | mmu_page_fault_inj | invalidate_tlb | 100% | 1 / 1 (10... | 1 |
| cpu0_core0,mmu_page_fault_inj,barrier | cpu0_core0 | mmu_page_fault_inj | barrier | 0% | 0 / 1 (0%) | 0 |
| cpu0_core0,mmu_page_fault_inj,cach... | cpu0_core0 | mmu_page_fault_inj | cache_rw_incr | 100% | 1 / 1 (10... | 4 |
| cpu0_core0,mmu_page_fault_inj,in_ex... | cpu0_core0 | mmu_page_fault_inj | in_exclusive_access | 0% | 0 / 1 (0%) | 0 |
| cpu0_core0,mmu_page_fault_inj,in_po... | cpu0_core0 | mmu_page_fault_inj | in_power_down | 0% | 0 / 1 (0%) | 0 |
| cpu0_core0,access_unmapped_addr_... | cpu0_core0 | access_unmapped_add... | invalidate_cache | 0% | 0 / 1 (0%) | 0 |
| cpu0_core0,access_unmapped_addr_... | cpu0_core0 | access_unmapped_add... | invalidate_tlb | 0% | 0 / 1 (0%) | 0 |
| cpu0_core0,access_unmapped_addr_... | cpu0_core0 | access_unmapped_add... | barrier | 0% | 0 / 1 (0%) | 0 |
| cpu0_core0,access_unmapped_addr_... | cpu0_core0 | access_unmapped_add... | cache_rw_incr | 0% | 0 / 1 (0%) | 0 |
| cpu0_core0,access_unmapped_addr_... | cpu0_core0 | access_unmapped_add... | in_exclusive_access | 0% | 0 / 1 (0%) | 0 |
| cpu0_core0,access_unmapped_addr_... | cpu0_core0 | access_unmapped_add... | in_power_down | 0% | 0 / 1 (0%) | 0 |
| cpu0_core0,ecc_memory_error_inject,... | cpu0_core0 | ecc_memory_error_inject | invalidate_cache | 0% | 0 / 1 (0%) | 0 |
| cpu0_core0,ecc_memory_error_inject,... | cpu0_core0 | ecc_memory_error_inject | invalidate_tlb | 0% | 0 / 1 (0%) | 0 |
| cpu0_core0,ecc_memory_error_inject,... | cpu0_core0 | ecc_memory_error_inject | barrier | 0% | 0 / 1 (0%) | 0 |
| cpu0_core0,ecc_memory_error_inject,... | cpu0_core0 | ecc_memory_error_inject | cache_rw_incr | 0% | 0 / 1 (0%) | 0 |
| cpu0_core0,ecc_memory_error_inject,... | cpu0_core0 | ecc_memory_error_inject | in_exclusive_access | 0% | 0 / 1 (0%) | 0 |
| cpu0_core0,ecc_memory_error_inject,... | cpu0_core0 | ecc_memory_error_inject | in_power_down | 0% | 0 / 1 (0%) | 0 |
| cpu0_core1,ras_core_error_inject,inv... | cpu0_core1 | ras_core_error_inject | invalidate_cache | 0% | 0 / 1 (0%) | 0 |
| cpu0_core1,ras_core_error_inject,inv... | cpu0_core1 | ras_core_error_inject | invalidate_tlb | 0% | 0 / 1 (0%) | 0 |
| cpu0_core1,ras_core_error_inject,bar... | cpu0_core1 | ras_core_error_inject | barrier | 0% | 0 / 1 (0%) | 0 |
| cpu0_core1,ras_core_error_inject,cac... | cpu0_core1 | ras_core_error_inject | cache_rw_incr | 0% | 0 / 1 (0%) | 0 |
| cpu0_core1,ras_core_error_inject,in_... | cpu0_core1 | ras_core_error_inject | in_exclusive_access | 0% | 0 / 1 (0%) | 0 |
| cpu0_core1,ras_core_error_inject,in_... | cpu0_core1 | ras_core_error_inject | in_power_down | 0% | 0 / 1 (0%) | 0 |
| cpu0_core1,mmu_page_fault_inj,invali... | cpu0_core1 | mmu_page_fault_inj | invalidate_cache | 0% | 0 / 1 (0%) | 0 |
| cpu0_core1,mmu_page_fault_inj,invali... | cpu0_core1 | mmu_page_fault_inj | invalidate_tlb | 100% | 1 / 1 (10... | 1 |
| cpu0_core1,mmu_page_fault_inj,barrier | cpu0_core1 | mmu_page_fault_inj | barrier | 0% | 0 / 1 (0%) | 0 |
| cpu0_core1,mmu_page_fault_inj,cach... | cpu0_core1 | mmu_page_fault_inj | cache_rw_incr | 100% | 1 / 1 (10... | 3 |
| cpu0_core1,mmu_page_fault_inj,in_ex... | cpu0_core1 | mmu_page_fault_inj | in_exclusive_access | 100% | 1 / 1 (10... | 1 |
| cpu0_core1,mmu_page_fault_inj,in_po... | cpu0_core1 | mmu_page_fault_inj | in_power_down | 0% | 0 / 1 (0%) | 0 |
| cpu0_core1,access_unmapped_addr_... | cpu0_core1 | access_unmapped_add... | invalidate_cache | 0% | 0 / 1 (0%) | 0 |

Figure 12. Sample coverage report

Using randomization and combining the coverage results of multiple runs allows us to cover all the relevant coverage goals. This is a significant time-saver compared to manually creating (and maintaining over time) a large number of directed tests to achieve the same level of coverage.

## V. FAULT TOLERANCE REPORT (FTR) GENERATION

Failure Mode and Effects Analysis (FMEA) determines all possible ways a system component can fail and determines the effect of such failures on the system. The DFI is selected based on the pre-defined FMEA items as shown below.

| | FMEA | | | | | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Name / Function | | | | | | | | | | Recommended Action(s) | | Responsibility & Target Completion Date | Action Results | | | |
| ID | Requirements | Potential Failure Mode(s) | Potential Effect(s) of Failure | Sev | Potential Cause(s) of Failure | Occur | Current Design Controls (Prevention) | Current Design Controls (Detection) | Det | Preventive Action(s) | Detection Action(s) | | Actions Taken | Sev | Occr | Det |
| M001 | Memory scheduler:ECC_logic | ECC error - double bit | Loss of basic functionality | | memory cell defect due to the electostatic | | Experienced Designer / Review | Simulation | | | interrupt/ error response | | system reboot/ masking problem area | | | |
| M002 | Memory scheduler:AXI_Interface | SFRs not writeable | Adress Mapping not correct / Loss of basic functionality | | AXI Slave Interface wrongly implemented/ SW fault | | Reuse / Family Concept | Simulation | | | error response | | system reboot/ masking problem area | | | |

Figure 13. Failure Mode and Effect Analysis (FMEA) example

Once the DFI is determined, the PSS selects an interference stimulus, which can be a coupling factor, to create a dependent failure scenario. Each scenario prints out the following information when simulation completes:
- Injected fault information

Shows which type of faults were injected at which address. It also shows the time when a fault was injected and the time when recovery process was completed.

- Executed interference action information

Shows which interference actions were performed to delay recovery process after fault injection. Another error injection can also be used as an interference action.

- Maximum Fault Tolerance Time Interval (FTTI) information

FTTI is the time taken to recover from an uncontrolled state due to a fault. The maximum FTTI can be measured for hundreds of fault conditions generated by the PSS tool, to determine which coupling factor affects the recovery process the most.

- External recovery monitor [8]

Provides recovery monitor information for situations where the system fails to recover from the generated failures.

The PSS tool generated error-handling scenarios with hundreds of different conditions, and by gathering the above information printed for each scenario, we created a Fault Tolerance Report (FTR):

**Fault Tolerance Report (FTR)**

| Fault Injection | | | | | Interference stimulus | | | | | Simulation result | | Scenario info | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| FMEA_ID | type | target | expected failures | FTR_ID | stimulus_1 | stimulus_2 | stimulus_3 | stimulus_4 | stimulus_5 | Recovery result | Fault Tolerance Report (FTTI) | Scenario name | Seed number |
| M001 | ECC error | DRAM | error interrupt/ error response | M001_1 | false sharing access | | | | | done | 80 | dram_1_ecc_1 | 3523 |
| | | | | M001_2 | false sharing access | exclusive access | | | | done | 100 | dram_1_ecc_2 | 3475 |
| | | | | M001_3 | false sharing access | exclusive access | MMU page remap | | | done | 105 | dram_1_ecc_3 | 2531 |
| | | | | M001_4 | false sharing access | exclusive access | MMU page remap | cluster powerdown | | done | 105 | dram_1_ecc_4 | 3767 |
| | | | | M001_5 | false sharing access | exclusive access | MMU page remap | cluster powerdown | DFS level change | done | 110 | dram_1_ecc_5 | 8236 |
| | | | | M001_6 | exclusive access | | | | | done | 50 | dram_1_ecc_1 | 3257 |
| | | | | M001_7 | exclusive access | MMU page remap | | | | done | 55 | dram_1_ecc_2 | 3278 |
| | | | | M001_8 | exclusive access | MMU page remap | false sharing access | | | done | 90 | dram_1_ecc_3 | 4291 |
| | | | | M001_9 | exclusive access | MMU page remap | false sharing access | DFS level change | | done | 93 | dram_1_ecc_4 | 3982 |
| | | | | M001_10 | exclusive access | MMU page remap | false sharing access | DFS level change | cluster powerdown | done | 97 | dram_1_ecc_5 | 7218 |

Figure 14. Generated Fault Tolerance Report (FTR)

As shown in figure 14, the FTR provides information to identify which interference stimulus interfered with the recovery operation in the fault injection scenario where the maximum FTTI was measured. The FTRs for each error generated in this way are reflected in the DFA result as shown in figure 15 to prove that safety is guaranteed under various error conditions.

**Dependent Failure Analysis (DFA)**

| FMEA_ID | Element | Redundant element | Functional dependency (Cascading failure) | Dependent failures initiators (Common cause failures) | | | DFA | | Responsible Person | Status |
|---|---|---|---|---|---|---|---|---|---|---|
| | Short name and description | Short name and description | Description | Systematic faults | Shared resources | Single Physical root cause | Measure for fault (A)voidance or (C)ontrol | Verification method | | |
| M001 | Memory scheduler:ECC_logic | CPU cluster 0/1/2/3/4 | False sharing | | Fault injection for generate ECC error from shared DRAM region | | interrupt / error response | **simulation : FTR_ID M001_1** | | |
| | Memory scheduler:ECC_logic | CPU cluster 0/1/2/3/4 | False sharing / exclusive access | | Fault injection for generate ECC error from shared DRAM region | | interrupt / error response | **simulation : FTR_ID M001_2** | | |
| | Memory scheduler:ECC_logic | CPU cluster 0/1/2/3/4 | False sharing / exclusive access / MMU page remap | | Fault injection for generate ECC error from shared DRAM region | | interrupt / error response | **simulation : FTR_ID M001_3** | | |
| | Memory scheduler:ECC_logic | CPU cluster 0/1/2/3/4 | False sharing / exclusive access / MMU page remap / cluster powerdown | | Fault injection for generate ECC error from shared DRAM region | | interrupt / error response | **simulation : FTR_ID M001_4** | | |
| | Memory scheduler:ECC_logic | CPU cluster 0/1/2/3/4 | False sharing / exclusive access / MMU page remap / cluster powerdown / DFS level change | | Fault injection for generate ECC error from shared DRAM region | | interrupt / error response | **simulation : FTR_ID M001_5** | | |

Figure 15. Dependent Failure Analysis (DFA) report include FTR information

## VI. RESULTS & LESSONS LEARNED FROM AN ACTUAL PROJECT

In this paper, we performed the DFA defined in ISO 26262 at the simulation level for the effect of coherence on the system when a fault occurs in the shared memory region. Through this, it was possible to systematically analyze the possible dependent failures and prepare the safety specifications.

We have found that this requirement analysis and verification methodology introduced in ISO 26262 can be usefully applied to the general SoC verification process as well as functional safety. The PSS model reusability and constrained-random test generation made it easy to generate tests with various conditions defined in safety requirements. We find PSS to be an optimized methodology for reproducing any particular target condition. These advantages will make PSS an essential methodology for safety verification.

## VII. ACKNOWLEDGMENT

The work described in this paper was done using Cadence's Perspec System Verifier[9], and the built-in System Modeling Library (SML). We also used Cadence's vManager [10] to manage and analyze simulation results.

## REFERENCES

[1] ISO-26262-11:2018 "Road vehicles –Functional safety- part11, 4.7 Semiconductor dependent failures analysis"
[2] ISO-26262:2018 "Road Vehicles – Functional Safety"; https://www.iso.org/obp/ui/#iso:std:iso:26262:-1:ed-2:v1:en
[3] PSS: Accellera, Portable Test and Stimulus: https://www.accellera.org/downloads/standards/portable-stimulus
[4] IEC-61508:2010 "Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems"; https://www.iec.ch/functionalsafety/standards/page2.htm
[5] ISO-26262-11:2018 "Road vehicles –Functional safety- Introduction"
[6] ISO-26262-11:2018 "Road vehicles –Functional safety- part11: Guideline on application of ISO 26262 to semiconductors"
[7] ARM, DDI 0587C.b "Reliability, Availability, and Serviceability(RAS) Specification: Armv8, for the Armv8-A architecture profile"
[8] Jang, M., Kim, J., Chung, H., Huynh, P., Shai, F. (DVCon 2019) Coherency Verification & Deadlock Detection Using Perspec/Portable Stimulus
[9] Cadence, Perspec System Verifier https://www.cadence.com/content/cadence-www/global/en_US/home/tools/system-design-and-verification/software-driven-verification/perspec-system-verifier.html
[10] Cadence, vManager Metric-Driven Signoff Platform https://www.cadence.com/content/cadence-www/global/en_US/home/tools/system-design-and-verification/planning-and-management/incisive-vmanager-solution.html