Are You Safe Yet? Safety Mechanism Insertion and Validation

Ping Yeung, Jin Hou, Vinayak Desai
Mentor Graphics, Fremont

Jacob Wiltgen
Mentor Graphics, Longmont

Abstract

As functional safety becomes increasingly important in today's industrial and automotive designs, many legacy designs have to be “upgraded” to meet the safety goal of the system. An efficient approach is to use safety synthesis and formal verification to incorporate a safety architecture into the design. The flow can consist of these major steps: 1) explore areas of the design where better fault detections are required, 2) introduce the right safety mechanisms into the design with safety synthesis, 3) validate the design changes with formal verification, and 4) perform formal fault injection to measure the diagnostic coverage.

Introduction

FMEDA (Failure Mode Effect and Diagnostic Analysis) evaluates the safety architecture with its collection of safety mechanisms and calculates the safety performance of the system. In Part 5 of the ISO 26262 [1] specification, a hardware architecture needs to be evaluated against the requirements for fault handling. It requires that the probabilities of random hardware failures are rigorously analyzed and quantified via a set of objective metrics [2]. If any of the architectural metrics fail to meet the criteria defined for the product’s Automotive Safety Integrity Level (ASIL), design teams will be mandated to re-evaluate the component’s safety concept, improve the existing safety mechanisms, and if necessary introduce new safety mechanisms.

To improve diagnostic coverage, a practical approach is to incorporate a collection of safety mechanisms into the design so that the number and types of faults detected can be improved. This is best to be done at the register transfer level, where functional verification can be performed efficiently. The process can consist of these major steps:

1) explore areas of the design where better fault detections are needed [3],
2) introduce safety mechanisms that have the right tradeoff for the RTL structures
3) validate the design changes with sequential logic equivalence checking (SLEC)
4) perform fault injection with a formal-based methodology to measure the diagnostic coverage

Safety exploration

The goal of safety exploration [3] is to identify the optimal safety architecture and safety mechanisms. Safety mechanisms come in a variety of flavors, each with its own level of effectiveness in detecting random hardware faults. During safety exploration, a series of “what-if” scenarios are performed to understand the impact of different safety mechanisms on the design, especially with respect to power, area, performance, safety metrics, and diagnostic coverage. This exploration is performed without modifying the design so that simultaneous analyses can be performed quickly and efficiently. If it is done correctly, safety exploration will help design teams meeting the safety targets successfully after safety insertion and safety verification are completed.

Safety Mechanisms Insertion

Modifying a legacy design to introduce safety mechanisms can be an error-prone process. The situation becomes worse if the initial safe mechanism does not meet the safety goal, and a different mechanism has to be used. Hence, safety synthesis is a new approach to augment design structures to improve fault tolerance. To create a safety architecture for legacy designs, safety synthesis introduces two types of safety mechanism automatically. One is used at the register-level, and the other is used at the module level.

Register-level insertion is more surgical and inserts safety mechanisms at the flip-flop level, such as register duplication and parity. This approach is commonly used to protect control and state machine structures. Some Register-level safety mechanisms include:

1. Parity generation and checking for critical control elements.
2. Double modular redundancy for a selected list of registers.
3. Triple modular redundancy for a selected list of registers.
4. Error correction, and single-error correction with double-error detection for banks of registers.
5. Protocol checking ensures valid state transitions for finite state machines

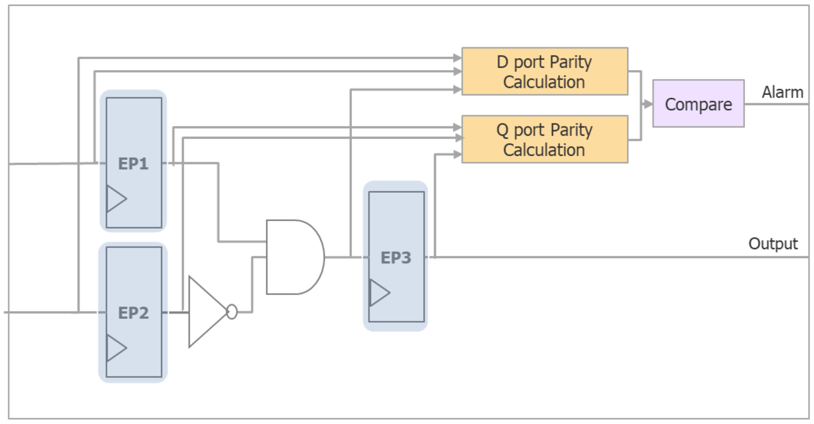


Figure 1, Parity generation and checking for control registers

Safety synthesis can add parity checking to all or a list of special registers in a module, Figure 1. Error-correction code (ECC), single-error correct with double-error detect (SECDED), can also be used for a bank of registers. In this mode, safety synthesis groups the registers by name and adds ECC by creating a syndrome. Later, the syndrome is used to determine whether there is an error and to recover the error from it. For finite state machines, safety synthesis will elaborate the valid state space and the state transition matrix to build a protocol checking safety mechanism.

Module-level insertion creates redundancy-based safety mechanisms at the instance level. Some module-level safety mechanisms include:

1. Double modular redundancy along with lockstep checker.
2. Triple modular redundancy along with lockstep checker and majority voting
3. Input and output parity checking on groups of interface signals
4. Memory parity generation and checking

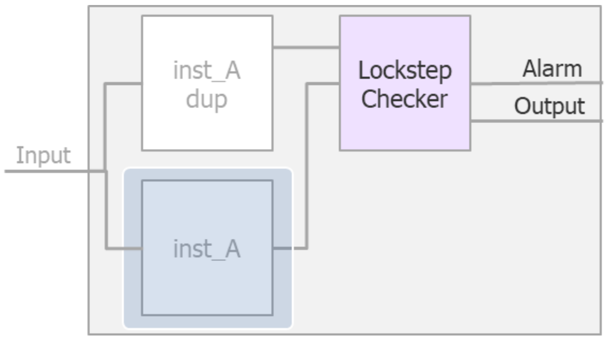


Figure 2, Double modular redundancy along with lockstep checker.

In Figure 2, safety synthesis creates a second instance of the module, and makes appropriate connections for all the outputs and inputs between the first and second instance, along with the lockstep checker.

For each safety mechanism, there are tradeoffs concerning efficiency, fault detection latency, and area overhead. Figure 3 shows where safety mechanisms can be used in a safety-critical design.

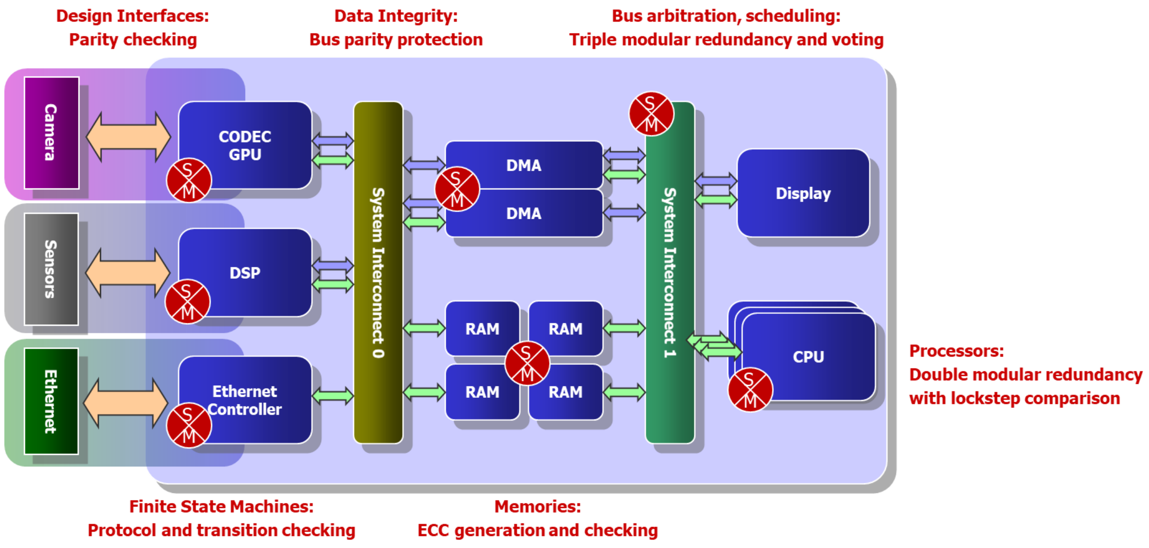


Figure 3, Example Safety Mechanisms for a Safety-Critical Design

A high-level architecture is shown in Figure 3. For the design interfaces, parity checks can be performed to ensure accurate data transmission between the interface modules and the interface controllers inside the design. Once the data are inside the design, they can be protected with data parity on the buses and error-correction code (ECC) in storage elements. On-chip bus transactions can be observed by dedicated bus monitors. Critical control components such as FSMs and arbitration logic will best be protected with triple module redundancy and majority voting. Central and embedded processors can be protected with double modular redundancy along with lockstep checkers.

Sequential Logic Equivalence Checking (SLEC)

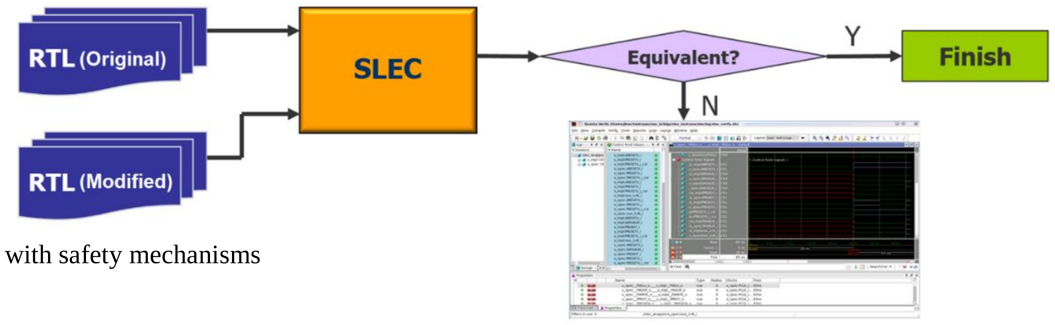


Figure 4, Sequential Logic Equivalence Checking

Sequential Logic Equivalence Checking (SLEC) formally verifies that two designs are functionally equivalent using formal verification technology, Figure 4. The implementation of the two designs can be different as long as the outputs are always the same. When SLEC proves the equivalence of two signals, one from each design, the two signals are equivalent for *all* inputs and for *all* times. If any signal pair are not equivalent, SLEC automatically generates an error trace using waveforms to show the cause-and-effect of the problem. When all outputs of the two designs are proven equivalent, we can be confident that the two designs are functionally equivalent.

Hence, SLEC can be used to verify:

- Safety Mechanism Insertion, ensuring that the functionality of the original design is not changed by the addition of the safety mechanisms (SMs)
- Safety Mechanism Operation, ensuring that the inserted functional safety mechanisms are working as designed.

Safety Mechanism Insertion Verification

After the insertion of safety mechanisms, it is important to ensure that the designs before & after insertion are functionally equivalent. Instead of verifying the functionality of the safety mechanism instrumented design using simulation regression, sequential logic equivalence checking (SLEC) is a more direct and comprehensive approach.

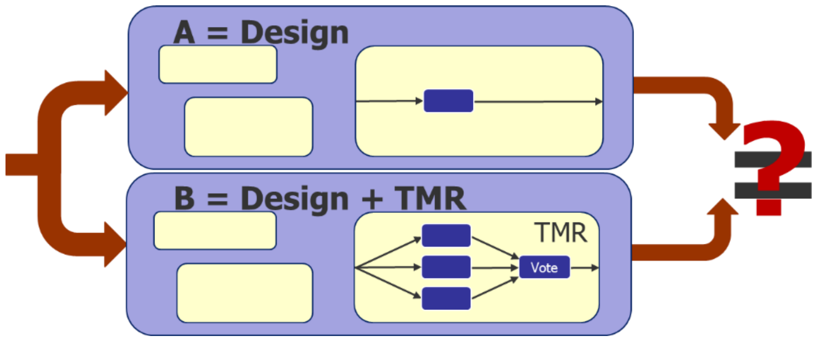


Figure 5, Safety Mechanism Insertion Verification with SLEC

In Figure 5, triple modular redundancy (TMR) is used as the safety mechanism to protect the design. By comparing Design A (without safety mechanism) and Design B (with TMR), SLEC can mathematically prove that the TMR has been correctly inserted into the design. This approach gives us better confidence as functional simulation can only simulate limited numbers of input sequences to verify the operation of the TMR.

Safety Mechanism Operation Verification

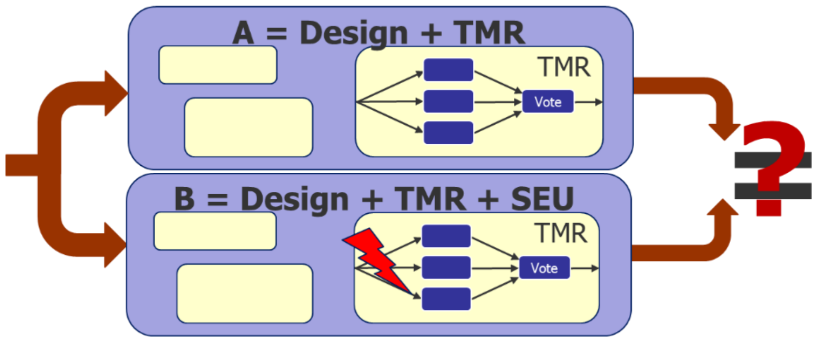


Figure 6, Safety Mechanism Operation Verification with Formal-based Fault Injection

Is the safety mechanism working? After the safety mechanisms have been introduced correctly, fault injection can be performed with a formal-based methodology [4][5]. Failure Mode and Effects Analysis (FMEA) can determine whether the safety mechanism is sufficient. The formal methodology is set up to inject both stuck-at and transient faults into a design, clock the fault through the design's state space, and see if the fault is propagated, masked, or detected by the safety mechanisms.

As depicted in Figure 6, a golden (no-fault) model and a fault injected model are used to perform on-the-fly fault injection and result analysis. By instantiating a design with a copy of itself, all legal input values are automatically specified for SLEC, just as a golden reference model in simulation predicts all expected outputs for any input stimulus. By comparing a fault injected design with a copy of itself without faults, the formal tool checks if there is any possible way for the fault to either escape to the outputs or go undetected by the safety mechanism.

The steps of the fault injection with SLECFS can be summarized as:

1. Specify two copies of the original design for SLEC. The input ports will be constrained together, and the output comparison will be checked automatically.
2. Run SLEC to identify any design elements — such as memory black boxes, unconnected wires, un-resettable registers, etc. — that will cause the outputs to be different. Constraints are added to synchronize them.
3. Use the fault list to automatically specify possible injection points in the faulty design. Normally, we focus on the single point faults; i.e., one fault is injected to check for equivalency.
4. Based on the fault list and the comparison points, SLEC automatically identifies and remove any redundant blocks and logic.
5. Run SLEC concurrently on multiple servers. The multi-core approach has significant performance advantage as multiple outputs can be checked on multiple cores concurrently.
6. If an injected fault has caused a failure at the comparison point, a waveform of the counter-example that captures the fault injection and propagation sequence can be generated for debugging.

Results

This methodology has been used to incorporate a safety architecture in the memory sub-system of an AMBA-based design. After safety exploration, safety synthesis is performed to insert a safety mechanism, double modular redundancy, into the memory sub-system. SLEC is used to verify the equivalence between the original design and the "safe" design to make sure the functionality of the "safe" design was not altered.

The tool, Austemper Annealer [7], was used to perform safety synthesis by duplicating part of the design for double modular redundancy. Figure 7 shows the "safe" design and the setup of verifying the double modular redundancy safety mechanism using SLEC. We not only compare the outputs of the original design and the "safe" design but also compare the outputs of the original design and the outputs of the two instances in the "safe' design to make sure that the two instances behavior the same as the original design.

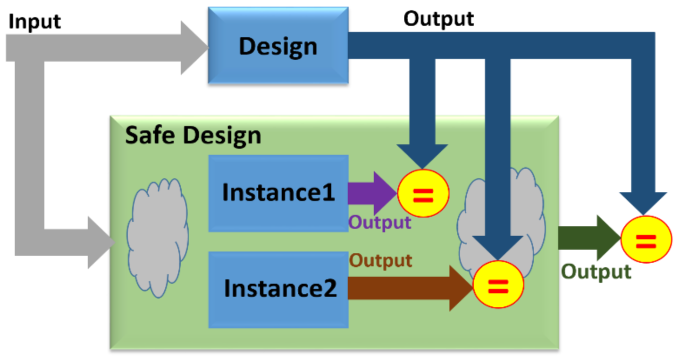
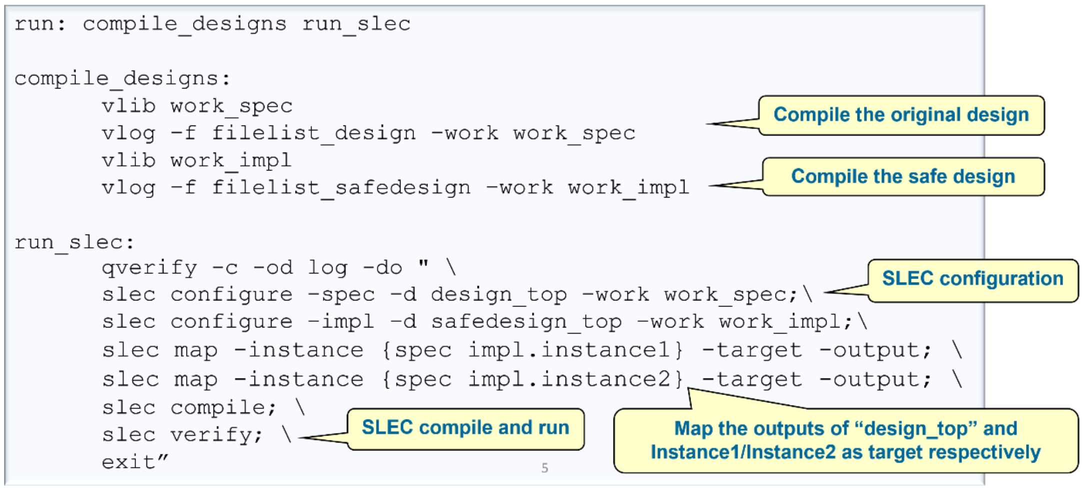


Figure 7. The flow of verifying double module redundancy with SLEC

The tool, Questa SLEC [8], was used for SLEC verification. The script to run Questa SLEC for verifying the safe design with double module redundancy is shown in Figure 8. From the script, we can see that to run Questa SLEC is simple: first compile the two designs, then configure the original design as specification and the safe design as implementation. Questa automatically maps the inputs of the two designs as assumptions and drives the inputs the same; it also automatically maps the outputs of the two designs as target for formal engines to verify their equivalence. To verify the equivalence between the original design and the two instances in the safe design, we need to add the two “slec map” commands in the script that map the outputs of the original design and the outputs of instance1 or instance2 as target respectively.



```
run: compile_designs run_slec

compile_designs:
      vlib work_spec
      vlog -f filelist_design -work work_spec
      vlib work_impl
      vlog -f filelist_safedesign -work work_impl

run_slec:
      qverify -c -od log -do " \
      slec configure -spec -d design_top -work work_spec;\
      slec configure -impl -d safedesign_top -work work_impl;\
      slec map -instance {spec impl.instance1} -target -output; \
      slec map -instance {spec impl.instance2} -target -output; \
      slec compile; \
      slec verify; \
      exit”
```

Compile the original design
Compile the safe design
SLEC configuration
SLEC compile and run
Map the outputs of “design_top” and Instance1/Instance2 as target respectively
5

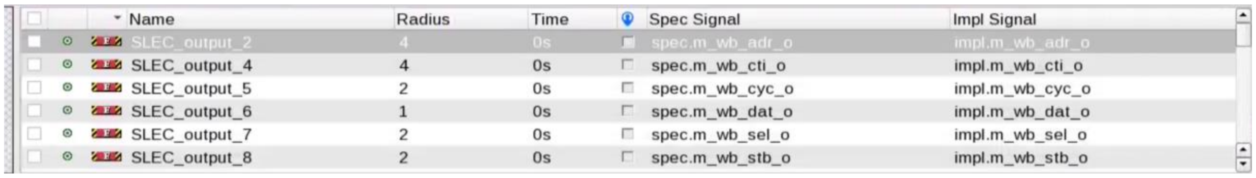
Figure 8. The script to run SLEC to verify the safe design

	Safety Mechanism	SLEC result		
		Proven	Fired	CPU time
AMBA Block A	module duplication	24	4	1s
AMBA Block B	register duplication	19	0	1s
AMBA Block C	ECC logic	16	0	36s
OpenRISC subsystem	module duplication	42	0	1s
Ethmac design (design bug)	module duplication and register duplication	31	23	2s
Ethmac design (bug fixed)	after bug fixed in safety mechanism	54	0	2s

Table 1. Results of safety mechanism verification for various designs

Table 1 summarized the many design blocks that have been “upgraded” with different safety mechanisms. For AMBA-based design block A and block B with duplication insertions, we have verified the equivalence between the outputs of the original block and the modified (original+safety mechanism) block. For block C with ECC insertion, we have verified the equivalency between the outputs of the original design and the modified design with ECC insertion.

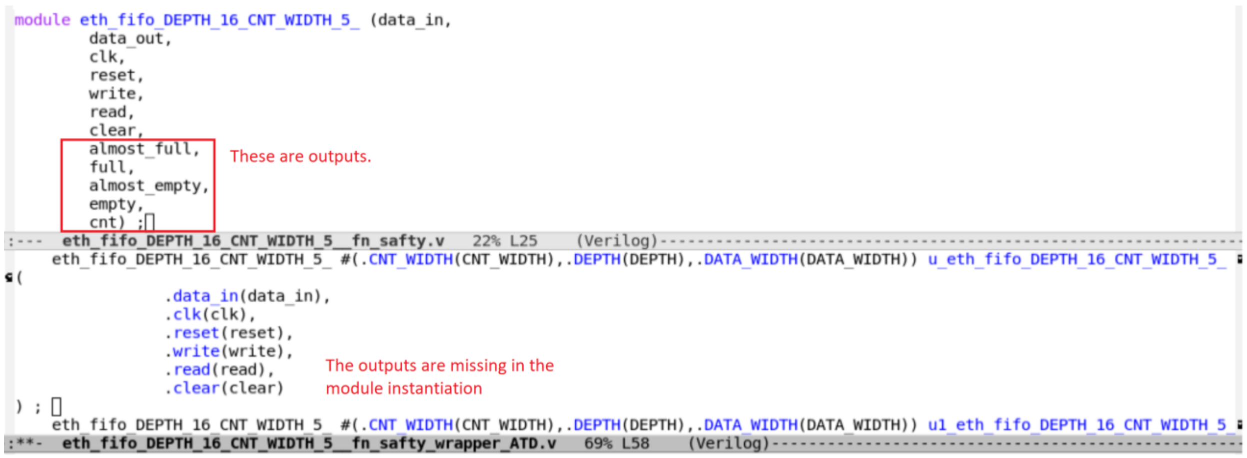
We had also applied this methodology to a sub-system of OpenRISC 1200 design and ethmac design. For example, the first run of comparing the original Ethmac design and the updated version with injected safety mechanism found 23 firings, including output non-equivalence and instance duplication non-equivalence. The results from Questa SLEC is shown in Figure 9 below.



```
Name            Radius  Time  Spec Signal        Impl Signal
SLEC_output_2   4       0s    spec.m_wb_adr_o    impl.m_wb_adr_o
SLEC_output_4   4       0s    spec.m_wb_cti_o    impl.m_wb_cti_o
SLEC_output_5   2       0s    spec.m_wb_cyc_o    impl.m_wb_cyc_o
SLEC_output_6   1       0s    spec.m_wb_dat_o    impl.m_wb_dat_o
SLEC_output_7   2       0s    spec.m_wb_sel_o    impl.m_wb_sel_o
SLEC_output_8   2       0s    spec.m_wb_stb_o    impl.m_wb_stb_o
```

Figure 9. SLEC output non-equivalence violations in the Ethmac design

Since the firings were marked with warnings (symbol F), we checked the design schematic and found floating signals (empty, cnt, etc.) in the updated design, depicted in Figure 10. We debugged the source code and found the errors in the module instantiations. After the bug was fixed, and rerun SLEC using the new injected safety mechanism, all equivalence targets were proven.



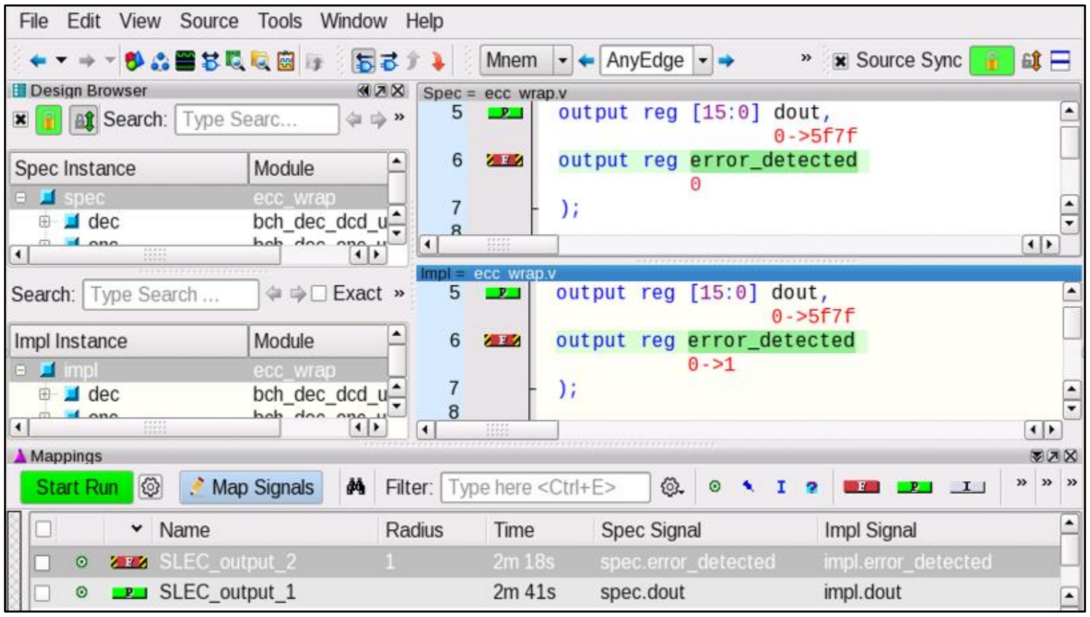
```
module eth_fifo_DEPTH_16_CNT_WIDTH_5_ (data_in,
        data_out,
        clk,
        reset,
        write,
        read,
        clear,
        almost_full,
        full,
        almost_empty,
        empty,
        cnt) ;
:---  eth_fifo_DEPTH_16_CNT_WIDTH_5__fn_safty.v    22% L25     (Verilog)------------------------------------
    eth_fifo_DEPTH_16_CNT_WIDTH_5_ #(.CNT_WIDTH(CNT_WIDTH),.DEPTH(DEPTH),.DATA_WIDTH(DATA_WIDTH)) u_eth_fifo_DEPTH_16_CNT_WIDTH_5_
(
                .data_in(data_in),
                .clk(clk),
                .reset(reset),
                .write(write),
                .read(read),
                .clear(clear)
) ;
    eth_fifo_DEPTH_16_CNT_WIDTH_5_ #(.CNT_WIDTH(CNT_WIDTH),.DEPTH(DEPTH),.DATA_WIDTH(DATA_WIDTH)) u1_eth_fifo_DEPTH_16_CNT_WIDTH_5_
:**-  eth_fifo_DEPTH_16_CNT_WIDTH_5__fn_safty_wrapper_ATD.v    69% L58     (Verilog)---------------------------
```

These are outputs.

The outputs are missing in the module instantiation

Figure 10, Safety Mechanisms with floating signals in the Ethmac design

After verifying that the safety mechanisms have been inserted correctly, faults were injected to test the operation of the safety mechanism. We were focusing on single-point faults (where one fault is injected at a time). Based on the fault list and the output comparison points, Questa SLEC will derive their dependency. Then, for each of the output comparison points, it will determine whether any fault can be introduced to cause the outputs to be different.



Spec Instance	Module
spec	ecc_wrap
dec	bch_dec_dcd_u

Impl Instance	Module
impl	ecc_wrap
dec	bch_dec_dcd_u

Spec = ecc_wrap.v
```
5   output reg [15:0] dout,
                     0->5f7f
6   output reg error_detected
                0
7   );
```

Impl = ecc_wrap.v
```
5   output reg [15:0] dout,
                     0->5f7f
6   output reg error_detected
                0->1
7   );
```

		Name	Radius	Time	Spec Signal	Impl Signal
☐	⊙ F	SLEC_output_2	1	2m 18s	spec.error_detected	impl.error_detected
☐	⊙ P	SLEC_output_1		2m 41s	spec.dout	impl.dout

Figure 11. Fault injection that had caused errors in the design

In Figure 11, “spec” of the original design while “impl” is the “upgraded” design with safety mechanisms. We can see that even though a fault had been injected into the design, the output of the design was still correct (impl dout the same as the spec dout). The ECC safety mechanism had recovered the data from the fault correctly. The error detection signal, error_detected, was asserted to alert the user of this situation. One the other hand, if an injected fault had caused a failure at the comparison point, a waveform of the counter-example that captures the fault injection and propagation sequence is generated for debugging.

References

[1] ISO 26262-5:2011 *Road vehicles Functional safety, Part 5: Product development at the hardware level*, https://www.iso.org/standard/51360.html
[2] Andrew Hopkins, *Silicon evolution for the automotive revolution*. ARM White Paper, 2019.
[3] Jacob Wiltgen, *Reducing Your Fault Campaign Workload Through Effective Safety Analysis*, Semi Engineering, Aug 2019.
[4] Avidan Efody, *Whose Fault Is It? Advanced Techniques for Optimizing ISO 26262 Fault Analysis.* DVCon 2016.
[5] Ping Yeung, et al., *Whose Fault Is It Formally? Formal Techniques for Optimizing ISO 26262 Fault Analysis.* DVCon 2018.
[6] Doug Smith, *It's Not My Fault! How to Run a Better Fault Campaign Using Formal*. Verification Academy, Jun 2018.
[7] Austemper Annealer User Guide, Mentor, A Siemens Business, 2019.
[8] Questa SLEC User Guide, Mentor, A Siemens Business, 2019.