International Journal of Computer Applications (0975 – 8887)
Volume 154 – No.8, November 2016

Formal based Verification to Build Safer Cars

Deva Phanindra Kumar
Analog Devices, Inc
RMZ Infinity
Bangalore

Shweta Pujar
Analog Devices, Inc
RMZ Infinity
Bangalore

Ranganayakulu Sri
Analog Devices, Inc
RMZ Infinity
Bangalore

ABSTRACT

Functional safety features are an essential part of automotive system-on-chip development. ISO26262 standard dictates ASIC development process in safety applications like airbag control, electronic stability control. This paper focuses on verification requirements and fault injection simulation requirement of ISO26262 standard. Verification of such ASICs requires much more than traditional UVM-SV functional verification. Prior to this effort, safety verification techniques involved injecting faults using tools like Certitude, Yogitech and validating safety mechanisms through functional simulations. In this paper, formal tool's ability to perform exhaustive breadth-first search to verify the functional safety features and thereby reducing time to market.

General Terms

Digital systems, automotive system on chip design verification, ISO26262.

Keywords

Formal verification, automotive, functional safety, ASIL.

1. INTRODUCTION

1.1 Functional Safety

The objective of automotive functional safety is to prevent risk of physical injury or death to people or damage to the property or to the environment. These functional safety systems in an automobile can be predominantly classified into active safety systems or passive safety systems. Active safety systems like electronic stability control, roll stability control prevents accidents from happening. Passive safety systems reacts after an accident to minimize damage like air bags. As one can see, any malfunction in these safety mechanisms like accidental deployment of an airbag can cause tremendous harm. Furthermore advancements in technologies like MEMS have resulted conversion of pure mechanical based safety systems to electronically controlled systems, hence posing new challenges to functional safety. Most of the modern automobiles are equipped with embedded electronic systems which include Electronic Controller Units (ECUs), electronic sensors, MEMS sensors, bus systems and software code. Due to the complex application in electrical, electronics and programmable electronics, the need to carry out detailed safety analyses which focuses on the potential risk of malfunction is crucial for automotive systems.

ISO 26262 standard is a functional safety standard for automotive applications. This standard evolved from IEC 61508 which caters to industrial safety applications. Both these standards span over entire life cycle of product development namely requirement specification, design, implementation, integration, verification, validation and configuration. ISO26262 standard describes methods to classify risk and specifies requirements on how to avoid, detect and control systematic design faults, in ASIC development. And also, how to detect random hardware faults that may occur in field due to ageing, temperature, voltage variations. This paper focuses on how to meet verification requirements of ISO26262 standard.

1.2 ISO26262 Verification Requirements and Challenges

Functional safety features called safety mechanisms (SM) are incorporated in automotive systems in compliance with ISO 26262 standard [1], [2]. These safety mechanisms are determined based on Automotive Safety Integrity Levels, ASILs of a product. ASIL requirement can range from 'A' to 'D'. 'A' being least stringent ASIL and 'D' being most stringent [3]. These safety mechanisms are expected to catch random hardware failures that may occur in field and report the health of sensor to the ECU. For example, if power on reset did not happen as expected device will be in undetermined state.

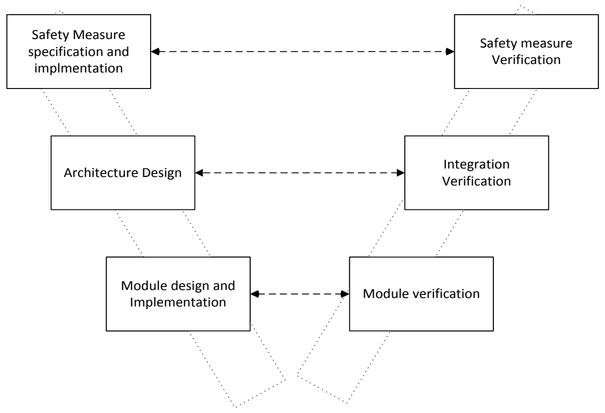

Safety Measure specification and implmentation
Safety measure Verification
Architecture Design
Integration Verification
Module design and Implementation
Module verification

Figure 1. V-model for functional safety verification

This condition needs to be recognized and should be notified to ECU through a health flag. ISO26262 standard recommends v-model for verification as depicted in figure 1

ISO26262 standard recommends similar V-model in entire product life cycle. However for the purpose of this paper V-model between design and verification is sufficient. First two items is similar to regular functional verification that is done in any product development. These two steps are needed to identify any systematic failures (bugs) in the product. Systematic failures doesn't depend on external factors and is repeatable in all devices. Third item in the diagram safety measure verification is required only in functional safety product.

The rigor of safety verification is dependent primarily on the ASIL (Automotive Safety Integrity Level) level. ASIL level for an automotive system is determined at the beginning of the development process. It is calculated on the basis of 3 factors: severity of failure, probability of exposure and possible controllability by a driver if a critical event occurs. The ASIL levels range from 'A' to 'D' with 'D' having highest severity and lowest controllability. Functional Safety verification for ASIL 'C' & 'D' systems mandates and ASIL 'A' & 'B' recommends fault insertion simulation. In Fault insertion simulations, design is modified to represent a random field failure and effectiveness of safety mechanism in identifying this field failure is verified. These faults are classified into safe faults, meaning this fault will not lead to a critical event or a safety hazard and dangerous fault, meaning this fault will lead to a safety hazard. Furthermore dangerous faults are classified into detected faults and undetected faults. Detected faults are the faults that are detected by safety mechanisms and reported to ECU through a health flag. For an ASIL 'D' device 99% (Diagnostic Coverage) of dangerous faults needs to be detected.

Furthermore fault detection time (FDT) is an important spec to verify. For example, in case of an accident head of a driver may hit steering wheel in approximately 300ms (time is only for representation and will vary with automobile model). Air bag deployment may take 100ms (time only for representation not actual number). Any dangerous fault in the system needs to be detected well within 100ms and backup safety mechanisms if any has to kick-in. Hence fault detection time is a crucial specification that needs to be verified.

Safety verification flow is shown in figure 2. Once the functional verification of safety mechanisms is complete fault injection simulations can start. Based on results from fault injection simulations, faults are classified into safe-undetected, safe-detected, dangerous-detected and dangerous-undetected [4], [5]. Diagnostic coverage number will be calculated based on fault classification. If the diagnostic coverage meets ASIL requirement safety verification is complete. If it doesn't then new safety mechanisms may need to be added or existing safety mechanisms may need to be modified and whole cycle needs to be repeated. This is a time consuming step in functional safety verification as well as in overall product development. Furthermore, fault injection simulations are time taking , long running simulations as most of the faults needs to be inserted after system reaches steady state.

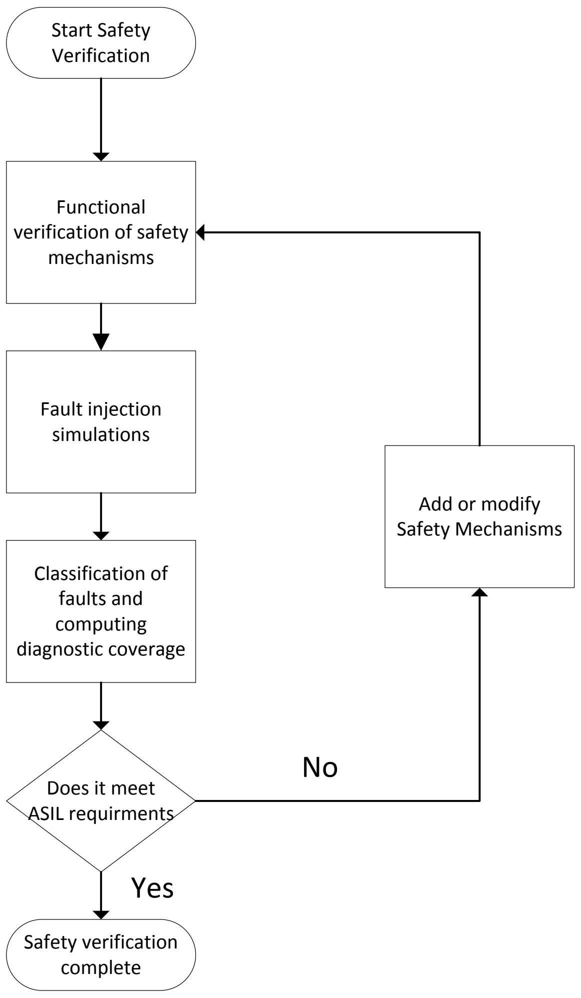

Start Safety Verification
Functional verification of safety mechanisms
Fault injection simulations
Classification of faults and computing diagnostic coverage
Does it meet ASIL requirments
No
Yes
Add or modify Safety Mechanisms
Safety verification complete

Figure 2. Functional safety verification flow

A typical sensor or a device contains thousands of nodes at which faults can occur. Simulating all the faults is cumbersome and time taking. Inserting five faults and subsequent testing in simulation based safety verification method requires approximately four hours. The reliability of a car is dependent on the safety features that are implemented and tested. Functional safety verification typically takes few months to complete.

This paper has proposed to use formal verification to verify the safety features in a quicker and efficient manner. This method will help reduce the complete safety verification effort by approximately 25% and the execution time to few weeks.

1.3 Formal Approach for Safety Verification

Formal verification offers exhaustive breadth-first state space exploration. Simple assertions can be written to verify the functional safety features. These assertions and the legal pin constraints are dependent on the project whereas the rest of the fault injection is automated. The formal tool from Cadence IEV has been used for the purpose of this paper. The IEV formal environment is easy to setup and IEV is able to prove assertions in minutes using underlying formal algorithms in comparison to simulation based method which takes hours. Same results can be reproduced with different vendor tools like VC formal, Questa formal, and Jasper.

As discussed earlier most of the faults needs to be inserted once design attains a stable state. Hence, a hybrid approach was used. Where in a functional simulation was run on the Design under test till design attains a stable state and at the end of the simulation state information is dumped. The formal run was preloaded with the stable state generated from the functional simulation run. To limit state space for formal tool and guide tool to valid scenarios constraints were written such as the active-low reset being held high throughout the simulation. The fault is injected as a constraint either on a pin at the top level or as interactive constraint to an internal module signal.

To automate the flow and reduce effort in fault injections, a Perl script is used to grep for all pins, signals or outputs in the RTL where faults can occur. The script then generates the constraints on the pins or signals for fault injection in a format that the formal tool can understand. Once the constraints are generated, they are provided as inputs to the tool and the result of whether the fault is detected or not is collected and presented to the user. Assertions are written to ensure health flag for the fault being injected is flagged after the expected interval of time determined by fault detection time. Thus the faults are classified into detected and undetected faults. Faults can be classified as dangerous or safe based on design judgement or all faults can be assumed as dangerous for worst case analysis. These numbers will be used to calculate diagnostic coverage. The above procedure is explained in more detail with an example in implementation section.

2. DESIGN DESCRIPTION

The Design under test (DUT) used for this work was a MEMS gyroscope for roll over and roll stability applications. It is similar to ADXRS810 mentioned in reference [7]. The DUT uses an internal continuous self-test architecture to check electro- mechanical system, PLL flag to check if internal PLL achieved lock, Checksum to check integrity of non-volatile memory, POR flag to check power-on-reset failure and many more. Details of failure will be available in FAULT registers in memory map. To reduce fault detection time in overall architecture summary of status is sent in every SPI communication through status vector bits (ST) indicating either device ok/device not ok/safety critical data/non safety critical data. In the formal verification environment faults are injected and assertions are written to ensure the fault is detected in the health flag register. In this design, close to 2000 fault nodes has been identified and verified. For the purpose of illustrating on how formal fault injection verification works power-on-reset failure has been used in the next section.

3. IMPLEMENTATION

Let us look at an implementation for a power-on-reset (POR) fault. If POR fails design will start in an unpredictable state. This is a dangerous fault and if undetected may cause hazardous event. Safety mechanism to detect such event is shown in representative RTL below. POR check is accomplished by getting a known value from non-volatile memory to volatile memory on power-on-reset. This value is compared with known value in non-volatile memory every clock-cycle. If the value doesn't match health flag is triggered. This safety mechanism can catch absence of power-on-reset and glitch in power supply which caused improper power-on-reset. Fault detection time for this failure is 10 clock cycles, i.e. within 10 cycles of the fault occurrence, the fail must be detected. The health flag is a sticky flag i.e. health flag will continue to show fault even if fault was transitory in nature until fault is read through SPI. The SPI read clears faults which are allowed to recur. This flag along with few other sticky flags are logically ORed to form status vector, the value of which determines if device is ok or in error state. The system can take further action based on the value of status vector. The implementation of por fail in digital logic is as follows:

```
wire [23:0] porid=24'hFACADE;
reg [23:0] por_reg;
always @(posedge clk or negedge porb)begin
 if(!porb)begin
  por_reg <=  24'hFACADE;
   end
else
  begin
  por_reg <=  por_reg;
  end
end
always @(*)begin
    por_fail    =~(por_reg==porid);
end
```

24'hFACADE is the predetermined value expected in the flops of interest. Since the fault detection time is determined from spec, constraints can be written accordingly. In this case, por_fail flag has been constrained to one (or) por_reg to an unexpected value as below in the tcl script:

```
constraint _add _pin {top.dut.digital_core.por_fail==1}
                      (OR)
constraint_add_pin {top.dut.digital_core.por_reg[23]==0}
```

System clock is setup, the device is constrained to be out of reset as this functionality is tested only when device is out of reset. The spi communication was turned off by constraining the chip select. DFT related functionality in the design has been disabled by constraining scan_mode to 0.

```
clock _add sys_clk -initial 0
constraint _add _pin porb 1
constraint _add _pin cs_n 1
constraint _add _pin scan_mode 0
```

Once the necessary constraints are applied, they must be reviewed to ensure design has not been over constrained. This is followed by writing the actual assertion required to test the fault.

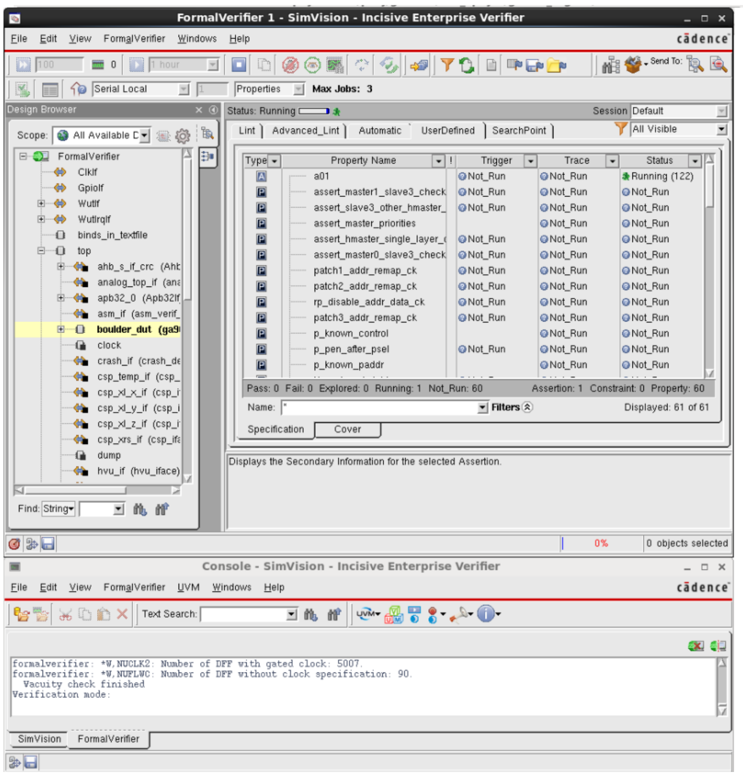
```
property status_flag;
@(posedge sys_clk)   ##[0:10]  (  Health0[3]  );
endproperty
a01  : assert property (status_flag);
```

In the above system verilog assertion Health0[3] is expected to flag within 10 cycles of injecting the fault. The formal tool tries to find scenarios to disprove the above assertion i.e. it tries to find cases where the expression @(posedge sys_clk)

##[0:10] (Health0[3]) results in zero. After a formal run completes the assertion can be in either pass, fail or explored state. If the formal tool proves the assertion it means that for the given set of constraints there are no scenarios where the status vector does not flag within 10 cycles of inserting the fault. As formal verification is not stimulus dependent and proves assertion pass or fail through binary arithmetic, results from this method is more fool proof the simulation methods. One must ensure the pass is not vacuous pass, i.e. the initial conditions for the assertions are not met hence resulting in a false pass. A failed assertion indicates there is at least a scenario where the property is not satisfied. Explored assertions are those where all scenarios are not covered yet, but for the ones tested there is no fail. Similarly, assertions are written for each fault.

3.1 Formal Tool

Formal tool used for the project is Incisive Enterprise Verifier (IEV) from Cadence, shown in Figure 3. IEV provides integration of formal analysis and simulation engines hence



Type	Property Name	!	Trigger	Trace	Status
A	a01		Not_Run	Not_Run	Running (122)
P	assert_master1_slave3_check		Not_Run	Not_Run	Not_Run
P	assert_slave3_other_hmaster_		Not_Run	Not_Run	Not_Run
P	assert_master_priorities			Not_Run	Not_Run
P	assert_hmaster_single_layer_(		Not_Run	Not_Run	Not_Run
P	assert_master0_slave3_check		Not_Run	Not_Run	Not_Run
P	patch1_addr_remap_ck		Not_Run	Not_Run	Not_Run
P	patch2_addr_remap_ck		Not_Run	Not_Run	Not_Run
P	rp_disable_addr_data_ck		Not_Run	Not_Run	Not_Run
P	patch3_addr_remap_ck		Not_Run	Not_Run	Not_Run
P	p_known_control			Not_Run	Not_Run
P	p_pen_after_psel		Not_Run	Not_Run	Not_Run
P	p_known_paddr			Not_Run	Not_Run

Pass: 0 Fail: 0 Explored: 0 Running: 1 Not_Run: 60 Assertion: 1 Constraint: 0 Property: 60

Name: * Filters Displayed: 61 of 61

Displays the Secondary Information for the selected Assertion.

```
formalverifier: *W,NUCLK2: Number of DFF with gated clock: 5007.
formalverifier: *W,NUFLWC: Number of DFF without clock specification: 90.
  Vacuity check finished
Verification mode:
```

Figure 3. Incisive Enterprise Verifier (IEV)

making it an ideal option for functional safety testing.

The DUT needs to be in known good state before the start of the formal run to avoid spurious fails hence the simulation engine of IEV can be used to simulate the DUT to known good state followed by the formal verification run for the assertions of interest. A simple tcl script containing constraints and commands can be used to provide inputs to IEV. The underlying formal algorithms scale to large and complex designs.

3.2 Challenges

1. Formal tool cannot proceed when the assertions involve a zero delay loop. In such cases cutpoints are created, basically cutting the logic leading upto the signal (for which the cutpoint is created) and creating a pin which toggles.

2. Formal tools in general are not scalable due to state space explosion. Some of the assertions for health flags which trigger at say 1000 cycles after injecting a fault require the tool to go deep into the design. Such assertions might not result in either a pass or a fail even after running for an hour. Black boxing the memory modules and using a powerful formal engine can help reduce convergence issues to a certain extent.

3. Formal tools do not support checks for safety mechanisms that are implemented in software. Formal will not replace functional simulation based approach. It will aid us to validate as many faults as possible very quickly and early in the design phase. For the remaining faults functional verification methods has to be used.

4. RESULTS

Certain assertions were proven within few minutes, whereas few assertions which required the formal tool to go deeper into the hierarchy were inconclusive. Such faults can be proven using tools like Certitude [6]. The result of the formal run is as follows:

Table 1

Fault Node	Detected	Detected Cycles	Undetected Proved	Undetected Explored	Explored
crash_csp	✓	**10**	**x**	**x**	**x**
Ntouch	✓	**100**	**x**	**x**	**x**
Por_fail	✓	**10**	**x**	**x**	**x**
smu_data_rdy	**x**	**x**	**x**	**x**	✓

Faults are detected when their corresponding assertions pass for the given number of cycles. In the above mentioned example if the health assertion passes, the fault is detected and the detected cycles is 10, else if it fails the fault is undetected proven. Cases which are challenging for the formal tool, for example when the spec mandates that the fault be detected in 5000 cycles, the user can chose to define the effort to say few minutes, such faults will be undetected explored. Explored faults are those for which the assertions result in neither a pass nor fail.

In cases of undetected explored and explored one needs to run Certitude simulations. Approximately 60% of faults can be verified by formal verification when the safety mechanisms are implemented in RTL. Remaining 40% needs to be validated with other methods like using Certitude.

Figure 4 shows witness waveform that can be generated from formal tool which aids in further debug if required.

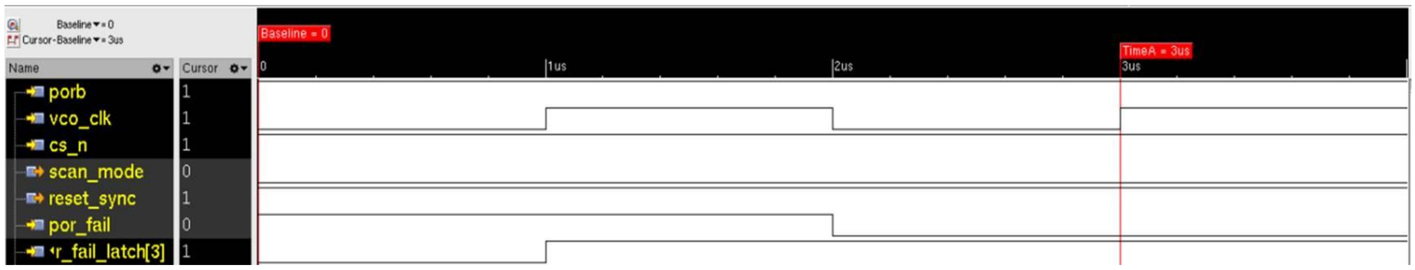


Figure 4. Witness waveform for debug

5. CONCLUSION

1. Formal verification for FuSa aims to prove the assertions for faults that are not too deep in design quickly and early in the design phase, while the harder faults can be simulated using simulation based safety verification techniques hence saving time and effort.
2. Formal tool is better at pointing out safety holes in the design compared to other methods like Certitude. Both simulation methods that are currently used have are dependency on stimulus and state of the system. Formal by nature will go beyond these to find safety holes.
3. In an industry where time to market is crucial and verification takes up almost 60% of the design cycle, formal verification can help build confidence in the safety verification process for automotive chips in a fast and efficient manner.

Future scope of this work is to create a formal app that can read design and generate constraints automatically. This will further reduce the amount of effort and time required in functional safety verification.

6. ACKNOWLEDGMENTS

Authors would like to thank Raigir, Kamalaker for his guidance, support and encouragement.

7. REFERENCES

[1] Ismail, Azianti, Qiang, Liu, 2014, ISO 26262 automotive functional safety: issues and challenges, International Journal of Reliability and Applications.

[2] Born, Marc, Favaro, John, Kath,Olaf , 2010, Application of ISO DIS 26262 in practice. In workshop on Critical Automotive Applications: Robustness & Safety.

[3] Alexandersson, Sabine, 2008, Functional safety and EMC for the automotive industry.

[4] Janos, Olah, Majzik, Istvan, 2009, A Model Based Framework for Specifying and Executing Fault Injection Experiments.

[5] Hsueh, Mei-Chen, Tsai, Timothy, Iyer, Ravishankar.k, 2014, Fault injection techniques and tools.

[6] Devaphanindra Kumar, Ranganayakulu Sri SNUG 2012, Bangalore, Certitude for functional safety

[7] ADXRS810 High Performance, SPI Digital Output, Angular rate sensor datasheet