Measuring the Effectiveness of ISO26262 Compliant Self Test Library

Frederico Pratas, Thomas Dedes, Andrew Webber, Majid Bemanian, Itai Yarom
MIPS Tech, LLC, 3201 Scott Blvd., SC, CA 95054, USA
E-mail: {frederico.pratas,thomas.dedes,andrew.webber,majid.bemanian,itai.yarom}@mips.com

Abstract— Automotive SoCs are constantly being tested for correct functional operation, even long after they have left fabrication. The testing is done at the start of operation (car ignition) and repeatedly during operation (during the drive) to check for faults. Faults can result from, but are not restricted to, a failure in a part of a semiconductor circuit such as a failed transistor, interconnect failure due to electromigration, or faults caused by soft errors (e.g., an alpha particle switching a bit in a RAM or other circuit element). While the tests can run long after the chip was taped-out, the safety definition and test plan effort is starting as early as the specification definitions. In this paper we give an introduction to functional safety concentrating on the ISO26262 standard and we touch on a couple of approaches to functional safety for an Intellectual Property (IP) part such as a microprocessor, including software self-test libraries and logic BIST. We discuss the additional effort needed for developing a design for the automotive market. Lastly, we focus on our experience of using fault grading as a method for developing a self-test library that periodically tests the circuit operation. We discuss the effect that implementation decisions have on this effort and why it is important to start with this effort early in the design process.

Keywords— ISO26262, automotive, functional safety, fault injection, Self-Test Library

I. Introduction

The automotive industry has been adopting a large number of electronic components into the design of all types of road vehicles. These electronic components include everything from Engine Control Units (ECUs), parking sensors, automatic braking systems and sensors for advanced driver assistance systems (ADAS) through to the radio/media player and the car dashboard. Standards were already established related to safety for aeronautics (DO254) but nothing was addressing the specific problem of consumer vehicles. Indeed, a standard as stringent as DO254 [1] can not be directly applied to cars and motorbikes because implementing the safety levels of such standards has a very serious impact in the product engineering cost, execution time (time-to-market) without a major benefit. However, as electronic components play a more important role in safety and mission-critical applications in today's automotive products, thus leading to a total system dependability, the need for a standard that properly regulates and unifies the way how industry is developing vehicles (or parts going into vehicles) can not be ignored. This is even more important since there are a significant number of vehicles on earth (more than a billion), and vehicles represent a public safety issue both from the inherent risk of the vehicle itself (carrying passengers at a speed) and their interaction with the surroundings. Thus, there was a need for functional safety to be defined as a concept, and that has been done through different standards over the years.

Functional safety has been considered in a number of standards - initially IEC 61508 [2] was used for automotive applications prior to the adoption of ISO26262 as a standard in 2011 [3]. Other related standards address other markets such as aeronautics, railways and industrial processes, however, this manuscript focus on automotive and the ISO26262 standard. The ISO26262 standard for "Road vehicles - Functional Safety" currently addresses the design and manufacture of electronic devices for cars in particular, however, a revision of the ISO26262 standard is underway that adds part 11 to address applications to semiconductors (including Intellectual Property or IP parts) and part 12 that addresses motorcycles. The new revision of the standard will also add material to address the requirements for trucks and buses. The current projection is for this update to the ISO26262 standard to be in draft in 2017 and to be adopted as edition 2 of the standard in 2018.

As such there is an new requirement for IP designers targeting the automotive market to comply with these standards and to understand how they can help higher tier customers address the requirements of Functional Safety through a distributed development model. In this model IP suppliers have to work closely with the semiconductor component vendor to understand the requirements of the OEM/Tier 1 end customer.

The Functional Safety requirements can be split into:

- First are the technical requirements, related to fault tolerance throughout the design, depending on which safety level its targeted. Indeed, ISO26262 provides different safety categories depending on different criteria like severity, exposure and controllability.
- Secondly there are the flow requirements. Functional Safety is not a standalone development flow, it has to be integrated into the regular development flow of the product, whether it is hardware or software. This involves introducing new development stages (such as fault injection, which is further discussed in this paper), to also adding traceability. These requirements are due to the fact that Functional Safety state-of-the-art is constantly changing and improving, thus one can trace that best practice was followed during the project development.

This paper presents the basics of Functional Safety and ISO26262, explaining what ASIL refers to, what are the different types of faults, and what are the metrics that ISO26262 uses in order to assess the safety level of the IP. Then, it develops on how the normal system-on-chip design flow is impacted by the addition of Functional Safety measures. Finally, it details some important aspects relative to the process of fault injection performed during the development and validation of the STL, namely, how chal-

lenges related with the design size should be overcome. In particular, it shows how different fault injection detection techniques can affect the simulation time by two orders of magnitude at the cost of introducing different levels of error in the results and how that can be used to accelerate the development phase for designs with different complexity.

II. Background

Reliable design and operation of a microprocessor involves extreme challenges. In particular, there are many sources of errors (faults) during different stages of the product lifetime, from design until operation. Faults as described by the ISO26262 standard largely fall into two categories: a) systematic faults - typically hardware or software bugs that were not found during the development of the IP, and b) random faults - typically faults that occur when the device is in operation which are inherently due to the technology used or transient events (such as alpha particles striking the device when in operation). Moreover, the ISO26262 gives several metrics to categorize each functionality of a system into an Automotive Safety Integrity Level (ASIL) by assessing the Controllability, the Severity and the Exposure of each potential failure. For example the authors in [5] assess how the ISO26262 would be applied to classify a microprocessor. The lowest level of all is Quality Managed (QM) that does not specify any concrete metrics for random fault tolerance but which may still be applicable to certain electronic components within an automobile. In addition to the lowest ASIL QM there are four other ASIL levels, ASIL A to ASIL D (high fault tolerance). The requirements for a particular ASIL level results from the role the item and its related elements play in the system. As mentioned before, the ASIL level is driven by the hazard and environment conditions relating to severity, exposure and controllability. For example, a braking system may need to have the highest level of fault tolerance (ASIL D), as it cannot brake or lock-up in an uncontrolled manner. Once the initial ASIL level for the item has been defined, the ASIL level for each of its components, parts and sub-parts, can be defined by means of ASIL decomposition. ASIL decomposition is a common practice to decompose the initial safety requirements into redundant safety requirements. Redundancy in this context means having multiple different solutions to avoid violation of the same safety goal, which are based on independent functional elements that are incorporated into the same component or part. In particular, if redundant safety requirements are mapped into independent elements, the probability of repeating the same error is reduced. In such a case, ISO26262 allows to lower the target of some of the metrics for each independent element, in the form of ASIL X[Y], where Y is the initial ASIL and X is the decomposed ASIL level for the independent element. The decomposed ASIL X[Y] differs from the standard ASIL X in that it requires the confirmation measures to be conducted at the initial ASIL level. For instance, an ASIL B[D] element requires the confirmation measures to be conducted by someone outside of the organization (i.e., an independent auditor) according to the ASIL D specifications.

Random faults within the ISO26262 standard fall into

TABLE I: ASIL levels [3]

Safety level	SPFM	LFM
QM	-	-
ASIL A	-	-
ASIL B	>= 90%	>= 60%
ASIL C	>= 97%	>= 80%
ASIL D	>= 99%	>= 90%

the categories: single-point fault metric ('SPFM'), latent multi-point fault metric ('LFM') and dependent faults. SPFM relates to faults that have an immediate effect in the device operation, whereas LFM refers to a fault that remains unnoticed (latent) in the system until another fault triggers its propagation into the system. This paper focuses on fault tolerance to SPFM and LFM, as quantitatively categorised to the different ASIL levels (see Table I).

Random faults are actually addressed in two ways the first of which is the incorporation of 'safety mechanisms' within the design of a part during development. An example of such a 'safety mechanism' would be the use of Error Correcting Codes (ECC) or parity around cache or other RAM structures included within the semiconductor design, as well as adding extra safety to the system, either by using highly redundant techniques such as processor lock-step, as proposed in [8], or by embodying optimized checking hardware structures such as the proposal in [9].

The second means to address random faults is to carry out regular testing on the system. Tests can be carried out both at start-up and during operation. In the former case, a key area that can be regularly tested is the memory consistency, e.g., using Memory BIST (MBIST). In the latter case, the system can take sub-parts of the system out of operation for testing, where testing is specifically checking for faults. There are a number of ways to perform testing during operation - for a general electronic circuit one common means would be to employ Logic BIST (LBIST) to perform fault coverage analysis through randomised patterns sent through all the flip-flops in the design via scan chains added and used for Design For Test (DFT).

The two main metrics regarding Hardware Functional Safety are "Hardware Architecture Metrics" and "Probabilistic Hardware Metrics". The former measures the ability for a design to cope with random hardware failures, while the latter measures the likelihood of random hardware failures to violate the safety goals. Although hardware safety mechanisms are the strongest and most efficient way to deal with safety, their cost in terms of area, power consumption and timing opens the way to software methods, less efficient, but also less costly, and with the advantage of being able to be developed after production of the hardware.

For a device that incorporates a microprocessor, software testing can regularly check the functioning of the device via a method called Self-Test Library (STL). STL has been applied in the past [6].Such techniques consist of letting the CPU run a sequence of code comprised of a library of tests executed at predefined time intervals by the Device

Under Test (DUT). These tests are specially designed to exercise and detect faults in the circuit [7], typically by comparing results obtained with a golden dataset or computed signature. In comparison to hardware mechanisms, software techniques support autonomously testing,and diagnosis both the CPU and some peripherals without the need to develop and implement any additional hardware mechanisms. The main constraint to this method is that it requires taking the execution unit out of production, thus being unable to target areas of logic mandatory to the safe state of the running application. For example, although one can stall the execution unit for some time in order to run the STL, it is not recommended to corrupt the memories, as it may have a functional impact on the application once returned to the original execution flow.

The main goal in the STL development process is the maximization of the fault coverage in the design, mainly when targeting high ASILs. Achieving high fault coverage in modern microprocessors imposes several challengesin the STL. In fact, the very concept of the STL puts constraints on its capabilities, namely:

- Duration of its execution can be crucial to the system. As a matter of fact, no safety related system can be put on hold indefinitely while performing its diagnostic. Therefore, there is a constraint on how long the STL takes to run. Additional constraints related to specific functionalities can also apply, for example in the case of interrupts.
- The size of the code can also be of importance, as we will often be looking at embedded systems where memory space is a limited resource.

In order to address such challenges engineers usually rely on a development flow that iterates between introducing software optimizations and assessing fault coverage. Several software related optimization issues have been discussed in the past, for example in [10] the authors describe the software development for targeting a small microprocessor with 200k stuck-at faults. However, as processors increase in size and complexity, the iteration process tends to be limited by the significant simulation time required to perform the fault coverage validation step [10]. In this paper we analyse the impact of fault injection simulation in the STL project development cycle for a small ($\approx 400K$ faults) and a medium size ($\approx 2.5M$ faults) microprocessors. In contrast to previous works, we show several strategies that can be used to overcome the limitations and accelerate the development cycle.

III. Flow

In a standard System-on-Chip (SoC) design, one usually starts with design specification, followed by RTL implementation and IP integration, and SoC verification toward a tape-out (see Figure 1). When designing an SoC for automotive it has been mentioned that there is the need to integrate safety within the design flow. It happens that most steps have a safety formal process associated. Starting with the safety specification, followed by the design of the safety mechanisms which are necessary to meet the safety requirements, and terminating with the validation of the safety mechanisms as part of the STL preparation. Developing the safety part of a project alongside with the SoC design is mandatory practice to avoid the cost of adding safety mechanisms at the end of the design process.

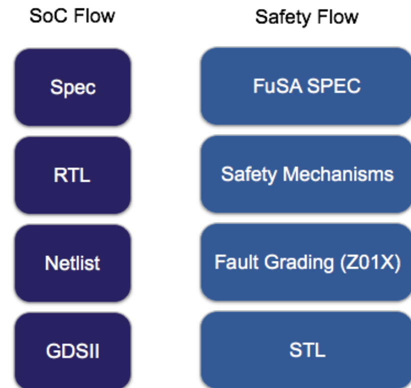


Fig. 1: Design and safety development steps

However, in order to claim a certain ASIL for an IP, an overall safety culture should be implemented within the company. This goes from having a stricter development flow: from specification breakdown, to code reviews gating a project's advancement, to traceability of most things (meeting minutes, bug tracking, change requests, etc...). This ensures that if something happens after tape-out, any company which had developped a part of a SoC can show that due diligence was done. This paper will assume that the SoC or IP have the safety mechanisms already and it will focus on the process of developing the Safe Test Library (STL) that is being used to check the correct behavior of the design during operation. The STL address faults that can occur during operation, while production faults are addressed by methods as scan & ATPG during production.

IV. Fault Injection and Grading

As described in Section **2**, in Functional Safety fault injection is a widely used technique for evaluating the system sensitivity to faults, it can be used with different goals in mind: to verify the performance of every Safety Mechanism, dependability validation [11], failure prediction [12], estimation of the fault tolerance level [13], and validation of the fault tolerant solutions.

In this case we use fault injection to validate the level of fault coverage of the STL. As described above the STL is one of the software (SW) methods to provide diagnostic coverage for safety-related elements. Its function is to be periodically executed by a processing unit in order to test that the processing unit itself or any part of the system is operating as it should. The STL can both complement existing Hardware (HW) Safety Mechanisms by adding extra coverage to reach higher ASILs, but it can also be suitable for systems having little to no HW Safety Mechanisms.

The results obtained from the fault injection allow the Functional Safety Engineer to do the safety analysis and determine whether the system achieves the targeted ASIL requirements or not. In our case study, we also assess the coverage of the STL itself, thus helping in driving its development. The key aspect when analysing software is in how we categorize that a fault has been detected. In order to

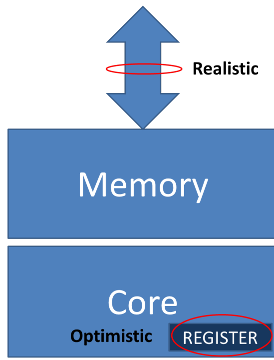


Fig. 2: Strobing points in simplified CPU model

achieve that, we set up mechanisms to signal to the fault injection tool that the software did indeed see the fault, so that it can be categorized as "detected".

Faults are usually classified according to their impact and "observability", i.e., assuming a specific "observation" point in the design (strobing point), either the boundaries of a defined area, or the memory bus. Namely, in the context of the STL development, we consider:

- **Observed** : The fault is observable if there is a difference to the reference run (aka golden run) that is propagated to the strobing point (in our case the pins of the processor) that has not necessarily been detected by the STL.
- **Not Observed** : A not observable fault is such that its effect is not observed in the strobing point, either because the test is not able to trigger it, or because it cannot be propagated to a detectable point (for example if it is always masked by another signal).
- **Detected** : The effect of the fault reaches the strobing point, and the STL is able to identify the execution difference and flag an error.

The main problem regarding the STL coverage is actually how to collect it, and whether it makes sense from a Functional Safety perspective. Indeed, the developer is in charge of picking the observation points, and those vastly impact on the resulting coverage figures. Different strobing points also impact the simulation run time, and we will cover that later on. The location of the strobing points defines how optimistic or pessimistic the coverage metrics will be in comparison to real numbers.

The STL detects faults by comparing an error signature (which is stored in a specific register) to a golden precalculated dataset. Let us consider the simplified CPU representation shown in Figure 2. We consider two possible strobing points in the design:

- **Realistic**: The strobing point is located at the boundaries of the system. Observing the CPU pins gives 100% certainty that any difference between the golden machine and the faulty machine has indeed been propagated to the outside. However, that doesn't mean that it has been detected by the STL. It only means that the fault is potentially dangerous, as it has left the boundaries of the system. To know whether it has been detected by the STL or not, we rely on the fact that the STL, when detecting a fault, is writing a specific value to a fixed memory position, which we can probe for on the memory bus. By looking specifically at the memory bus we know that, if a difference occurs, the fault has been detected by the STL. Note that we had to separate the strobing point on the memory bus from any other strobing points on pins of the CPU, as one (memory bus) will mean fault detection, while others (others pins) will mean that the fault is dangerous, but not necessarily detected. This observation point is the most realistic one.
- **Optimistic**: Another possibility is to probe directly the register in which the STL is storing the error signature. If we observe a difference there, then it means that the STL has the possibility to detect it. As we are observing a point in the system before the actually detection mechanism triggers, we can only ensure the STL has the data available to detect the fault, not that it actually detected it, hence why this strobing point can be quite optimistic, depending on the STL's behaviour and resilience. However, taking a decision between considering a fault detected or not occurs much earlier, which presents advantages, as shown below.

TABLE II: Target IP cores and respective number of faults

Core Name	#Faults	Relative size
uController	$\approx 400K$	Small
CPU	$\approx 2.5M$	Medium

V. Experimental Setup and Results

The case study presented herein is based on a real project developed within MIPS. This project targeted two different MIPS processor IP cores as defined in Table II: *i)* a smaller micro controller (henceforth named uController) composed of a single threaded core with L1 caches and embedded SRAM, and *ii)* a larger multi-threaded CPU (henceforth named CPU) composed of four multi-threaded cores with L1 caches and a shared L2 cache level. To perform fault injection in the target IP we have used Synopsys VCS L-2016.06 and Synopsys Z01X 3.2.3, both tools installed on a Linux environment. VCS was mainly used when cross-checking fault simulations with RTL simulations for debugging purposes, while the fault injection tasks where performed in Z01X. Z01X is a massively parallel fault injection simulator that, in this context, was used to help develop the STL. It provides detailed fault detection coverage reports used when analysing the effectiveness of the STL. In particular, we are interested in detailed information about which faults in the design are: a) not observable; b) not detected; and c) detected.

As explained in Section **4**, a fault is considered as not observable if it is not possible to drive its defect to the detection point, i.e., there is not a visible effect, either because the STL is not able to detect the fault or because it is not propagated through IO pins. The fault is considered as observable but not detected if its effects are propagated to an observable point in the design but STL is not able to detect it. And finally, a fault is classified as detected when the STL is able to return a positive detection result during its execution. Z01X further classifies a subgroup of not observable faults and untestable, namely those faults that

due to design/circuit constraints can not be exercised or controlled. Such faults are by definition safe faults as they do not have an impact in the functionality of the circuit[1].

Because fault simulation is a computationally intensive task, Z01X supports several features that can help optimize the execution. The most relevant ones are: *i)* minimize the list of faults that actually require a full simulation by circuit analysis, *ii)* execute multiple fault simulations in parallel, and *iii)*) use statistical methods to speedup some of the analysis (at the expense of some accuracy). In *i)*, for example, the tool discards untestable faults, as well as other not observable faults by performing an early stage classification based on toggle analysis of the golden run, i.e., a simulation without any faults being injected. For *ii)*, the tool can simulate multiple batches of faults at the same time. Finally, in *iii)* the designer can configure the tool to both perform fault sampling and only simulate a subset of the total fault universe, thus obtaining representative coverage figures in a quicker way; as well as skip any faults that require a significantly larger amount of computational resources and time, usually due to factors such as oscillations and instability in the design, we refer to this type of faults from this point forward as HyperFaults (HF). Examples of this class of faults are:

- **HyperActive**: Indicates the faulty machine's activity overwhelmed the simulation. Hyperactive faults are typically a very small number of faults that require more system resources than other faults.
- **HyperMemory**: Indicated high memory usage by the faulty machine. Hyperactive faults are typically a very small number of faults that require more system resources than other faults.

As explained above, the main target of this project was to develop an STL capable of providing diagnostic coverage for the target IP cores. Fault injection was performed on target modules in the IP, namely those that do not include safety-related mechanisms implemented in HW. Several iterations were required during the project life cycle in order to drive the STL development and achieve the target fault detection coverage figures. Figure 3 shows the evolution for the two different designs. As expected, achieving the target coverage in the largest (and more complex) design was a lot more challenging.

The shaded regions correspond to the amount of HFs in the design. Due to the significant run time required to simulate these faults, Z01X allows to drop them during execution to accelerate the overall execution time (as explained above), at the cost of introducing an error in the final result.

Since in our design the number of HFs was quite significant we had to look into strategies to reduce this number and accelerate the overall execution time. Figure 4 shows the relative run time (in log scale) for both IP designs while using different strobing strategies, as explained in Section **4**. One can see that the run time is approximately 60x (uController) / 48x (CPU) faster when dropping HFs, when compared with the full run. However, this means that about 15% (uController) / 21% (CPU) of the faults in the design are not simulated. By changing the strobing point deeper into the design, as explained in Section **4**, we are able to reduce the number of HFs significantly. In fact, by strobing the register that holds the STL error signature we are able to reduce the amount of HFs to only 0.17% (uController) / 6.49% (CPU) of the total number of faults in the design. This strategy further reduces the total simulation time by about 2.7x (uController) / 3.0x (CPU), resulting in an overall speedup of 160x (uController) / 145x (CPU) relative to the full run (see Figure 3), with the additional gain that we now are simulating all the faults.

While the different approaches mentioned above improve the execution time quite significantly, they also introduce

[1]We have performed additional simulation tests to confirm that all faults classified as untestable by the tool were actually safe.

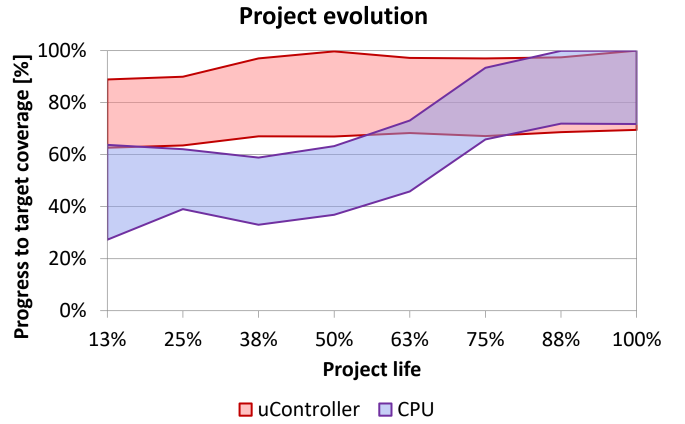


Fig. 3: Fault coverage evolution during project lifecycle

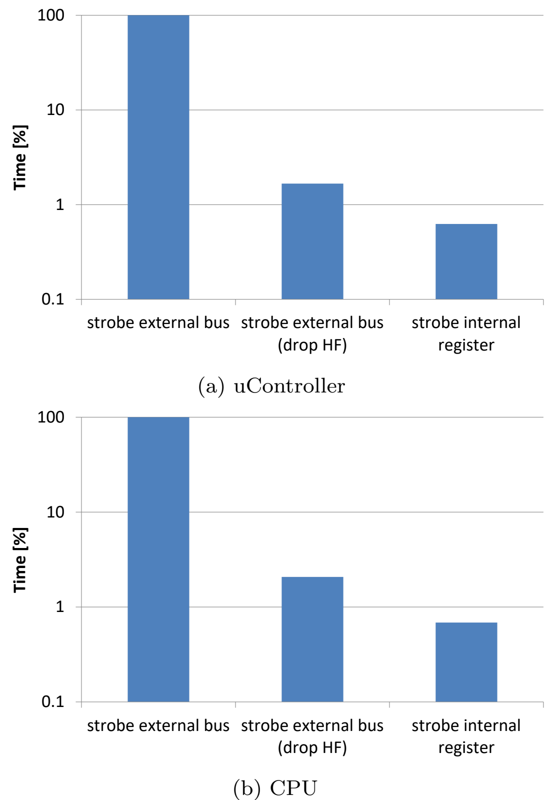
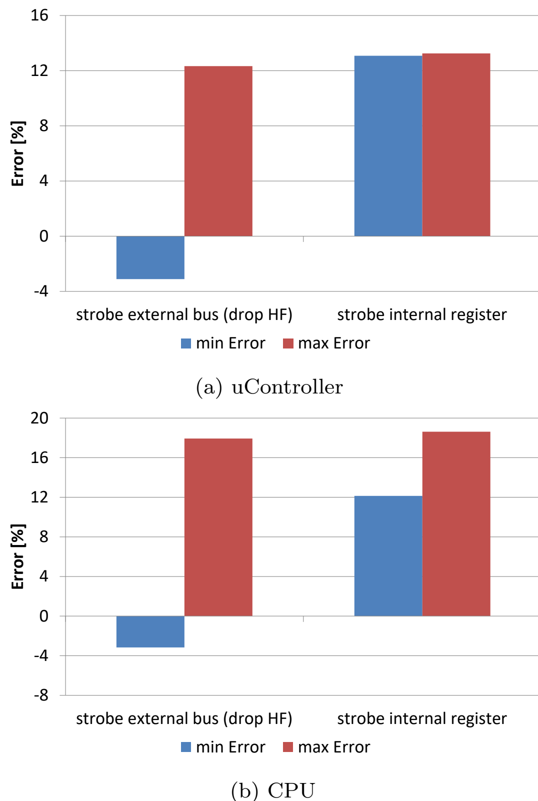


Fig. 4: Simulation time for different strobing strategies



(a) uController

(b) CPU

Fig. 5: Coverage error for different strobing strategies

an error in the results that needs to be taken into account when analysing and using the data for decision making. Figure 5 shows how the different approaches relate in terms of error relative to the full run. Namely, one can see in Figure 5 that dropping all HFs results in an uncertainty of 15.44% (uController) / 21.1% (CPU) relative to the full run. This results from the fact that, though all the simulated faults have maximum certainty, we don't know the exact result of the skipped HFs (which may be detected or not). In particular, on the one hand, if we consider all HFs as not-detected then we get a conservative result which is 3.11% (uController) / 3.17% (CPU) below the exact result. On the other hand, classifying all HFs as detected faults leads to an overoptimistic result with error of 12.33% (uController) / 17.93% (CPU).

When the strobing point is changed to stop the simulation as soon as there is a deviation in the error signature register, we also obtain an optimistic result. In this case the difference is obtained because not all deviations are actually detected by the STL (in some cases the failure can be masked as explained in Section **1**). For the examples shown in Figure 5 the error observed ranges between 13.08% and 13.25% (uController) / 12.13% and 18.62% (CPU). For comparison purposes the coefficient of variation of the root-mean-square deviation for the two strategies is of 1.95 and 1.00 (uController) / 1.744 and 1.00 (CPU), respectively. These results show that both strategies result in a similar maximum error. Nevertheless, while dropping the HFs results in a larger uncertainty, strobing the internal register results in very optimistic results. These differences are important mainly during the STL development phase, because larger uncertainty complicates the assessment of the improvements introduced in different versions of the library, while final values used for safety analysis should be reported using the most conservative approach, i.e, following the pessimistic results.

VI. Conclusions

Recently there has been a significant drive in the automotive market towards the implementation of more and more complex automated functions (ADAS). This is increasing the safety risks and thus the safety awareness and responsibilities deeper into the supplier tiers. In fact, a revision of the ISO26262 standard is currently underway that adds part 11 to address applications to semiconductors (including Intellectual Property or IP parts). In this paper we focus on a method for driving the development of a self-test library that will periodically be used to check the circuit operation of IP already in silicon. Namely, we use fault injection simulation data, obtained with Synopsys Z01X, as a metric to gauge the effectiveness of the STL software. Experimental results show that different fault injection detection techniques can affect the simulation time by two orders of magnitude at the cost of introducing different levels of error into the results for designs with different complexity. We show how these have an impact on the project development life cycle and that different techniques should be used at different development stages.

References

[1] Radio Technical Commission for Aeronautics: DO-254, Design Assurance Guidance for Airborne Electronic Hardware International Standard, 2000

[2] International Electrotechnical Commission: IEC 61508: Functional Safety Standard International Standard, 2010

[3] International Organization for Standardization: ISO26262: Road vehicles - Functional safety International Standard, 2011

[4] Synopsys: Z01X Functional Safety Assurance: High-Speed Fault Simulation Solution for IEC 61508 and ISO26262 Compliance Technical Reference Manual

[5] Y. C. Chang, L. R. Huang, H. C. Liu, et. al: Assessing automotive functional safety microprocessor with ISO26262 hardware requirements Technical Papers of 2014 International Symposium on VLSI Design, Automation and Test, Hsinchu, 2014, pp. 1:4

[6] C. H. P. Wen, Li. C. Wang and K.-T. Cheng: Simulation-based functional test generation for embedded processors IEEE Trans. Comput., vol. 55, no. 11, Nov. 2006, pp. 1335:1343

[7] M. A. Skitsas, C. A. Nicopoulos and M. K. Michael: DaemonGuard: OS-assisted selective software-based self-testing for multi-core systems Proc. IEEE Int. Symp. Defect Fault Tolerance VLSI Nanotechnol. Syst., Oct. 2013, pp. 45:51

[8] H. Kimura, H. Noda, H. Watanabe, et. al: 3.5 A 40nm flash microcontroller with 0.80s field-oriented-control intelligent motor timer and functional safety system for next-generation EV/HEV Proc. IEEE ISSCC, San Francisco, CA, 2017, pp. 58:59

[9] I. Wali, A. Virazel, A. Bosio, et. al: A Hybrid Fault-Tolerant Architecture for Highly Reliable Processing Cores J. Electron. Test., vol. 32, no. 2, 2016, pp. 147:161

[10] P. Bernardi, R. Cantoro, S. De Luca, et. al: Development Flow for On-Line Core Self-Test of Automotive Microcontrollers IEEE Trans. Comput., vol. 65, no. 3, March 2016, pp. 744:754

[11] J. Arlat, M. Aguera, L. Amat, et al.: Fault injection for dependability validation: a methodology and some applications IEEE Trans. Softw. Eng., vol. 16, no. 2, Feb 1990, pp. 166:182

[12] M. Vieira, H. Madeira, I. Irrera, et. al: Fault injection for failure prediction methods validation IEEE/IFIP Intl. Conf. on Dependable Systems and Networks, 2009

[13] M. Cukier, D. Powell and J. Ariat: Coverage estimation methods for stratified fault-injection IEEE Trans. Comput., vol. 48, no. 7, Jul 1999, pp. 707:723