Formal Fault Injection in Digital Blocks with Mined Assertions

Damiano Zuccalà*†, Paul Breuil*, Jean-Marc Daveau*, Philippe Roche*, Katell Morin-Allory†
*STMicroelectronics, 850 Rue Jean Monnet 38926 Crolles Cedex, France,
firstname.name@st.com
† *Univ. Grenoble Alpes, CNRS, Grenoble INP**, TIMA*, 38000 Grenoble, France
firstname.name@univ-grenoble-alpes.fr
** Institute of Engineering Univ. Grenoble Alpes

***Abstract*—As digital systems keep miniaturizing and becoming more complex, new methods are required to ensure their fault tolerance. Research and industry are working together to automatize such task, and, recently, great progress has been made to overcome this challenge. Starting from an exhaustive reference simulation, we have a general procedure to determine, by Model Checking, the fault tolerance of flip-flops that are perturbed by bit-flip events. Assuming that the design is correctly implemented, the temporal properties are used as fault detectors, automatically generated through a re-adapted version of the open source software Goldmine. By this, we can also address circuits without explicit specification (blind blocks), using automatic assertions induced from the reference golden waveform. Properties are mainly mined inside the design, allowing to calculate by Model Checking the fault masking capacity of the addressed modules.**

We consider two medium sized blocks, showing results with much-improved performance and accuracy over classical simulated fault injection.

***Index Terms*—Formal Fault Injection, Goldmine, Functional Verification, Fault Masking.**

I. Introduction

The design automation of hardware systems is increasing its importance over time, as their in-silicon down scaling constantly augments the circuits complexity. The need to ensure the correct information flow in many critical sectors (automotive, banking, avionics, radiation detectors) also imposes strict conditions on the specification, whose flaws may lead to severe jeopardy and compromising consequences.

After decades of commitment, the development of algorithms for automatic specification [1] leads to new paradigms. Combining such tools with the study of fault tolerance, we implemented a complete automatic flow for fault injection and detection by Model Checking [2], [3], to characterize the hardware fault tolerance. In particular, we focus on Single Event Upset (SEU [4]) faults - single bit flips that may occur in any non-protected flip-flop. The module's fault tolerance is characterized by injecting SEUs by Model Checking, instead of classical simulation. This optimizes the computational effort and improves the fault detection accuracy, being test-bench independent and exhaustively traversing all functional states.

Automatic functionality checkers are normally used to detect design bugs, with temporal properties representing the specification [3]. In this work, the design is priorly assumed to be correct, and assertions (proven in absence of external errors) have the role of fault detectors. The automatic generation of assertions for designs that are not ensured to be bug-free, can be risky and questionable approach. Nevertheless, this eventuality does not concern our scenarios, as the functionality of the addressed blocks is well known and exhaustively tested.

Human-built properties usually concern only the output pins of a digital block, as its complexity makes hand-construction hard for non-documented circuit structures. In presence of partial specifications, Intellectual Properties, intricate multi-modules systems, the manual construction of assertions through reverse engineering turns out to be a very tough task, even for teams of expert engineers. Thus, the choice of automatic assertion generation may be justified, to address the problem for "black boxed" circuits (free of design bugs).

We adapted the open software Goldmine [1], that automatically generates properties, leveraging on the waveform of a reference simulation. In this work, such mining process focuses on creating assertions mainly inside and at the output pins of two medium-sized test cases. This choice permits to estimate the fault masking capacity of the circuit: the number of fragile flip-flops (injecting faults by Model Checking) detected by internally mined properties, over the ones causing protocol failure with output boundary assertions.

Mining inside the module provides a much deeper insight of how the system behaves, catching the faults on their early stage of propagation. Such innovative analysis allows to directly prevent the faults propagation with a much lower effort and at a cheaper price. This characterization is essential in the automotive industry, as the early detection and absorption of failures plays a central role in modern vehicles safety [5].

We describe the automation of the assertion generation along with the formal fault injection flow, mining inside and at the module outputs, then comparing the enhanced performances over simulated statistical fault injection.

2832-6520/24/$31.00 ©2024 IEEE
DOI 10.1109/MEMOCODE63347.2024.00014

Finally, the formal fault injection results with distinct properties (internal and boundary) are used to compute the fault masking capacity of the test cases, a result which would be practically very hard to reproduce in simulation, as this is test-bench dependent, and with randomly sampled faults over time.

II. State of the Art

A. Functional Verification

A necessary step across the whole design flow is the matching between the hardware specification and its effective execution. Functional verification addresses this task, and nowadays is a major topic for a wide range of industry and research applications [6], augmenting its importance as long as integrated circuits continue to miniaturize.

1) Simulation: Simulating the design is the most deployed methodology to face the verification problem, by generating several input stimuli patterns, then observing the behaviour at the primary outputs [7].

Assertion-based verification [8] may be a better solution to detect design bugs. Such approach aims to model the functioning protocol of the concerned system by a set of well defined properties, then considering as errors only the flaws causing their breakdown. In all cases, the outcome of the simulated approach depends on the inputs stimuli quality and their coverage capability, when traversing the state space.

2) Model Checking: From automata theory [9], hardware and software environments may be represented as transition systems, while the set of temporal properties are time dependent sequences expressed in a language accepted by the first [10]. In particular the Model Checking algorithm, given a design and a property (assertion) that doesn't hold on it, is designed to report a counterexample trace of such failure.

In this work, such search algorithm is used to quantitatively define the response of complex hardware modules when faults, *e.g.* errors generated in the state words, propagate through the digital network.

B. Protocol Characterization

The verification task aims to verify the system's protocol, which is represented by temporal properties. Their quality strictly depends on the human capacity to encode the specification, that can be unclear, nor existing (making the problem even more complex), often occupying whole verification teams of experts for long periods. This is a common situation in industry environments, where Intellectual Properties (IPs) are treated as black boxes among institutions, or without a standard description for a given block. To address these challenges, Goldmine [1] is designed to automatically generate properties for the digital device, relying only on the target module structure and the exercised golden stimuli.

In general, there are no standard techniques to use, also because the selection of the most suitable method varies as a function of the particular problem to solve.

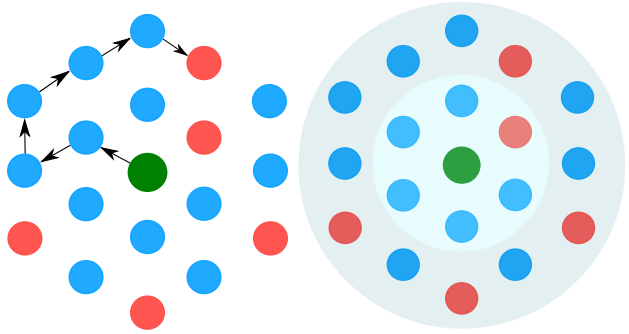
C. Fault Tolerance

Fault tolerance is the system's capability to endure and recover from external anomalies, that may cause diversions from the foreseen execution of the program.

1) Simulated Fault Injection: The most common technique to address the tolerance task relies on Monte Carlo sampling [11], even if this method shows several limitations as the design size begins to enlarge. Here, firstly a reference (golden) dynamic is computed and stored, then the strobes (error detectors) are defined, finally several distinct additional "faulty" simulations are performed [7].

Several distinct types of errors can be injected in the hardware network, such as locking a flip-flop to a specific state (Stuck at One - SA1, Stuck at Zero - SA0) or flipping its value until a new event overwrites the register state (SEU).

Such statistical approach turns out to be extremely expensive in terms of resources, even to extract approximated information. Moreover, via simulation the system only traverses an exiguous fraction (depending on the input stimuli) of all the accessible paths in the state space, with the possibility to rewind on already explored configurations.



(a) Simulation (b) Model Checking

Fig. 1: Visual state space exploration

Even by estimating the required number of realizations to get a sufficiently small marginal error on the sample [11], the number of all dysfunctional combination sequences grows exponentially in the dimension of the considered circuit (N flip-flops implies at priori 2^N possible combinations), making practically impossible to extensively enumerate all the configurations.

2) Formal Fault Injection: Evaluating the hardware fault tolerance via formal methods has been a central topic in research during last decades [12], [13], due to the great advancement of both literature and testing of academia and industry. As computers and informatics tools [3] are nowadays advanced enough to attain satisfactory results, this strategy provides several advantages.

The Model Checking algorithm is designed to explore the whole set of states without missing any dysfunctional corner case, traversing exactly once each attainable configuration (see Fig. 1). This approach lends itself better than statistical simulation for fault injection, potentially providing a super-set of results in shorter times. We use here the formal engine Jaspergold [3] to determine the list of fragile flip-flops, with the support of the open software Goldmine [1] that automatically generates properties as fault detectors inside the module.

III. Formal Flow

In this section we present our methodology. We combine the reference simulation and Model Checking to perform fault injection on a digital block. Faults are detected whenever they violate the temporal properties, which are manually (at the output pins) or automatically generated (mined internally, using Goldmine).

Firstly, we run the golden simulation with maximized functional coverage, by stimulating the hardware with an exhaustive set of reference test-benches.

Secondly, such trace and the module's netlist are passed through Goldmine, that automatically generates a list of true assertions basing on the design and the simulation waveform (see III-B). A preliminary sanity formal proof is then exercised without faults, to select the properties that do not only hold in simulation, but also by Model Checking. From the reference simulation we also extract the corresponding module's state machine (see III-A).

Thirdly, from each state of the golden graph and for every flip-flop, the model checker automatically generates a bit-flip at the most convenient clock cycle, trying to violate the assertions. If the properties still hold (even in presence of the fault), then the flip-flop is proven to be SEU-resilient, and no further action is taken on it. Otherwise, at least one property is violated, meaning that the injected fault successfully propagated until contradicting the functional hardware protocol.

Finally, the fault injection results are collected and the masking capacity is calculated (see Eq. 1, as number of faults that propagate until the outputs, over the total causing a failure).

A. Reference Finite State Machine

The golden simulation is exercised encompassing the active control states of the design, for a given active functional mode. Being S the set of golden states and R the set of its N internal registers, the element $s \in S$ is defined as the vector of their values at a given time:

$s = \{r_0, \ldots, r_N\}, r \in R$.

By sampling at each clock cycle s and its transition during the golden simulation, we obtain the reference state machine (see Fig. 2).

Partitioning the protocol into its modes and phases (a phase can be the writing phase, the reading phase, *etc.*) selects specific sub-graphs of the reference state machine.

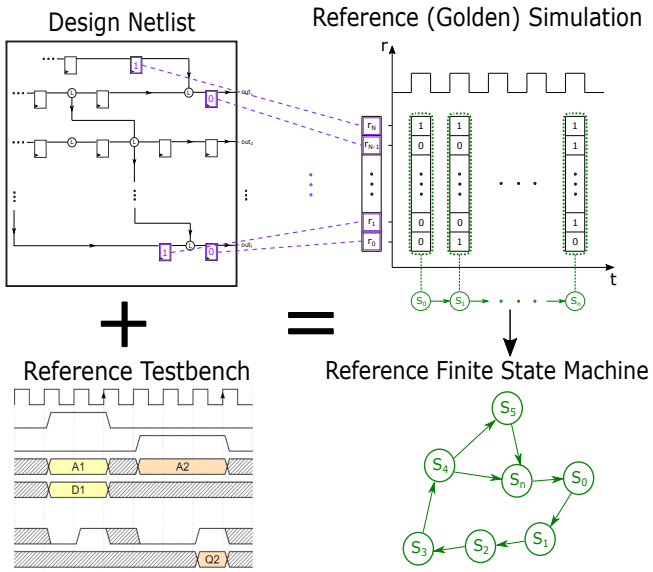


Fig. 2: Finite state machine encoded from the golden simulation

Some properties that are proven on the entire automaton, may be unreachable when constraining on specific sub-regions.

When a bit-flip (SEU) occurs during a chosen phase, assertions may be violated if initializing the model checker in some states (the fault propagates and a counterexample is found), and may hold when beginning from others (the SEU is absorbed and the design is fault tolerant). For a given phase, such peculiarity permits to know by Model Checking until when a flip-flop is dangerous.

B. Assertion generation with Goldmine

Goldmine is a tool to enhance verification efficiency, and for us, to automatically generate assertions on a design [14], [1]. Our version is nearly the one proposed by its creators, integrating some ad hoc modifications, described later on. The software accepts the design as a Verilog netlist (Goldmine is not configured to parse SystemVerilog without a licensed tool): for us, this is also the most appropriate framework for the subsequent fault injection process.

Firstly, the signals considered relevant to be part of the atomic formulas are selected (at the output boundary or internally). The netlist is then parsed, elaborated and analysed; the first difference between the open source version of Goldmine and our updated one consists in the option to store the linking phase result (the preliminary process of the static analysis [15]), reusable for further mining runs of the same module (which can be costly for large designs). There are two more considerations: the first is that buses may present cycles in their Cone of Influence (COI) (as they are not bit-blasted at this stage), likely

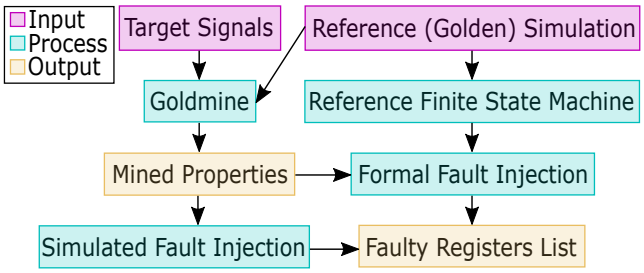


Fig. 3: Global flow overview

to cause infinite loops in the computation. This is solved by imposing Goldmine to skip already encountered edges while constructing the COIs. Secondly, the "complexity" and "importance" values (defined in [15] to rank the signal participating in the assertions as atomic formulas), are evaluated after concluding the full COI construction (thus according to the complete topological structure), using the average formula defined in [15].

To launch the mining process, we provide to Goldmine the golden simulation waveform (in `.vcd` format), the design netlist and target signals to generate the assertions. It is possible to significantly speed up and improve the quality of the mining outcome, by focusing on the control flow of the block, rather than its data path (*e.g.* by excluding data or reset signals, generating much more control-adherent assertions).

The golden reference simulation is necessary to reduce the combinatorial explosion of possibilities when building an assertions on the targets [16], optimizing the resources and the effort of the mining engine PRISM [17]. It identifies and couples the most strongly correlated signals with the targets, to build a set of always true assertions (with respect to the given input stimuli). We synthetically resume the mining algorithm steps as follows:

- The user selects the target signals to mine on.
- The user provides the golden simulation waveform to Goldmine.
- The algorithm selects the top contributor antecedents signals of the chosen targets, by evaluating the correlation coefficients for each couple antecedent-target (basing on their values during the simulation).
- By this, for each selected couple antecedent-target, a temporal property is built (of the form "if antecedent - then target"), assigning values to the variables as a function of their frequencies of occurrence.
- The process is repeated for each target signal.

Through progressive traversal of the configurations, the mining process successfully completes. The generated assertions are then verified with Jaspergold, keeping only the ones that turns out to be formally proven.

C. SEU Injection and Tolerance by Model Checking

SEU faults are modelled by a sabotaging signal named `fault`, connected to the target register, able to flip its output value when activated. Its waveform is a pulse, enabled only once during the injection process. Its behaviour is described by the temporal assumption h_f

$$h_f\text{: } \textbf{assume } \texttt{fault} \mapsto \textbf{always } (\textbf{!}\ \texttt{fault})$$

In simulated fault injection, we obtain statistical results on the tolerance level of each register (number of propagated errors over the total injected). Some works [11] have even measured the quality of the statistical results according to the number of injected faults. In the case of formal injection, we just know whether or not a register is tolerant. The outcome is much more reliable by Model Checking, as the flip-flop is proven to be totally fault invulnerable, which is not the case by simulation (it may exist an undiscovered combination of input stimuli that has been not exercised).

Intellectual Properties (IPs) of industrial designs are very versatile, able to support multiple behavioural modes, to be easily reused in different contexts. Given a chosen mode, several test benches are developed according to it, with the possibility to describe distinct phases as a function of the required operation to perform (*e.g.*, as the writing phase, the reading phase, *etc*). Similarly, distinct sets of assertions and assumptions describe different modes and phases of the protocol.

As it is not possible to fully determine the whole set of initial states for which a register is tolerant, the study is restricted to the ones given by the golden stimuli of the functional verification, whose quality is characterized by different coverage (code coverage, transition coverage, assertions coverage, ...). Focusing on such states provides an accurate overview of the fault-tolerant targets.

Let $\mathcal{C}$ be a circuit composed by N registers in R: its protocol specification is described by the sets $\mathcal{H}$, $\mathcal{A}$ of assumptions and assertions properties. Our method is described by the algorithm in Figure 4. Let $\mathcal{H}_m$ and $\mathcal{A}_m$ be respectively the restrictions of $\mathcal{H}$ and $\mathcal{A}$ to a given phase and a given mode of the circuit $\mathcal{C}$ (*e.g.*, $\mathcal{H}_m \subset \mathcal{H}$ and $\mathcal{A}_m \subset \mathcal{A}$) and let $\mathcal{J}$ be a set of golden stimuli.

```
1: procedure TOLERANCE(C, A_m, H_m, J)
2:     R = COI(A_m)
3:     G = EXTRACT_GRAPH(J, R)
4:     G = REMOVE_UNREACHABLE_STATES(G, H_m)
5:     for all r ∈ R do
6:         for all s ∈ S do
7:             M(r, s) = (C, s ⊨_{h_f} H_m → A_m)
```

Fig. 4: Formal Fault Injection

The first (line 2) step restricts the study to the registers in the COI of the assertions.

The second step (line 3) evaluates the list of golden states according to the golden stimuli in a given mode (and therefore of different phases). The stimuli correspond to the test-bench of the golden simulation: we record over

time the corresponding values of each register in $R_{\mathcal{A}_m}$. We obtain the set of states S (see III-A) that composes the reference state machine (from where each formal proof is initialized). If there is a transition between states s_i and s_j, it means that in the sequence evaluated by simulation, the state s_j was computed one cycle after s_i.

The third step (line 4) removes all unreachable states. The final step (line 7) is the formal proof for each golden state in S and each concerned flip-flop, by injection of the SEU through h_f. All fault injection results are stored into the Boolean matrix $M(r, s)$.

D. Fault Masking Capacity

Let be $R', R'' \subseteq R$ respectively the sets of fragile flip-flops with respect to output (boundary) and internal properties. The masking capacity index $\eta \in [0, 1]$ of the block is evaluated as:

$$\eta = 1 - \frac{|R'|}{|R' \cup R''|} \quad (1)$$

Such metric, extracted by Model Checking and encoded by matrix M, provides the fault masking capacity of the block, evaluating how much the circuit is able to absorb and delete single faults, during their propagation through the circuit.

IV. System Architecture

The flow is applied to the Serial Peripheral Interface (SPI [18]) and the Direct Memory Access (DMA) of a radiation test vehicle RISC-V System-on-Chip (SoC), shown in Figure 5. The system comprehends a CPU [19] and a RAM memory, communicating through an AHB bus [20]. Either the SPI de-serializes data and provides it to the CPU or the DMA, or the CPU or the DMA send data to the SPI, which serializes and forwards it to the external world. The SPI communicates with the AHB using an APB to AHB converter module, and interacts with the DMA through a four phase handshake protocol, allowing standalone data transfer (see Fig. 5).

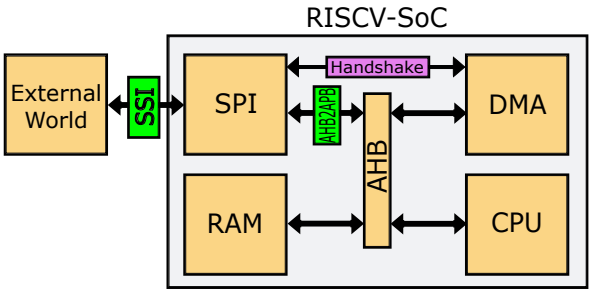


Fig. 5: RISCV SoC - System architecture

In this research, the whole analysis is performed at netlist level (after the Register Transfer Level synthesis).

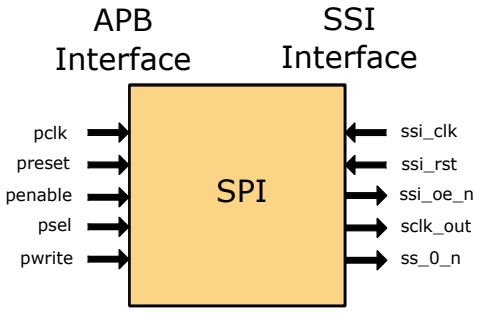


Fig. 6: SPI interfaces

V. Application Test Cases

The flow shown in Figure 3 is applied to the Serial Peripheral Interface (SPI), and to the Direct Memory Access Controller (DMA).

A. SPI block

The SPI module (see Fig. 6) is connected to a RISC-V processor using the Advanced Peripheral Bus (APB) [21], and to the external world through the Synchronous Serial Interface (SSI) [18]. Thus, the SPI has two interfaces:

The APB interface: This is controlled by the reset `preset` and clock `pclk`. Whenever a master (CPU or DMA) needs to communicate, it selects the SSI (`psel` is high) and asserts the signal `penable`. According to the value of `pwrite`, the SPI either reads or writes data on the registers.

The SSI interface: This is controlled by the reset `ssi_rst`, and a clock `ssi_clk`. Whenever the SPI needs to communicate with the external world, it asserts the signal `ssi_oe_n`, and selects the slave (`ss_o_n`) as the clock `sclk_out` is enabled.

1) SPI verification environment: We consider the functional mode of simultaneous data reception and transmission (RX_TX mode), assuming the information stored in the SPI and that the communication with the DMA is inactive. The two protocols (APB, SSI) can be manually modeled using System Verilog Assertions (SVA) language [8] with 4 assumptions and 10 assertions (see Table. I).

As an example, assumption h_1 indicates that `psel` is always asserted for two cycles. Assumption h_2 indicates that when the SPI is selected, the communication with the bus is enabled. Assertion a_1 states that if the clock `sclk_out` is enabled, then the external slave `ss_o_n` is not asserted. Assertion a_2 states that if `ss_o_n` is enabled, then the signal `ssi_oe_n` is not asserted.

2) Mining and fault injection on the SPI: We first generate assertions with Goldmine on the design boundaries only (*i.e.* the inputs and outputs of the SPI), to reproduce and compare the manual implementation of the protocol. This required some Goldmine manipulations, to make it able to create properties involving various outputs (not

h_1 : **assume** $rose psel $\mapsto$ psel[*2]
h_2 : **assume** $rose psel $\mapsto$ penable
a_1 : **assert** $rose sclk_out $\mapsto$!ss_o_n
a_2 : **assert** !ss_o_n $\mapsto$!ssi_oe_n

TABLE I: SPI manual properties

assert sclk_out ##2 sclk_out $\mapsto$!ss_o_n
assert penable ##2 sclk_out $\mapsto$!ss_o_n
assert !ss_o_n ##1 ss_o_n $\mapsto$!sclk_out
assert !ss_o_n ##2 ss_o_n $\mapsto$!sclk_out
assert ssi_oe_n $\mapsto$ ss_o_n
assert !ss_o_n $\mapsto$!ssi_oe_n

TABLE II: SPI boundary mined assertions

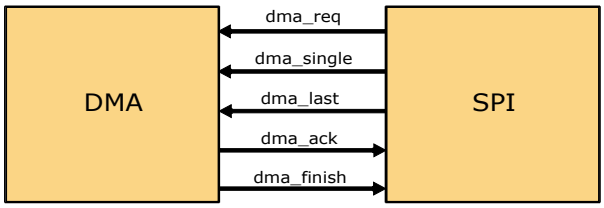


Fig. 7: DMA handshake interface

in the same COI), thus assertions that translate correlation, and not causality. After verifying the validity of the generated properties with Jaspergold, their relevance to the protocol is ensured manually. Obviously, reading and filtering out already proven properties is a much easier and faster task than generating the assertions by hand, interpreting the protocol and writing them from scratch.

The second and major study consists in providing to Goldmine internal signals as input for the property generation (using the tool more in its intended way). As before, properties are pruned by the model checker Jaspergold, then manually eliminating all the proven properties that are meaningless for the supposed protocol description.

B. DMA block

This module, extracted from the same SoC (see Fig. 5), is a core controller that transfers data from a source to a destination peripheral over one or more Advanced High-performance Buses (AHB) [22] (see Fig. 7).

The study concentrates on the handshaking interface in burst-transaction mode (see Fig. 7), to qualify the transfer test for multiple handshakes with the module for channel 2. When the SPI starts the handshake with the DMA, it first asserts signal `dma_req` in the desired channel, then `dma_ack` in the corresponding one. Finally, `dma_single` is asserted when the peripheral can transmit one more data items. As `dma_finish` and `dma_last` are not relevant for our scopes, the only output signal to focus on is `dma_ack`.

h_1 : **assume** !dma_req[i] && !dma_single[i] && !dma_last[i] for i in $(3, \ldots, 15)$
h_2 : **assume** dma_req[i] && dma_single[i] && !dma_last[i] for i in $(0, 1)$
h_3 : **assume** dma_req[2] && dma_ack[2] $\models\!\Rightarrow$!dma_req[2]
h_4 : **assume** $fell dma_req[2] $\models\!\Rightarrow$ always(!dma_ack[2])
a_1 : **assert** !dma_req[i]
a_2 : **assert** $fell dma_req[2] && dma_ack[2] $\models\!\Rightarrow$!dma_ack[2]

TABLE III: DMA manual properties

assert hc_dma_ack_dst[2] $\mapsto$ dma_ack[2]
assert !hc_dma_ack_dst[2] $\mapsto$!dma_ack[2]
assert tr_c_dst_md[1] $\models\!\Rightarrow$ dma_ack[2]

TABLE IV: DMA internal mined assertions

1) DMA verification environment: The inputs behaviour in the verification environment are driven by assumptions in SVA. To exhaustively describe the protocol, 6 properties are used (see Table III).

First, all DMA handshake channels from 3 to 15 are deactivated (h_1), then numbers 0, 1 are locked accordingly to the chosen test (h_2). Channel 2 is characterized by h_3 following the protocol in [22]. Finally, h_4 reduces the state space for the formal engine to traverse. Assertions a_1 and a_2 represent the expected block behaviour without error injection (such properties are proven via Model Checking in absence of faults).

2) Mining and fault injection on the DMA: To fully exploit the Goldmine potential for the treatment of correlated signals, and to broaden the variety of collected results, we directly focused on the internal block mining. Through the same procedure to select valid assertion used with the SPI, we obtained an exhaustive set of properties, some of which are shown in Table IV.

VI. Results

This section presents the formal and simulated SEU fault injection results for the SPI and DMA blocks, comparing the detection capacity for internal and boundary assertions.

The results put in evidence the increased rate of detection for faults in their early state of propagation, and by using Equation 1, we estimate (by Model checking) the masking capacity of the block. This provides a first metric to measure how long a fault can "survive" inside the circuit, before being absorbed by the logic. Moreover, by internal mining it is possible to detect the faults in their early stage of propagation, potentially canceling them by reinforcing the design with selective hardening techniques (Error Correcting Code, Triple Modular Redundancy, *etc.*).

Finally, the time performances of the formal and simulated approaches are confronted, highlighting the great efficiency of the first method over the second (all results

are re-scaled in 1 CPU Equivalent). All the analysis have been performed at netlist level, concerning the control flow of the blocks.

A. SPI results

1) Mining on SPI: By mining on the boundary, 6 nontrivial assertions were found (see Table II), 5 of which detecting at least one fault in formal injection. By going inside the module a total of 27 properties were formed, 19 of which composed by signals whose reproduction would be too complex for humans. The mining process took around 1 hour, including 45 minutes of reusable pre-processing, 10 minutes for simulation data parsing, and 5 minutes for the mining itself. As an example, Table II shows a list of mined assertions for the concerned output signals.

2) Fault injection on SPI: We consider as valid fault targets only the registers lying in the COI of at least one assertion. The data path is free to adopt all possible configurations, as in this work, we concern the control flow (this also avoids the well known state space explosion problem). We select the functional RX_TX mode during the reading phase on the registers, focusing on the SSI communication (nontrivial assertions concern this protocol part).

Focusing on control flip-flops, the COI computation lasts 5 minutes, getting 176 flip-flops out of the total 1000 (the remainder is in the the data part). Then, from the reference simulation (whose duration is about 30 seconds with Xcelium by Cadence) 27 distinct initial states were extracted in about 5 seconds.

In terms of Model Checking, not all of them are reachable for all properties, as the graph is extracted from a simulation (for instance, if there is a writing action from the APB to the SPI). Performing a first proof without faults, we skim all the non reachable states, remaining with a set of 25 elements.

We then deploy the methodology given in Figure 4, to obtain the fault tolerance metric of each flip-flop over time. The input signal `fault` representing the SEU is added to the register to target, modifying its truth table in the netlist modules' library.

The results concerning the usage of boundary mined assertions are very close to the manual ones (Fig. 8), only missing a group of 4 registers ($\{44, 45, 46, 47\}$). Nevertheless, the real versatility of Goldmine arises when mining inside the module, obtaining a super-set of critical flip-flops (Fig. 8). In this case, the set of assertions focuses with enhanced accuracy on the internal block functionality, and by injecting faults and detecting them at an early stage of propagation, a much larger family of dangerous flip-flops is found. There are some groups of registers in Figure 8 that do not manifest failures for all the initial states (e.g. $\{44, 45, 46, 47\}$ or $\{150, \ldots, 160\}$): they are formally proven to be robust after a finite amount of time. This nontrivial feature provides a crucial information, as it

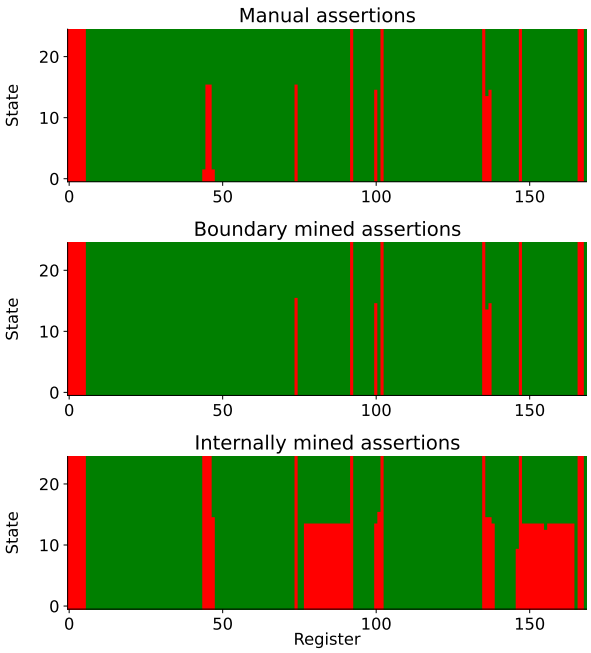


Fig. 8: SPI formal metrics

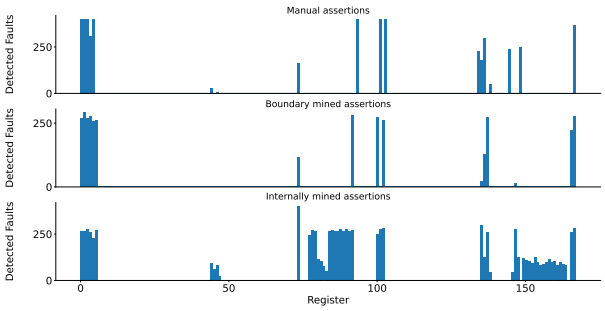


Fig. 9: SPI simulated injection histograms - 400 realizations per register

allows to exactly quantify (by Model Checking) until when a given flip-flop is dangerous. This is practically impossible to reproduce via statistical simulation, as it is test-bench dependent and because faults are randomly sampled over time.

To compare results, we run a standard Monte Carlo campaign injecting 400 SEUs per target, assuming a 95% confidence level and a margin of error of $\sim 1\%$.

Looking at Figure 8, over a total of 176 SEU targets, the 22 faulty registers found with boundary assertions are a subset of the 26 determined via manual implementation of the properties, which in turn are contained in the 55 calculated by internal mining. This is also evident by looking at the simulated counterparts in Fig. 9.

Assertions	Fault injection - 169 FFs	
	Formal	Simulated
Manual	26 FFs: $\sim$ 15 min	19 FFs : $\sim$ 58 h
Boundary mining - 1 h [i]	22 FFs : $\sim$ 15 min	22 FFs : $\sim$ 58 h
Internal mining - 1 h [i]	55 FFs : $\sim$ 155 min	55 FFs : $\sim$ 58 h
Masking capacity	$\eta \sim 53\%$	$\eta \sim 34\%$

[i] This represents the length of one complete mining run. Reusing previous processing, this can be reduced to 15 minutes.

TABLE V: Time performances and detection accuracy on SPI

We remark that the metrics displayed in Figures 8 and 9 are not directly comparable on the y-axis, as the first shows the formal matrix $M(r, s)$ described in Algorithm 4, while the second contains the cumulative histograms of the Monte Carlo faults campaigns. Even if matrix M is not an histogram as its simulated counterpart, the two images can be compared over the same x-axis (the set of dangerous flip-flops).

On the other hand, the families of 22 and 55 flip-flops found by formal (by mining on the boundaries and internally, respectively) coincide with the ones obtained by simulation, with the huge performance gain displayed in Table V (around 15 and 155 minutes vs 58 hours).

The time discrepancy between simulation and formal is mainly due to the distinct nature of the two approaches: the first consists in brute force realizations, depending on the input data and the chosen SEU injection time. On the other hand, Model Checking optimizes the computational effort (Depth First Search states traversal), looking for the shortest path counterexample.

Finally, we estimate the device masking capacity by applying Equation 1, getting $\eta \sim 53\%$ by formal. For completeness, we also report its approximated version via simulation $\eta \sim 34\%$.

B. DMA results

1) Mining on DMA: By internally mining on the DMA, a total of 3 assertions were found in 20 hours, more 15 trivial properties (`dma_ack` isn't asserted on other channels than 2, see Table IV).

2) Fault injection on DMA: By COI analysis 2857 flip-flops were found as valid fault targets (the processing lasted 2 hours), and from the reference simulation (25 seconds with Xcelium by Cadence) 12 distinct states (composing the reference state machine) were sampled. Again, the formal results obtained for the mined assertions provide a larger and more punctual set than the manual ones (see Figs. 11 and 10). Again, even if matrix M is not an histogram as its simulated counterpart, the two images can be compared over the same x-axis (the set of dangerous flip-flops).

Assertions	Fault injection - 2857 FFs	
	Formal	Simulated
Manual	51 FFs : $\sim$ 101 h	38 FFs : $\sim$ 150 h
Internal mining - 20 h[i]	72 FFs : $\sim$ 68 h	35 FFs : $\sim$ 150 h
Masking capacity	$\eta \sim 30\%$	–

[i] This represents the length of one complete mining run. Reusing previous processing, this can be reduced to 40 minutes.

TABLE VI: Time performances and detection accuracy on DMA

Indeed, looking at Figures 11 and 10, over a total of 2857 SEU targets, the 51 faulty registers found with boundary assertions are a subset of the 72 determined by internal mining (this is evident looking also at the approximated simulated counterparts in 11 and 10).

Once again, formal injection found the same results, in a much shorter time than simulation - performing 40 fault injections per target (see Table VI). It should be noted that the confidence interval is 47.26%, so some faulty registers may have been missed by simulation. Even if the simulation accuracy in this sense is not maximized (transcending from the scope of this work), the already evident time discrepancy with formal would have been enormously increased by some magnitude order, rendering the computation of simulation meaninglessly larger.

While for the SPI the same set of faulty registers was determined by Model Checking and Monte Carlo simulation, for the DMA the simulated campaign only found 35 faulty flip-flops, a subset of the 72 determined by formal injection (see Table VI). This confirms that the novel approach can be faster and also more accurate, also when using mined properties. Moreover, the simulation limits start to emerge for this larger block, while we see that formal keeps scaling well (on control flow), if suitably managed. The simulation inaccuracy also appears when calculating the masking capacity with Equation 1: by formal, we get a meaningful $\eta \sim 30\%$. Via simulation, such index is not defined ($38 > 35$): the test-bench choice, and the "blindness" of faults temporal sampling, limit the detection capacity of internally mined properties (35 dangerous flip-flops), which are unable to recover the statistic at the output boundary (38 dangerous flip-flops).

In this paper we applied the flow on specific operational modes of the SPI and the DMA. Such procedure can be easily extended to every device functionality, as assertion generation and fault injection are automated, scaling for larger blocks when separately applying the methodology on internal sub-modules.

VII. Conclusions

In this work, we automated the assertions generation by adapting Goldmine, to perform formal SEU fault injection into two medium-sized digital blocks of a RISCV-SoC for

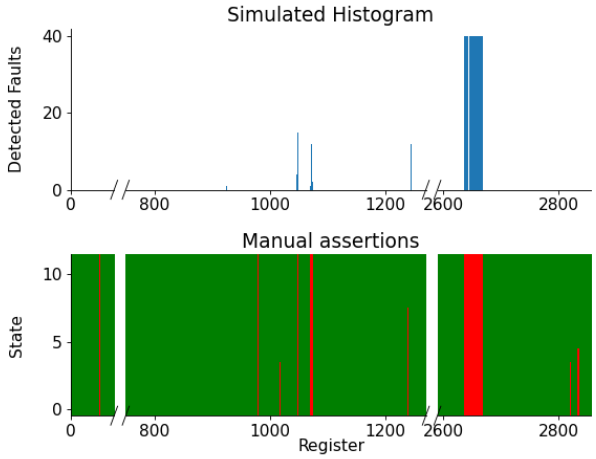


Fig. 10: DMA metrics for boundary manual assertions

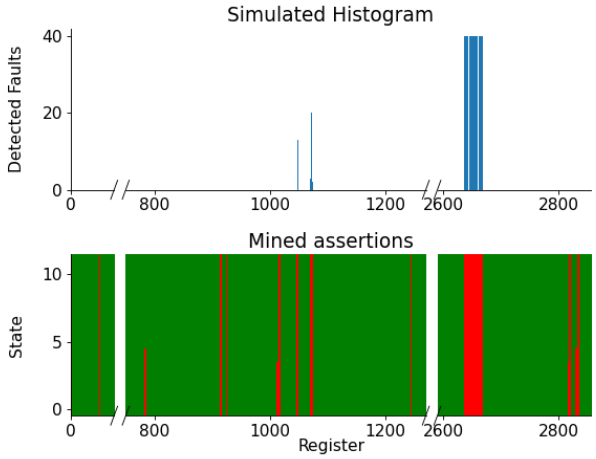


Fig. 11: DMA metrics for internally mined assertions

radiation automotive safety tests. This extends the procedure to evaluate the hardware fault tolerance to modules without an explicit or clear specification. In particular, Goldmine is here mainly employed to construct assertions inside the SPI and the DMA, to measure the fault masking capacity of the device. All the results are compared with the standard simulated counterpart, showing the huge gain in time and accuracy of our formal injection flow with mined properties.

This new method is indeed much more efficient than simulation for both SPI (internal mining: 55 flip-flops detected by formal in $\sim$ 155 minutes, compared to the same set found by simulation in $\sim$ 58 hours) and DMA (internal mining: 72 flip-flops detected by formal in $\sim$ 68 hours, compared to the subset of 38 found by simulation in 150 hours). This exhibits enhanced detection accuracy, and scalability (at most $\sim$ 155 minutes with 176 registers and 25 initial states for the SPI, and $\sim$ 101 hours with 2857 registers and 12 initial states for the DMA, with all results re-scaled in 1 CPU Equivalent).

Finally, the possibility to detect faults at their early stage inside the module allows to evaluate by Model Checking the module masking capacity η ($\sim$ 53% for the SPI, $\sim$ 30% for the DMA).

Further steps of this research will be the refinement of the fault masking capacity metric, and the flow extension to include the combination of several modules (to study fault propagation through interlaced systems). Moreover, the internal properties will be implemented in hardware as functional checkers, to halt the faults on their early stage of propagation, focusing on critical automotive safety applications.

References

[1] D. Pal, V. Dodeja, A. S. Kumar, and S. Vasudevan. Goldmine: A tool for enhancing verification productivity. [Online]. Available: https://woset-workshop.github.io/PDFs/2020/a25.pdf

[2] "Spyglass," 2022, march, 2023. [Online]. Available: https://www.synopsys.com/verification/static-and-formal-verification/spyglass.html

[3] "Jaspergold," 2022, march, 2023. [Online]. Available: https://www.cadence.com/en_US/home/tools/system-design-and-verification/formal-and-static-verification/jasper-gold-verification-platform/formal-property-verification-app.html

[4] P. Hauge, *IBM Journal of Research and Development, Volume 40, Number 1.* Riverton, NJ, United States: IBM Corp., 1996.

[5] "Egas architecture concept," 2023. [Online]. Available: https://nvdungx.github.io/EGAS-concept/

[6] A. Meyer, *Principles of Functional Verification.* Burlington: Newnes, 2004.

[7] T. Fiorucci, J.-M. Daveau, G. di Natale, and P. Roche, "Automated dysfunctional model extraction for model based safety assessment of digital systems," in *2021 IEEE 27th International Symposium on On-Line Testing and Robust System Design (IOLTS)*, 2021.

[8] J. Havliceck, *SVA: The Power of Assertions in SystemVerilog*, USA, 2014.

[9] J. Pin, *Mathematical Foundations of Automata Theory*, 2022.

[10] F. Renkin, P. Schlehuber-Caissier, A. Duret-Lutz, and A. Pommellet, "Effective reductions of mealy machines," M. R. Mousavi and A. Philippou, Eds. Cham: Springer International Publishing, 2022, pp. 114–130.

[11] R. Leveugle, A. Calvez, P. Maistri, and P. Vanhauwaert, "Statistical fault injection: Quantified error and confidence," in *2009 Design, Automation and Test in Europe Conference and Exhibition*, 2009.

[12] R. Leveugle, "A new approach for early dependability evaluation based on formal property checking and controlled mutations," in *11th IEEE International On-Line Testing Symposium*, 2005, pp. 260–265.

[13] S. Baarir, C. Braunstein, R. Clavel, E. Encrenaz, J.-M. Ilié, R. Leveugle, I. Mounier, L. Pierre, and D. Poitrenaud, "Complementary formal approaches for dependability analysis," in *2009 24th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, 2009, pp. 331–339.

[14] S. Vasudevan, D. Sheridan, S. Patel, D. Tcheng, B. Tuohy, and D. Johnson, "Goldmine: Automatic assertion generation using data mining and static analysis," in *2010 Design, Automation & Test in Europe Conference & Exhibition*, 2010, pp. 626–629.

[15] D. Pal, S. Offenberger, and S. Vasudevan, "Assertion ranking using rtl source code analysis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39, no. 8, pp. 1711–1724, 2020.

[16] S. Hertz, D. Sheridan, and S. Vasudevan, "Mining hardware assertions with guidance from static analysis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 32, no. 6, pp. 952–965, 2013.
[17] J. Cendrowska, "Prism: An algorithm for inducing modular rules," *International Journal of Man-Machine Studies*, vol. 27, no. 4, pp. 349–370, 1987. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S0020737387800032
[18] Synopsys, *Designware DW_apb_ssi Databook*, Mountain View, California, USA, 2015.
[19] "Syntacore scr1," 2023. [Online]. Available: https://syntacore.com/page/products/processor-ip/scr1
[20] ARM, *AMBA AHB Protocol Specification*, 2018.
[21] Vani.R.M, *Design of AMBA Based AHB2APB Bridge*, Bulbarga University, Bulbarga, 2010.
[22] Synopsys, *DesignWare DW_ahb_dmac Databook*, Mountain View, California, USA, 2014.