SEU injection framework for radiation-tolerant ASICs, a formal verification approach

M. Lupi[a], A. Pulli[a]

[a]CERN, Geneva (CH)

matteo.lupi@cern.ch

Introduction

The high flux of ionising particles in the detectors at CERN LHC poses a challenge to the operation of digital electronics as they are responsible for **Single Event Effects** (SEEs) [1, 2]. An effective approach to limit the SEEs susceptibility consists in applying Triple Modular Redundancy (TMR) to the design. The design triplication can be partially automated by tools, such as TMR Generator (TMRG) [3].

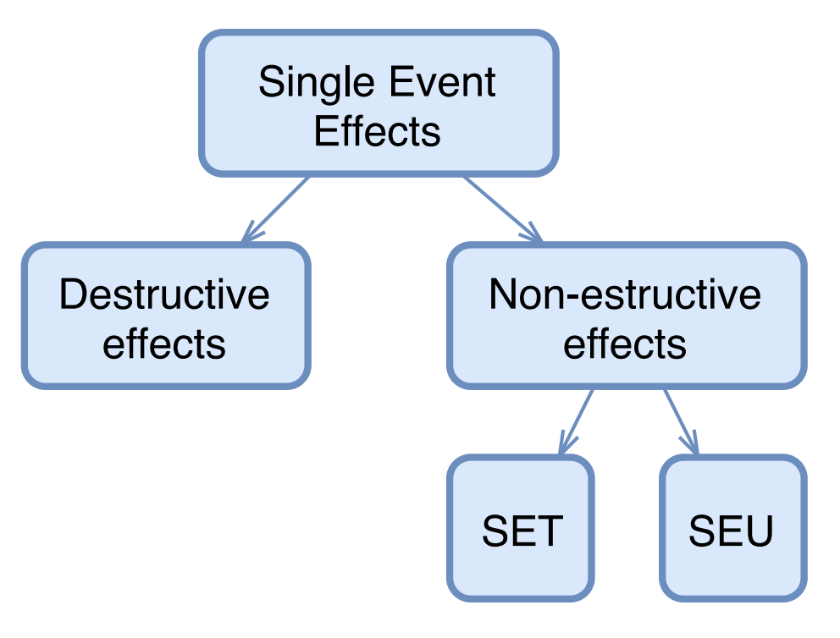

Fig. 1: Categorisation of Single Event Effects

Even with automated tools there is still a margin for human error: the Register Transfer Level (RTL) to the final netlist is a long process involving multiple engineers and tools. The verification of the correct TMR implementation is a fundamental step to achieving first-time-right silicon.

When verifying the triplication, there are two questions to be answered:

- Did we triplicate everything that **needs to be triplicated**?
- Did we triplicate everything that **we intended to triplicate**?

Formal verification

A common approach to verification is simulation. Stimuli are provided to the Design Under Test (DUT) and its outputs are compared with those of a reference model. If the two match, the system is working as intended.

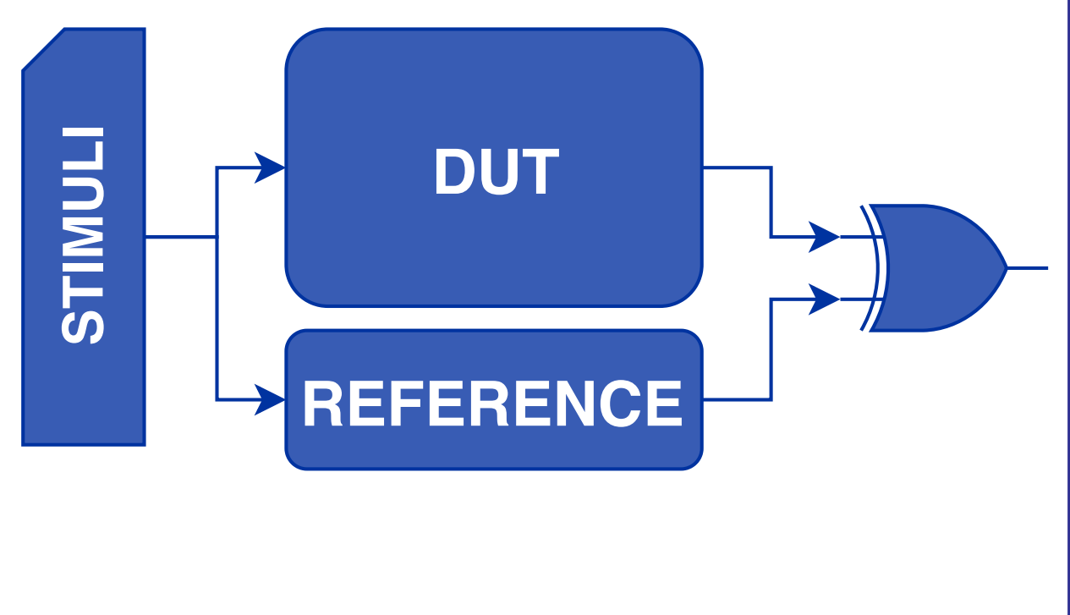

Fig. 2: Verification by simulation

Formal verification is instead an orthogonal approach. The design's expected behaviour is described through assumptions (on its input) and assertions (on its outputs or internal nodes). The formal verification tool is then mathematically proving that, given the assumptions, the assertions always hold (proven). If the assertion does not always hold, then a counterexample (CEX) is produced.

The main advantages:

- **Early start** (no testbench required)
- **Proof is exhaustive** given that the assumptions hold
- **Fast debug** (CEX correspond to a specific property)

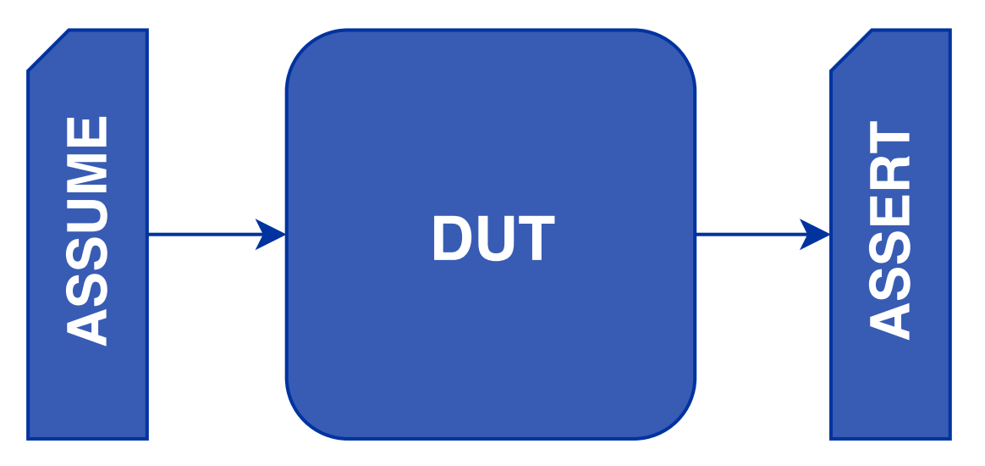

Fig. 3: Formal verification

One of the main issues with formal verification is **proof convergence**. The number of states to be explored by the formal tool is exponential with the number of clock cycles of the assertions/assumptions.

Framework

We believe that the question of proving correct triplication can be answered with very high certainty using formal methods, without any user intervention. The framework presented in this contribution operates in two phases: a **generation phase** and a **proof phase**. In the generation phase, the design is analysed and the assumptions and assertions are generated. In the proof phase, the formal proof of the aforementioned assertions is run.

The generation phase is further subdivided into three steps: **elaboration**, **node filtering** and **extraction**, and **assertion generation**.
During the elaboration, the netlist and a fault intent are provided to a commercial fault simulator. The netlist can be a synthesised or an implemented netlist. The fault intent for this kind of verification is always set to SEUs, i.e. the tool will extract all the nodes which can be subject to SEU.

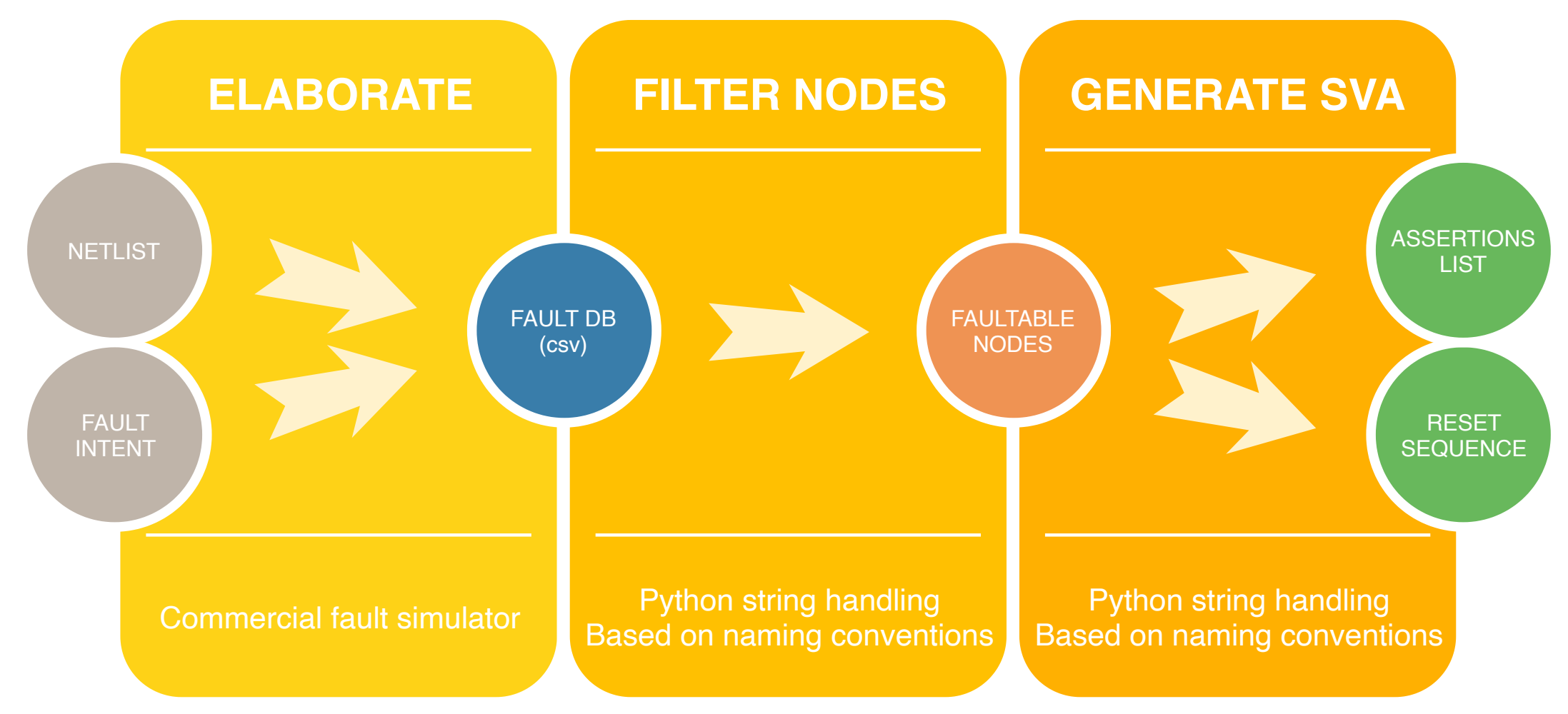

Fig. 4: Generation phase

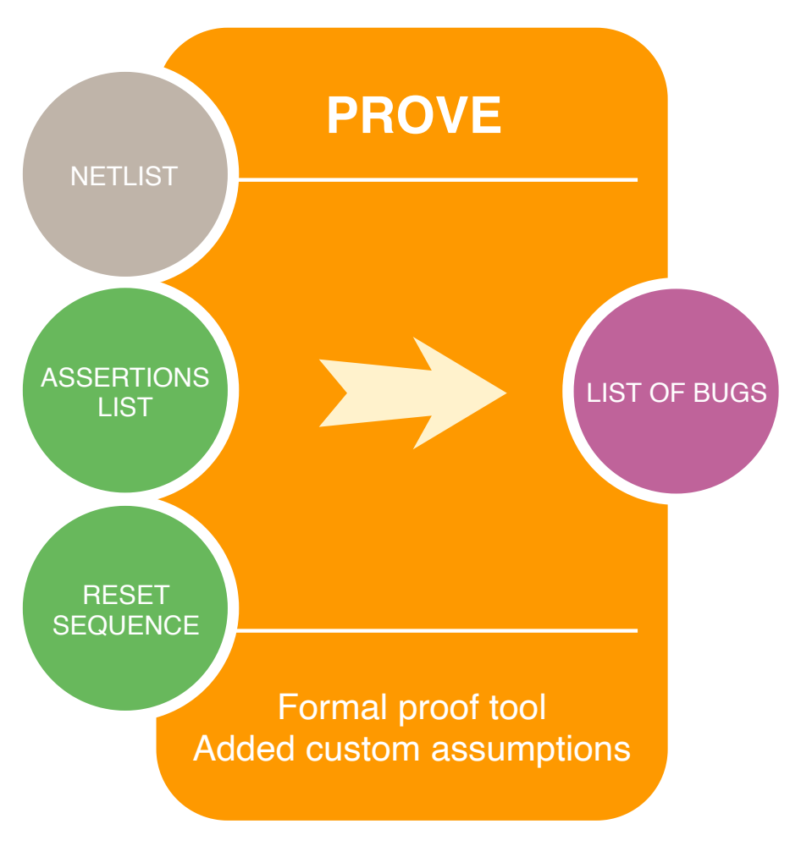

Fig. 5: Proof phase

Once the netlist is elaborated with the fault intent, the faultable nodes are extracted from the elaborated database using a commercially available fault set generator. The list of faultable nodes is provided in CSV format with the hierarchical path to the nodes.

The faultable nodes need to be refined such that flip-flops (DFFs) and latches (DLAs) can be extracted. The nodes also need to be divided between triplicated and non-triplicated nodes. A set of python scripts are used to identify the triplicated nodes through string handling and regular expressions. The tool leverages the usage of naming conventions for signals, which is implemented by TMRG: all the triplicated signals will be present three times ending in A, B, and C.

In the last step of the generation phase, the filtered list of faultable nodes is used to generate the set of assumptions and assertions together with a reset sequence. The formal tool runs a first reset of the design and it assumes that the design is no longer reset. To correctly verify the design, fault injection is not run during the reset phase. This is achieved by providing the tool with a reset sequence which prevents any SEU to be injected during the reset.

In the run phase, the netlist is provided to the formal tool together with the artefacts from the generation phase. The provided design needs to be synthesizable. In the case of hard macros present in the design, which cannot be modelled using synthesizable code, an option for black-boxing them is available.

It is to be noted that in principle, there is no need to know or specify the behaviour of the inputs to the design. The formal tool **can run without any assumption** on the behaviour of the inputs. In some cases, however, adding assumptions on the input signals' behaviour can be useful to reduce the space of states to be explored by the formal tool, providing faster proof.

The SEUs are modelled by **modifying the standard cells primitives** for DFFs and DLAs. An internal pin, denominated `q_seu` is added to the primitive of the sequential logic elements. The pin is left floating in the standard cell, allowing the formal tool to control it. This is effectively one input to the design. When this signal is asserted the content of the sequential logic element is toggled, emulating the effect of an SEU.

Properties

Assumptions (pseudo-code)

```
• $onehot0({regA.q_seu,regB.q_seu,regC.q_seu})

• |{regA.q_seu,regB.q_seu,regC.q_seu} |=> \\
      ({regA.q_seu,regB.q_seu,regC.q_seu}==3'b000)[*1]
```

Assertions (pseudo-code)

```
• regA.q_seu |=> ##1 (regA.q == regB.q == regC.q)

• regB.q_seu |=> ##1 (regA.q == regB.q == regC.q)

• regC.q_seu |=> ##1 (regA.q == regB.q == regC.q)
```

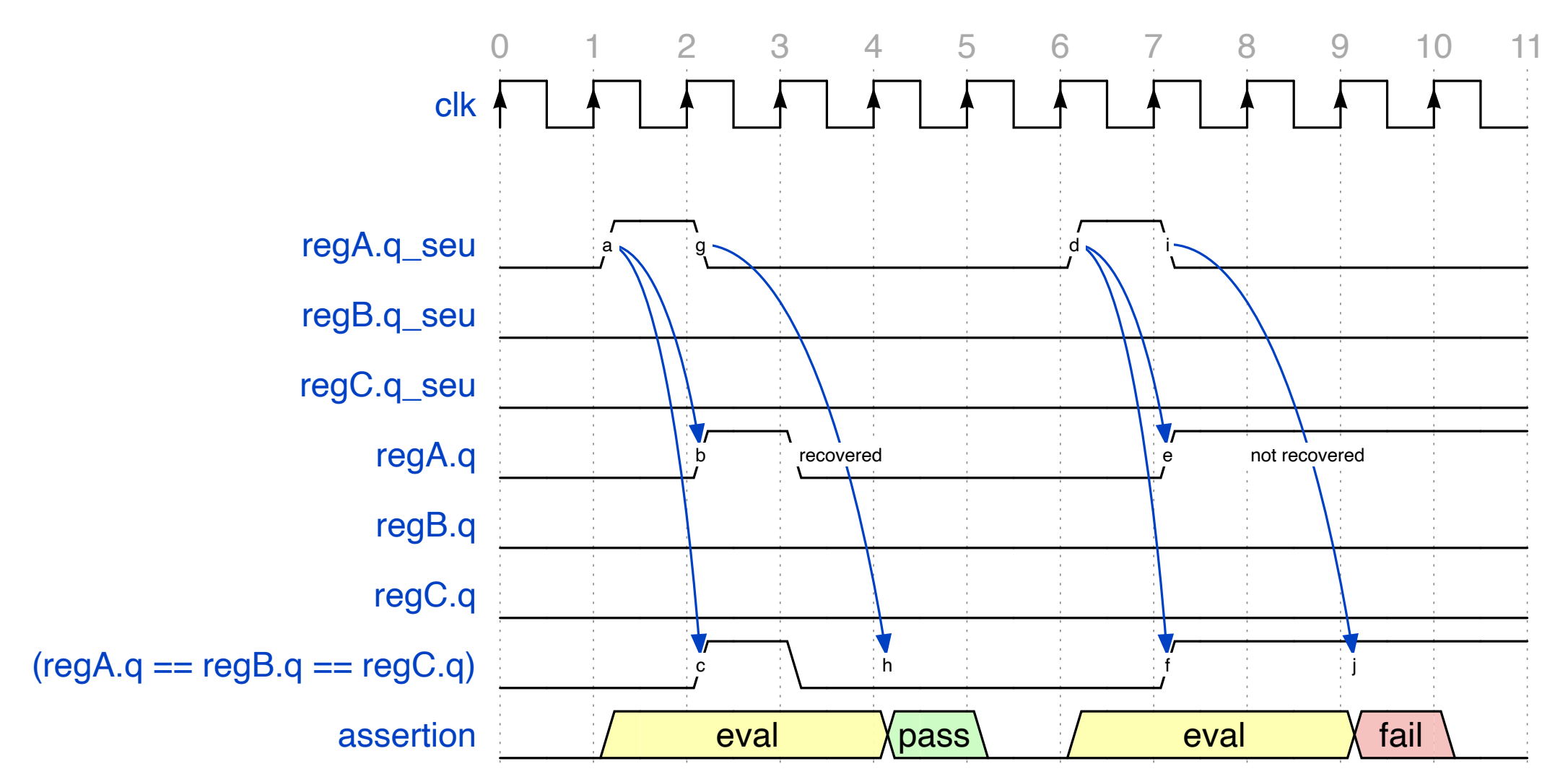

Fig. 6: Example of triplication assertions

Results

The presented solution was applied to multiple designs with different characteristics. Table 1 reports the design complexity versus proof time. The proofs were run on a machine with two AMD EPYC™7302 processors (for a total of 32 cores running at 3 GHz).

During the verification phase, it was noted that the **proof time is highly dependent on the number of failing assertions**. The formal tool exploits triplication errors to produce CEXs. This has an impact on the Cone Of Influence (COI) of the assertion which results in a deeper (i.e. longer) minimal proof. This affects the memory usage and the CPU time required to prove (or find a CEX). In some cases, the verification benefitted from fixing the bugs already identified, re-running implementation, and re-running the proof.

Since most of the bugs were already present in RTL, it is **beneficial to run the formal proof on the synthesised netlist**, w.r.t. running on the implemented one. The turn-around time between identifying the bug and testing for its resolution is sensibly shorter.

Tab. 1: Different design verified with the framework

ASIC	Module name	Design complexity[bit] non-triplicated	Design complexity[bit] triplicated	Proof time	Issues Identified
EXP28	Slow control	-	319	90 s	0
ECOND	Word aligner	-	797	80 s	12
ECOND	Slow control[a]	-	573	0.6 h	4
ECOND	Ping-pong SRAM[b,c]	61 440	5166	2.2 h	2
ECOND	Packet tracking	-	10 798	2.5 h	14
ECOND	Channel processors	-	20 441	35 h	2
ECOND	Formatter & output buffer[b,c,d]	98 304	5152	36 h	7

[a] clock gating [b] SRAM macro [c] multi-clock [d] selective triplication

References

[1] L. D. Edmonds, C. E. Barnes, and L. Z. Scheick. An introduction to space radiation effects on microelectronics. Technical Report NASA-JPL-00-06, NASA-JPL, May 2000.
[2] C. Claeys and E. Simoen. *Radiation Effects in Advanced Semiconductor Materials and Devices.* Springer, 2002.
[3] S. Kulis. Single event effects mitigation with TMRG tool. *Journal of Instrumentation*, 12(01):C01082–C01082, jan 2017.

EP-ESE
ELECTRONIC SYSTEMS FOR EXPERIMENTS

CHIPS