A formally verified circuit transformation to tolerate SEMTs

Vincent Bonczak, Pascal Fradet

A formally verified circuit transformation to tolerate SEMTs

Vincent Bonczak, Pascal Fradet

RESEARCH REPORT
N° 9523
Oct. 2023
Project-Team Spades

ISSN 0249-6399 ISRN INRIA/RR--9523--FR+ENG

A formally verified circuit transformation to tolerate SEMTs

Vincent Bonczak*, Pascal Fradet†

Project-Team Spades

Research Report n° 9523 — Oct. 2023 — 26 pages

Abstract: Digital circuits are subject to transient faults caused by high-energy particles. A particle hit may produce a voltage pulse that propagates through the circuit and upsets its logical state and output. Most fault-tolerance techniques consider this kind of single event transients (SETs). However, as technology scales down, a single particle becomes likely to induce transients faults in several adjacent components. These single-event multiple transients (SEMTs) are becoming a major issue for modern digital circuits.
In this paper, we follow a formal approach to study fault-tolerance *w.r.t.* SEMTs. We first show how to formalize SEMTs provided that the layout of the circuit is known. We show that the standard triple modular redundancy (TMR) technique can be modified so that, along with some placement constraints, it completely masks SEMTs. In order to prove such a fault-tolerance property for all circuits with unknown layouts, we define a fault-model representing all possible SEMTs for all TMR circuits.
Circuits are expressed in LDDL, our gate-level hardware description language. The modified TMR technique is described as a circuit transformation in LDDL, and the fault models for SEMTs are specified as particular semantics of LDDL. We show that for any circuit its transformed version masks all faults of the considered SEMT fault model. All this development is formalized in the Coq proof assistant where fault-tolerance properties are expressed and formally proved.

Key-words: Fault-tolerance, triple modular redundancy, single-event multiple-transients, circuit transformation, certification, Coq.

* ENS Paris-Saclay
† Univ. Grenoble Alpes, Inria, CNRS, Grenoble INP, LIG

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Une transformation de circuits vérifiée formellement pour tolérer les SEMT

Résumé :

Les circuits numériques sont sujets à des fautes transitoires causées par des particules à haute énergie. L'impact d'une particule peut produire une impulsion de tension qui, se propageant dans le circuit, corrompt sa mémoire et sa sortie. La plupart des techniques de tolérance aux fautes prennent en compte ce type de fautes (SETs). Cependant, à mesure de la miniaturisation des semi-conducteurs, une seule particule est susceptible d'impacter plusieurs éléments adjacents. Ces fautes multiples (SEMT) deviennent un problème majeur pour les circuits modernes.

Dans cet article, nous suivons une approche formelle pour étudier la tolérance aux fautes de type SEMT. Nous montrons comment formaliser les SEMT en supposant connu l'agencement du circuit. Nous montrons ensuite que la technique standard de triple redondance modulaire (TMR) peut être modifiée pour, avec quelques contraintes de placement, qu'elle tolère les SEMT. Afin de prouver cette propriété de tolérance aux fautes pour tout circuit (dont l'agencement précis est inconnu), nous définissons un modèle de fautes incluant tous les SEMT possibles pour tous les circuits TMR.

Les circuits sont exprimés dans notre langage de description de matériel LDDL. La technique TMR modifiée est décrite comme une transformation de circuit dans LDDL et les modèles de fautes pour SEMT sont spécifiés en tant que sémantique dédiée de LDDL. Nous montrons que pour tout circuit, sa version transformée tolère le modèle de fautes SEMT considéré. Tous ces développements sont formalisés dans l'assistant de preuve Coq. Les propriétés de tolérance aux fautes y sont exprimées et prouvées formellement.

Mots-clés : Tolérance aux fautes, redondance modulaire triple, défauts transitoires multiples, transformation de circuit, certification, Coq.

Contents

1 Introduction

In digital circuits, transient faults occur when a high-energy particle, such as a cosmic ray or a radioactive decay product, strikes the circuit, leading to temporary electrical charges. These charges can upset the logical state of the circuit, causing soft (non-destructive, non-permanent) errors.

The fault-tolerance of soft errors has become a design characteristic of circuits as important as performance and power consumption [24]. This is especially true in critical domains such as aerospace and nuclear power plants where digital devices are typically exposed to high energy particles such as neutrons and protons. Fault-tolerance techniques must be used to prevent critical failure of such systems.

A particle strike may cause several types of transient faults. A Single-Event Upset (SEU) changes the value of a single flip-flop. A more general type of faults is Single-Event Transients (SETs). In that case, the particle causes a voltage pulse or glitch that propagates through the combinational circuit and eventually corrupts one or several flip-flops or memory cells. As technology scales down in terms of transistor size and spacing, the susceptibility to SETs grows. Moreover, it is estimated that below 90nm[1] a single particle becomes likely to induce transient faults in several adjacent components [22]. These faults, called Single-Event Multiple Transients (SEMTs), subsumes SEUs and SETs and are becoming a major issue for modern digital circuits.

The most widely-used methods to make circuits fault-tolerant rely on hardware redundancy. Triple Modular Redundancy (TMR) [30] remains the most popular technique along with Finite State Machine (FSM) encoding (one hot, hamming, *etc.*). A TMR circuit is triplicated and makes use of majority voters to mask errors. There are several versions of TMR ranging from a single voter at the primary outputs to triplicated voters after each memory cell. This last version, called full TMR tolerates any SET provided that they do not occur more than once every two cycles.

Contrary to most works, which are based on experiments (fault injection) and evaluate reliability properties in terms of probabilities, our approach is to prove that a fault-tolerance technique masks all faults of a specific fault model.

In this article, we study how full TMR can be adapted to tolerate SEMTs. We follow the formal approach that we have proposed to prove several fault-tolerance techniques for SETs [8]:

- circuits are specified using LDDL, a gate-level functional description language;
- fault models, which specify the kind and occurrences of faults to be masked, are formalized as an LDDL semantics;
- the fault-tolerance technique is described as an automatic circuit transformation defined on the syntax of LDDL;
- the proof that any transformed circuit tolerates the fault model is conducted in the Coq proof assistant [21].

Compared to SETs, SEMTs pose specific problems. They cause transients only on adjacent elements and their effects depend on the layout of the circuit. This makes the specification of the fault model more difficult. It also complicates the definition and proof of fault-tolerance properties. Indeed, we must prove that all possible transformed circuits tolerate SEMTs. The properties should not rely on specific layout but rather on generic constraints about all layouts. This implies that we must consider that, apart from these constraints, all other elements can be

[1] This measure refers to the size of transistors that can be etched in a circuit (each logic gate is composed of several of these transistors).

impacted by an SEMT. An obvious constraint for TMR is that the redundant copies are distant from each other. Indeed, an SEMT could otherwise corrupt two copies and that would make majority voting unable to mask errors. But the three copies of TMR have also interconnections that must be taken care of as well.

The article is organized as follows. We introduce the background useful for our work in Sec. 2. We describe in particular the syntax and semantics of LDDL, our circuit description language. Sec. 3 explains how faults and fault models can be formalized. In Sec. 4, we present SEMTs, their formal specification taking into account the layout of the circuit and the fault models we consider. In Sec. 5, we describe the standard full TMR as a circuit transformation. We then describe the adjustments and layout constraints needed so that it tolerates SEMTs. Sec. 6 presents the general fault-tolerance property of the modified TMR satisfied. We outline its formal proof by describing its structure and the main lemmas. Finally, Sec. 7 presents related work and Sec. 8 concludes.

Throughout this article, we use standard mathematical and semantic notations. The corresponding Coq specifications and proofs are available online [1].

2 LDDL: A Low-level Dependent Description Language

Our approach consists in describing circuits at the gate level using a purely functional language called LDDL (Low-level Dependent Description Language) inspired from Sheeran's combinator-based languages such as muFP [29] and Ruby [17]. The operational semantics of LDDL describes the stream of outputs that a circuit returns for a given stream of inputs under normal operation. The fault model is expressed as a fault semantics describing the streams of outputs that a circuit may return for a given stream of inputs under all possible faults in the model. Fault-tolerance techniques are described as circuits transformations expressed on LDDL syntax. Fault-tolerance properties can then be expressed by relating the behavior of the source circuit under normal operation with the behavior of the transformed circuit under the fault model.

The types of LDDL along with the language syntax ensure that circuits are well-formed *by construction*: gates are correctly plugged, there are no dangling wires nor combinational loops.

Contrary to muFP or Ruby, our primary goal is not to make the description of circuits easy but to keep the language as simple and minimal as possible to facilitate formal proofs. Our language contains only 3 logical gates, 5 plugs and 3 combining forms. It is best seen as a low-level core language used as the object code of a synthesis tool. The following sections are devoted to present its syntax and semantics.

2.1 Syntax

Digital circuits operate on discrete signals whose value is either 0 or 1. A circuit takes as input and returns as output buses of signals. A *bus* of signals is described by the following type

$$B := \omega \mid (B_1 * B_2)$$

A bus is either a single signal (ω) or a pair of buses. In other terms, buses are defined as nested pairs.

A circuit is either a logic gate, a plug, or a composition of circuits. The constructors of LDDL annotated with their types are gathered in Fig. 1.

The sets of logical gates and plugs are minimal but expressive enough to specify any combinational circuit. Actually, extending those sets would have a marginal impact on the proofs.

The gate NOT is a unary operator taking and returning a single signal as indicated by its type $\omega \to \omega$. Gates AND and OR, with type $(\omega * \omega) \to \omega$, are binary gates taking a bus made of two signals and returning one signal.

Gates

$$
\begin{array}{rlcl}
G ::= & \textsc{not} & : & \omega \to \omega \\
\mid & \textsc{and} & : & (\omega * \omega) \to \omega \\
\mid & \textsc{or} & : & (\omega * \omega) \to \omega
\end{array}
$$

Plugs

$$
\begin{array}{rlcl}
P ::= & \textsc{id} & : & \alpha \to \alpha \\
\mid & \textsc{fork} & : & \alpha \to (\alpha * \alpha) \\
\mid & \textsc{swap} & : & (\alpha * \beta) \to (\beta * \alpha) \\
\mid & \textsc{lsh} & : & ((\alpha * \beta) * \gamma) \to (\alpha * (\beta * \gamma)) \\
\mid & \textsc{rsh} & : & (\alpha * (\beta * \gamma)) \to ((\alpha * \beta) * \gamma)
\end{array}
$$

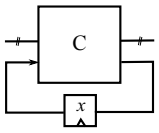
Circuits

$$
\begin{array}{rlcll}
C ::= & G & : & \alpha \to \beta & \\
\mid & P & : & \alpha \to \beta & \\
\mid & C_1 \multimap C_2 & : & \alpha \to \gamma & \text{if } C_1 : \alpha \to \beta \text{ and } C_2 : \beta \to \gamma \\
\mid & [\![C_1, C_2]\!] & : & (\alpha * \beta) \to (\gamma * \delta) & \text{if } C_1 : \alpha \to \gamma \text{ and } C_2 : \beta \to \delta \\
\mid & \boxed{x} \text{—} C & : & \alpha \to \beta & \text{if } x : \mathsf{bool} \text{ and } C : (\alpha * \omega) \to (\beta * \omega)
\end{array}
$$

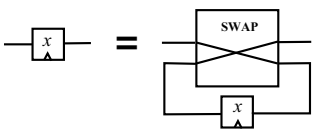
Figure 1 – LDDL syntax and types

Plugs, used to express (re)wiring, are polymorphic functions that duplicate or reorder buses of signals: ID leaves its input bus unchanged, FORK duplicates its input bus, SWAP inverts the order of its two input buses, LSH and RSH modify the grouping of their three input buses. Their polymorphic types describe precisely their semantics. Note that plugs formalize inter-connections but do not appear concretely in circuits.

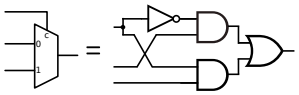
A circuit is either a gate, a plug, a sequential composition of two circuits ($\cdot \multimap \cdot$), a parallel composition of two circuits ($[\![.,.]\!]$), or a composition of a circuit with a cell (flip-flop) within a feedback loop ($\boxed{.}$— $\cdot$). The typing of the sequential operator ensures that the output bus of the first circuit has the same type as the input bus of the second one. The typing of the parallel operator expresses the fact that the inputs (resp. outputs) of the resulting circuit is made of the inputs (resp. outputs) of the two sub-circuits. The last operator is the only way to introduce feedback loops in the circuit. $\boxed{x}$—C is better seen graphically as the following circuit



The circuit C can have an arbitrary input/output bus but it also takes and returns a single wire connected to a memory cell set to the Boolean value x (tt or ff). This operator ensures that any loop contains a cell. It prevents combinational loops by construction. Note that it does not impose all cells to be within loops. A simple cell without feedback can be expressed as $\boxed{x}$—SWAP:



To illustrate the language, a multiplexer



can be expressed in LDDL as the following expression of type $(\omega * (\omega * \omega)) \to \omega$.

$$[\![\text{FORK}, \text{SWAP}]\!] \multimap \text{LSH} \multimap [\![\text{NOT}, \text{RSH} \multimap \text{SWAP}]\!] \multimap \text{RSH} \multimap [\![\text{AND}, \text{AND}]\!] \multimap \text{OR}$$

This LDDL expression takes three signals $(a_1, (a_2, a_3))$ and returns a_2 if $a_1 = 0$ and a_3 otherwise. In logical form it computes $(\neg a_1 \wedge a_2) \vee (a_1 \wedge a_3)$. Its evaluation proceeds as follows:

	$[\![\text{FORK}, \text{SWAP}]\!] \multimap \text{LSH} \multimap [\![\text{NOT}, \text{RSH} \multimap \text{SWAP}]\!] \multimap \text{RSH} \multimap [\![\text{AND}, \text{AND}]\!] \multimap \text{OR}$	$(a_1, (a_2, a_3))$
$=$	$\text{LSH} \multimap [\![\text{NOT}, \text{RSH} \multimap \text{SWAP}]\!] \multimap \text{RSH} \multimap [\![\text{AND}, \text{AND}]\!] \multimap \text{OR}$	$((a_1, a_1), (a_3, a_2))$
$=$	$[\![\text{NOT}, \text{RSH} \multimap \text{SWAP}]\!] \multimap \text{RSH} \multimap [\![\text{AND}, \text{AND}]\!] \multimap \text{OR}$	$(a_1, (a_1, (a_3, a_2)))$
$=$	$\text{RSH} \multimap [\![\text{AND}, \text{AND}]\!] \multimap \text{OR}$	$(\neg a_1, (a_2, (a_1, a_3)))$
$=$	$[\![\text{AND}, \text{AND}]\!] \multimap \text{OR}$	$((\neg a_1, a_2), (a_1, a_3)))$
$=$	$(\neg a_1 \wedge a_2) \vee (a_1 \wedge a_3)$	

LDDL is best seen as a back-end language produced by synthesis tools. It is simple and expressive enough. Its types make inputs and outputs of each sub-circuit explicit and ensure that all circuits are well-formed.

In the following, we often use the term *components* to denote the gates and cells of the circuit and *elements* to denote its gates, cells and wires.

2.2 Semantics

The values of signals are either 0 or 1 which are latched in memory cells as ff and tt respectively. The semantics of gates and plugs is as expected. It is given by functions denoted by $[\![.]\!]$:

$$[\![\text{NOT}]\!]\, 0 = 1 \quad [\![\text{NOT}]\!]\, 1 = 0 \quad [\![\text{AND}]\!](1,1) = 1 \quad [\![\text{AND}]\!](0,1) = 0 \quad \ldots$$
$$[\![\text{ID}]\!]\, x = x \quad [\![\text{FORK}]\!]\, x = (x,x) \quad [\![\text{SWAP}]\!](x,y) = (y,x) \quad [\![\text{LSH}]\!]((x,y),z) = (x,(y,z)) \quad \ldots$$

The semantics of circuits is described by the inductive predicate step relating a source circuit C (of type $\alpha \to \beta$) and an input a (of type α) to an ouput b (of type β) and a resulting circuit C' (of type $\alpha \to \beta$). The circuits C and C' are structurally identical and can differ only by the values of their cells. The expression $\mathsf{step}\ C\ a\ b\ C'$ can be read as "after one clock cycle, the circuit C applied to the inputs a may produce the outputs b and the new circuit (state) C' ".

The semantics rules are gathered in Fig. 2.

$$\text{Gate}\ \frac{[\![G]\!]\, a = b}{\mathsf{step}\ G\ a\ b\ G} \qquad \text{Plug}\ \frac{[\![P]\!]\, a = b}{\mathsf{step}\ P\ a\ b\ P}$$

$$\text{Seq}\ \frac{\mathsf{step}\ C_1\ a\ b\ C_1' \qquad \mathsf{step}\ C_2\ b\ c\ C_2'}{\mathsf{step}\ (C_1 \multimap C_2)\ a\ c\ (C_1' \multimap C_2')} \qquad \text{Par}\ \frac{\mathsf{step}\ C_1\ a\ c\ C_1' \qquad \mathsf{step}\ C_2\ b\ d\ C_2'}{\mathsf{step}\ [\![C_1, C_2]\!]\ (a,b)\ (c,d)\ [\![C_1', C_2']\!]}$$

$$\text{Cell}\ \frac{\mathsf{step}\ C\ (a, \mathsf{b2s}\ x)\ (b, s)\ C' \qquad \mathsf{s2b}\ s\ y}{\mathsf{step}\ \boxed{x} \!\!-\!\! C\ a\ b\ \boxed{y} \!\!-\!\! C'}$$

Figure 2 – LDDL semantics for a clock cycle

Gates and plugs are stateless: they always return an unchanged circuit. The rules for sequential and parallel compositions are the expected recursive definitions. The rule for $\boxed{x}$–C is more peculiar as it may change the state of the circuit (*i.e.,* the values of its memory cells). It makes use of:

- the function b2s which converts a Boolean into a signal (b2s(ff) = 0 and b2s(tt) = 1);
- the predicate s2b which relates a signal to a Boolean (s2b(0) = ff and s2b(1) = tt) and models the latching of a signal by a cell. We define it as a predicate and not a function because, as we see in the following section, we also use it for faulty signals which are latched non deterministically as tt or ff.

The reduction of C applied to the inputs a and the signal corresponding to x returns (b, s) (s being a signal) and a new circuit C'. The output of $\boxed{x}$–C applied to a is b and the resulting circuit is $\boxed{y}$–C' where the boolean y represent the latching of s.

The predicate step describes the behavior of a circuit during one normal cycle. It is deterministic and could be described by a function as well. In the presence of faults, the semantics must be formalized using non-deterministic predicates.

The circuit behavior for any infinite stream of inputs is given by the co-inductive predicate $\mathsf{eval} : (\alpha \to \beta) \to \mathsf{Stream}\ \alpha \to \mathsf{Stream}\ \beta$ with

$$\text{Eval}\ \frac{\mathsf{step}\ C\ i\ o\ C' \qquad \mathsf{eval}\ C'\ is\ os}{\mathsf{eval}\ C\ (i : is)\ (o : os)}$$

The rule Eval states that if C applied to the inputs i returns after one clock cycle the outputs o and the circuit C' and if C' applied to the infinite stream of inputs is returns the output stream os then the evaluation of C with the input stream $(i : is)$ returns the output stream $(o : os)$.

LDDL has several clear benefits. As already pointed out, a key feature is that any LDDL expression is a valid circuit. Furthermore, its combinator (variable-less) nature prevents the semantics to have to deal with bindings and environments. It avoids many administrative matters (reads, updates, well-formedness of environments) and simplifies enormously formal proofs as already pointed out in [8].

3 Formalisation of SETs and Fault Models

There are two main types of soft errors caused by particle strikes: SEU (*i.e.,* bit-flips in flip-flops) and SET (*i.e.,* glitches propagating in the combinational circuit). An SEU can be modeled by changing the value of an arbitrary memory cell during a clock cycle. An SET in a combinational circuit can lead to the non-deterministic corruption of any memory cell connected (by a purely combinational path) to the place where the SET occurred. Since an SET may potentially lead to several bit-flips, SETs subsumes SEUs. We focus here on this more general kind of fault.

3.1 Modeling SETs

In [25, 26], experiments show that a glitch propagates in the forward logic cone only, *i.e.,* logic gates located downstream of the signal. We may consider that an SET occurs only when a particle strikes a component (gate or cell): it then produces a glitch at its output that propagates forward where it may be latched and corrupt several cells. A second, more general view, is to consider that a strike on a connection may induce a glitch at the output of the source (gate or cell) of that connection which then propagates forward. For instance, a strike on the connection in Fig. 3

may corrupt the cells c_1 and c_2 by propagating backward up to the NOT gate and then forward. In both views, all possible SETs can be modeled by introducing a glitch at the output of a gate or a cell. All possible SETs on the small circuit of Fig. 3 are modeled by introducing a glitch at the output of either the NOT gate, the c_1 or c_2 cells. We see in Sec. 4 that these two views give rise to different fault models for SEMTs.

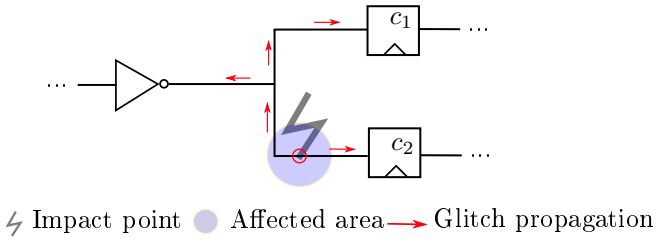


Figure 3 – An SET on a connection

In a well-formed circuit, any connection comes from a single source being either a gate, a cell or a primary input. To simplify the presentation and to model all possible SETs in the same way, we introduce a YES gate, represented as —▷—, as a source of all primary inputs (see Fig. 10 or Fig. 11) This gate, often called *buffer* gate, can be expressed in LDDL as NOT -○- NOT. It does not change the functionality of the circuit. All connections have now a source components and any SET can be modeled by introducing a glitch at the output of a gate or cell. Without YES gates, SETs on primary inputs would have to be modeled specifically.

We use signals that can take 3 values: 0, 1, or a glitch written $\lightning$. Glitches propagate through plugs and gates (*e.g.*, $\text{NOT}(\lightning) = \lightning$, $\text{AND}(1, \lightning) = \lightning$). Glitches can be also logically masked (*e.g.*, $\text{AND}(0, \lightning) = 0$, $\text{OR}(1, \lightning) = 1$). The majority voter Voter31 depicted in Fig. 4 can be defined as the following LDDL expression of type $((\omega * \omega) * \omega) \to \omega$:

$$\begin{array}{c}\lVert \text{FORK}, \text{FORK} \rVert \text{ -○- LSH -○- } \lVert \text{ID}, \text{LSH} \rVert \text{ -○- } \lVert \text{ID}, \text{RSH} \rVert \text{ -○- SWAP} \rVert \text{ -○- RSH} \rVert \\ \text{-○- } \lVert \text{AND}, \lVert \text{AND}, \text{AND} \rVert \rVert \text{ -○- } \lVert \text{ID}, \text{OR} \rVert \text{ -○- OR}\end{array}$$

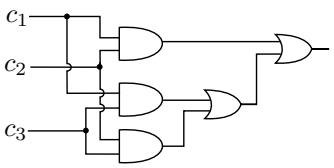


Figure 4 – Voter31: A majority voter

It is easy to check that Voter31 masks a glitch on one of its inputs provided the two other inputs are identical. For instance, $\text{Voter31}((\lightning, 0), 0) = 0$ (the first AND gate masks the glitch) and $\text{Voter31}((\lightning, 1), 1) = 1$ (the last OR gate masks the glitch).

If a corrupted signal is not logically masked and reaches a cell, it is latched non-deterministically as tt or ff. This is specified by the predicate s2b such that

$$\text{s2b } 0 \text{ ff} \quad \wedge \quad \text{s2b } 1 \text{ tt} \quad \wedge \quad \text{s2b } \lightning \text{ ff} \quad \wedge \quad \text{s2b } \lightning \text{ tt}$$

The propagation of SETs and the associated fault models are specified by semantics relations presented next.

3.2 Fault semantics for SETs

A cycle with an SET is represented as the inductive predicate $\mathsf{stepg}\ C\ a\ b\ C'$ that can be read as "after one cycle with an SET occurrence, the circuit C applied to the inputs a may produce the outputs b and the new circuit/state C'". The predicate stepg introduces non-deterministically a glitch after a *single* cell or gate and represents all possible SETs in the circuit. Its rules are gathered in Fig. 5.

$$\text{Gate}\ \frac{}{\mathsf{stepg}\ G\ a\ \lightning\ G} \qquad \text{Plug}\ \frac{[\![G]\!]a = b}{\mathsf{stepg}\ G\ a\ b\ G}$$

$$\text{SeqL}\ \frac{\mathsf{stepg}\ C_1\ a\ b\ C_1' \quad \mathsf{step}\ C_2\ b\ c\ C_2'}{\mathsf{stepg}\ (C_1 \multimap C_2)\ a\ c\ (C_1' \multimap C_2')} \qquad \text{SeqR}\ \frac{\mathsf{step}\ C_1\ a\ b\ C_1' \quad \mathsf{stepg}\ C_2\ b\ c\ C_2'}{\mathsf{stepg}\ (C_1 \multimap C_2)\ a\ c\ (C_1' \multimap C_2')}$$

$$\text{ParL}\ \frac{\mathsf{stepg}\ C_1\ a\ c\ C_1' \quad \mathsf{step}\ C_2\ b\ d\ C_2'}{\mathsf{stepg}\ [\![C_1, C_2]\!]\ (a,b)\ (c,d)\ [\![C_1', C_2']\!]} \qquad \text{ParR}\ \frac{\mathsf{step}\ C_1\ a\ c\ C_1' \quad \mathsf{stepg}\ C_2\ b\ d\ C_2'}{\mathsf{stepg}\ [\![C_1, C_2]\!]\ (a,b)\ (c,d)\ [\![C_1', C_2']\!]}$$

$$\text{CellC}\ \frac{\mathsf{stepg}\ C\ (a, \mathsf{b2s}\ x)\ (b,s)\ C' \quad \mathsf{s2b}\ s\ y}{\mathsf{stepg}\ \boxed{x}\text{—}C\ a\ b\ \boxed{y}\text{—}C'} \qquad \text{CellM}\ \frac{\mathsf{step}\ C\ (a, \lightning)\ (b,s)\ C' \quad \mathsf{s2b}\ s\ y}{\mathsf{stepg}\ \boxed{x}\text{—}C\ a\ b\ \boxed{y}\text{—}C'}$$

Figure 5 – LDDL semantics with SET

The rule (Gate) describes the introduction of a glitch after a logical gate. Each composition operator (sequential, parallel and memory cell) is associated with two rules which specify non-deterministic choices. The two rules for sequential composition represents two mutually exclusive cases where the SET occurs in the left sub-circuit (SeqL) or in the right one (SeqR). In both cases, the other sub-circuit is evaluated using the standard evaluation predicate step (*i.e.,* without introducing additional SET). The two rules for the parallel operator are similar. The rule (CellC) represents the case where an SET occurs inside C. The rule (CellM) represents the case where an SET occurs at the output of the memory cell x that is fed to C. If a circuit has n gates and m cells, stepg specifies $n+m$ possible SETs. When a glitch reaches a cell, it may introduce a bit upset or not as specified by the non deterministic predicate s2b. Therefore, stepg specifies also all possible cases of cell upsets after an SET.

Note that an SEU could be described along the same lines: no glitch would be introduced after gates and the rule (CellM) would be changed not to introduce a glitch but instead to non deterministically replace y by tt or ff.

3.3 Fault-models for SETs

Fault-models specify the kind and number of faults considered. For instance, $SET(n,k)$ is a fault model where at most n SETs can occur within k clock cycles. Most fault-tolerant techniques are designed to mask fault models of the form $SET(1,k)$. Masking more simultaneous SETs involves unrealistic levels of redundancy (*e.g.,* two SETs within the same cycle require 5 redundant copies to mask them by majority voting). Even in environments with high levels of ionizing radiations (*e.g.,* space, particle accelerators), k is considered to be larger than 10^{10} [3].

Fault models $SET(1,k)$ are expressed by the predicate $\mathsf{setk\_eval} : \mathsf{Nat} \to (\alpha \to \beta) \to \mathsf{Stream}\ \alpha \to \mathsf{Stream}\ \beta$ formally defined by the two following rules:

$$\text{SetN}\ \frac{\mathsf{step}\ C\ i\ o\ C' \qquad \mathsf{setk\_eval}\ (n-1)\ C'\ is\ os}{\mathsf{setk\_eval}\ n\ C\ (i:is)\ (o:os)}$$

$$\text{SetG}\ \frac{\mathsf{stepg}\ C\ i\ o\ C' \qquad \mathsf{setk\_eval}\ (k-1)\ C'\ is\ os}{\mathsf{setk\_eval}\ 0\ C\ (i:is)\ (o:os)}$$

The first argument of setk_eval plays the role of a clock counter; it is initialized to k. The rule (SetN) describes a normal cycle without SET regardless of the value of the counter (we assume here a substraction defined on natural numbers such that $0-1=0$). A glitch can occur only if the counter is 0 (rule (SetG)). This glitch is introduced in C by stepg and the counter is reset to enforce at least $k-1$ normal execution steps (SetN) before (SetG) can be considered again.

4 Characterization and formalisation of SEMTs

A Single Event Multiple Transient (SEMT) is a single event that affects several nearby elements and produces several transients. A first point of view is to consider these elements to be nearby gates or cells the outputs of which would then produce a glitch. As already discussed for SETs, a second, more general, point of view is to consider also nearby connections and their corresponding source components (which may not be close).

Consider Fig. 6 and assume that all logic gates and memory cells are remote. With the first viewpoint, a strike may only produce an SET at the output of a single component (c_1, c_2, c_3, c_4 or one of the two NOT gates). At the worst case, an SET can corrupt only two cells when it hits a NOT gate (c_1 and c_2 or c_3 and c_4). With the second viewpoint, if a particle strikes two nearby connections (as illustrated in Fig. 6) then two glitches propagate from their source components (the two NOT gates) and possibly corrupt the four cells.

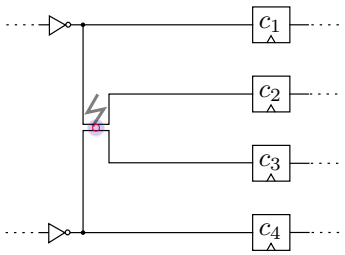


Figure 6 – Possible effects of SEMTs

In the following, we adopt the second point of view and take into account all nearby gates, cells and wires. This model clearly subsumes the first viewpoint. Furthermore, it also takes into account some crosstalk effects (capacitive or inductive coupling) where a glitch may affect unrelated but close connections [2].

We assume that a particle hitting a circuit has an influence over all the elements (gates, cells, wires) within a fixed radius from the point of impact. Experiments have shown that for neutrons the radius of the zone is between 500nm and 1000nm for circuits etched in 15nm and 45nm [16, 12].

4.1 Neighbor sets

To model a disturbance affecting several components, the layout of the circuit must be known and taken into account. We assume a 2D circuit with coordinates of the form (x, y). The layout is represented by a function taking the coordinates of a strike (a, b) and returning the set of components (gates and cells) that can be affected assuming a radius of influence r.

We refer to these sets as *neighbor sets*. An SEMT is described by a neighbor set and is modeled by introducing a glitch at the output of each of the gates and cells of that set.

Computing neighbor sets of a circuit implies knowing not only the position of the components but the complete layout of the circuit with the position of all its connections. This knowledge is formalized by the function $\mathcal{S}_r(P)$ which returns the set of connections close to the coordinates P *i.e.,* within a radius r. Let XY be the connection from component X to component Y then $XY \in \mathcal{S}_r(a, b)$ if XY goes through a point (x, y) within the radius r around the point (a, b) *i.e.,* such that

$$(x - a)^2 + (y - b)^2 \leq r^2$$

The function $\mathcal{I}_r(P)$ returning the neighbor set associated to the coordinate P is defined as

$$\mathcal{I}_r(P) = \{X \mid XY \in S_r(P)\}$$

The neighbor set $I_r(P)$ is made of the gates or cells having an outgoing connection going close to the coordinate P. If the impact occurs right on a component X at position P we assume that $\mathcal{S}_r(P)$ returns its output wire and that $X \in \mathcal{I}_r(P)$.

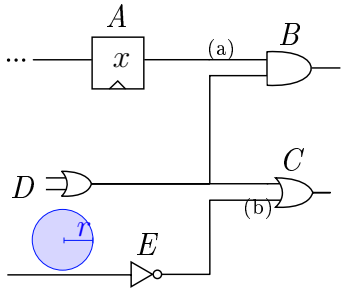


Figure 7 – Circuit with distant components and close connections

The circuit in Fig. 7 has four connections AB, DB, DC and EC. All individual components are distant from each other. The connections AB and DB come close to each other when reaching the B gate. Both can be impacted by the same SEMT and their source gates A and D are in the same neighbor set. Similarly DC and EC are close to each other when reaching the C gate so they belong to the same neighbor set. Of course, each component (gate or cell) belong also to a singleton neighbor set.

A neighbor set describes all possible SEMTs corrupting the outputs of some of its components. So, we only need to consider maximum neighbor sets *i.e.,* those that are not included in another one. A circuit whose layout is defined by a function $S_r(P)$ has as set of neighbor sets $\mathcal{I}_r(P)$ for all coordinates P. The set of its maximum neighbor sets is defined as

$$\mathcal{E} = \{\mathcal{I}_r(P) \mid \nexists P',\ I_r(P) \subset I_r(P')\}$$

For instance, the set of possible SEMTs of the circuit of Fig. 7 is described by the following set of maximum neighbor sets:

$$\mathcal{E} = \{\{A, D\}, \{C\}, \{B\}, \{D, E\}\}$$

4.2 Semantics and fault models for SEMTs

In Sec 3.2, we presented the predicate stepg that models all possible SETs by introducing a glitch at the output of a non deterministically chosen component. The semantics of circuits with an SEMT is described by the predicate $\mathsf{stepmg}_{\mathcal{H}}$ (Fig. 8) which may introduce a glitch at the outputs of the components of a neighbor set $\mathcal{H}$ non deterministically chosen in the set of the maximum neighbor sets.

Let $\mathcal{E}$ be the set of the maximum neighbor sets of C. The predicate $\mathsf{Stepmg}_{\mathcal{E}}$ describes the evaluation of C for all possible SEMTs described by $\mathcal{E}$. Such an evaluation starts by a non-deterministic choice of one of the neighbor sets of $\mathcal{E}$ (*i.e.,* a possible SEMT).

$$\text{StepM}\ \frac{\mathcal{H} \in \mathcal{E} \quad \mathsf{stepmg}_{\mathcal{H}}\ C\ a\ b\ C'}{\mathsf{Stepmg}_{\mathcal{E}}\ C\ a\ b\ C'}$$

The predicate $\mathsf{stepmg}_{\mathcal{H}}\ C\ a\ b\ C'$ can be read as "after one cycle with an SEMT impacting some (possibly all or none) of the gates and cells of $\mathcal{H}$, the circuit C applied to the inputs a may produce the outputs b and the new circuit/state C'". Its rules are gathered in Fig. 8.

$$\text{GateIn}\ \frac{G \in \mathcal{H}}{\mathsf{stepmg}_{\mathcal{H}}\ G\ a\ \lightning\ G} \qquad \text{Gate}\ \frac{[\![G]\!]a = b}{\mathsf{stepmg}_{\mathcal{H}}\ G\ a\ b\ G} \qquad \text{Plug}\ \frac{[\![P]\!]a = b}{\mathsf{stepmg}_{\mathcal{H}}\ P\ a\ b\ P}$$

$$\text{Seq}\ \frac{\mathsf{stepmg}_{\mathcal{H}}\ C_1\ a\ b\ C_1' \quad \mathsf{stepmg}_{\mathcal{H}}\ C_2\ b\ c\ C_2'}{\mathsf{stepmg}_{\mathcal{H}}\ (C_1 \multimap C_2)\ a\ c\ (C_1' \multimap C_2')} \qquad \text{Par}\ \frac{\mathsf{stepmg}_{\mathcal{H}}\ C_1\ a\ c\ C_1' \quad \mathsf{stepmg}_{\mathcal{H}}\ C_2\ b\ d\ C_2'}{\mathsf{stepmg}_{\mathcal{H}}\ [\![C_1, C_2]\!]\ (a,b)\ (c,d)\ [\![C_1', C_2']\!]}$$

$$\text{CellIn}\ \frac{\mathsf{stepmg}_{\mathcal{H}}\ C\ (a, \lightning)\ (b,s)\ C' \quad \mathsf{s2b}\ s\ y \quad \boxed{\cdot}\!\!-C \in \mathcal{H}}{\mathsf{stepmg}_{\mathcal{H}}\ \boxed{x}\!\!-C\ a\ b\ \boxed{y}\!\!-C'}$$

$$\text{Cell}\ \frac{\mathsf{stepmg}_{\mathcal{H}}\ C\ (a, \mathsf{b2s}\ x)\ (b,s)\ C' \quad \mathsf{s2b}\ s\ y}{\mathsf{stepmg}_{\mathcal{H}}\ \boxed{x}\!\!-C\ a\ b\ \boxed{y}\!\!-C'}$$

Figure 8 – LDDL semantics with SEMTs

The rules (GateIn) and (CellIn) introduce a glitch at the output of any gate or cell belonging to $\mathcal{H}$. The rules (Gate) and (Cell) can also be applied regardless of whether the component belongs to $\mathcal{H}$ or not. This models the non deterministic introduction of glitches.

The other rules as similar to the rules of step. Compared to the rules (Seq) and (Par) of stepg, $\mathsf{stepmg}_{\mathcal{H}}$ is applied recursively to both sub-circuits since the whole circuit must be traversed to introduce a glitch after all gates and cells belonging to $\mathcal{H}$.

The fault model $SEMT(1,k)$ describing at most one SEMT each k cycles is formalized by the predicate $\mathsf{semtk\_eval} : \mathsf{Set\ NS} \to \mathsf{Nat} \to (\alpha \to \beta) \to \mathsf{Stream}\ \alpha \to \mathsf{Stream}\ \beta$ in Fig. 9.

Compared to setk_eval (see Sec. 3), the predicate semtk_eval takes in addition the set of maximum neighbor sets $\mathcal{E}$ depending on the layout of C. The rule (SemtN) describing a normal cycle is similar to (SetN). Compared to (SetG), the rule (SemtG) chooses non deterministically a neighbor set $\mathcal{H}$ within $\mathcal{E}$. It is then used by the predicate stepmg to insert glitches.

This formalization can be used to prove fault-tolerance properties of specific circuits *w.r.t.* SEMTs. In that case, the first step should be to analyze the layout of the circuit in order to compute its set of maximum neighbor sets. However, our aim is to propose circuit transformations

$$\text{SemtN}\ \frac{\mathsf{step}\ C\ i\ o\ C' \quad \mathsf{semtk\_eval}_{\mathcal{E}}\ (n-1)\ C'\ is\ os}{\mathsf{semtk\_eval}_{\mathcal{E}}\ n\ C\ (i:is)\ (o:os)}$$

$$\text{SemtG}\ \frac{\mathcal{H} \in \mathcal{E} \quad \mathsf{stepmg}_{\mathcal{H}}\ C\ i\ o\ C' \quad \mathsf{semtk\_eval}_{\mathcal{E}}\ (k-1)\ C'\ is\ os}{\mathsf{semtk\_eval}_{\mathcal{E}}\ 0\ C\ (i:is)\ (o:os)}$$

Figure 9 – The $SEMT(1,k)$ fault model

to ensure fault-tolerance properties for *all* transformed circuits. To prove such properties, we will have to define generic layout constraints.

5 A modified TMR to tolerate SEMTs

Before describing sufficient and general constraints, we first present intuitively the standard full TMR and formalize it as a program transformation expressed on the LDDL syntax. We then examine the consequences of SEMTs on TMR circuits and discuss why they may not tolerate SEMTs. We finally propose a modified TMR transformation that, along with some layout constraints, masks the fault model $SEMT(1,2)$ *i.e.,* one SEMT every other cycle.

5.1 The standard full TMR transformation

The simplest form of triple modular redundancy amounts to triplicating the circuit and inserting a single majority voter at each primary output. It masks most SETs occurring in a combinational circuit. To remove the single point of failure, the final voter is sometimes also triplicated. For a sequential circuit this scheme is not sufficient since an SET can upset the memory of one copy which may stay corrupted until another SET corrupts a different copy. In such case, a triplicated voter would be able to mask two incorrected signals out of three.

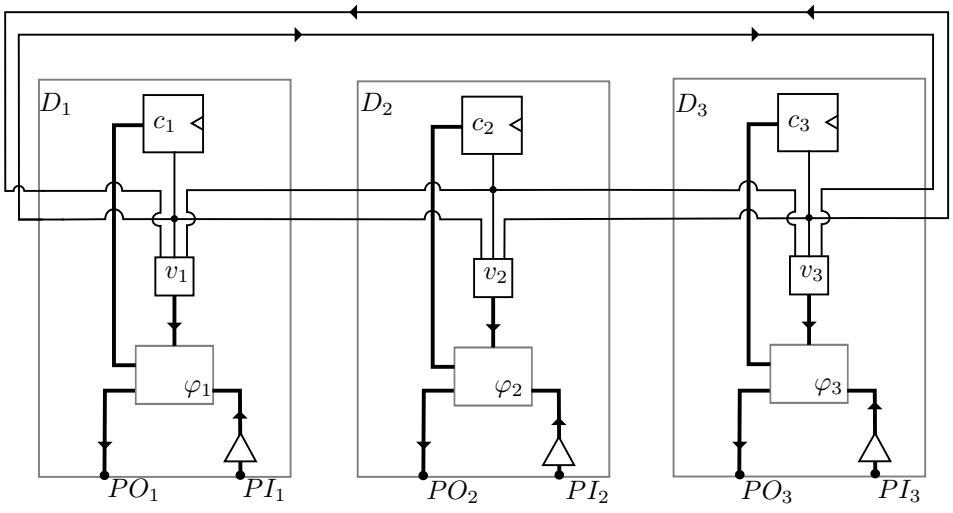


Figure 10 – Full TMR to mask all SETs

To provide full guarantees against SETs the circuit must be triplicated and a voter placed

after *each* memory cell in each copy/domain. This architecture, called full TMR, ensures that an SET can corrupt only a single domain and will be corrected by voting in the next cycle. Even with an SET every other cycle at least two primary outputs out of three are correct at any time. Fig. 10 represents full TMR where the φ_i's represent the combinational parts of the circuit; the c_i's and v_i's denote single memory cells and voters but can also be seen as memory and voter banks.

Note that interconnections between copies go directly to a voter. Hence, an SET in, say, D_1 may propagate to D_2 and/or D_3 but that glitch will be immediately masked by the voter v_2 and/or v_3. Nevertheless, such an SET may corrupt (possibly all) cells in D_1 as well as PO_1 but PO_2 and PO_3 remain correct. In the following cycle, the voters will restore correct values to the cells in D_1 and all primary outputs will be correct. The circuit is back in a sound state and able to mask another SET.

Full TMR cannot mask an SET every cycle. Take, for example, the case where an SET in φ_1 (see Fig. 10) corrupts c_1. In the next cycle, a second SET occurring at the output of c_2 will make all voters unable to mask the fault: their inputs from c_1 and c_2 may be incorrect (a wrong value from c_1 and a glitch from c_2). The three domains may become and stay corrupted forever.

The full TMR technique can be formalized by a program transformation on the syntax of LDDL. It takes a circuit of type $\alpha \to \beta$ and returns a circuit of type $(((\alpha * \alpha) * \alpha)) \to (((\beta * \beta) * \beta))$. Inputs/outputs are triplicated to play the role of the inputs/outputs of each copy.

$$\begin{array}{lcll}
\mathrm{TMR}(X) & = & [\![[\![X, X]\!], X]\!] & \text{with } X \text{ a gate or a plug} \\
\mathrm{TMR}(C_1 \multimap C_2) & = & \mathrm{TMR}(C_1) \multimap \mathrm{TMR}(C_2) & \\
\mathrm{TMR}([\![C_1, C_2]\!]) & = & \mathrm{s}_1 \multimap [\![\mathrm{TMR}(C_1), \mathrm{TMR}(C_2)]\!] \multimap \mathrm{s}_2 & \\
\mathrm{TMR}(\boxed{x}\!\!-\!\!C) & = & \boxed{x}\!\!-\!\!\boxed{x}\!\!-\!\!\boxed{x}\!\!-\!\!(\mathsf{vot} \multimap \mathrm{TMR}(C) \multimap \mathrm{s}_3) &
\end{array}$$

where s_1, s_2, s_3 are reshuffling plugs. Their are defined in LDDL so that:

$$\begin{array}{lcl}
\mathrm{s}_1\ (((s,t)(s,t)),(s,t)) & = & (((s,s),s),((t,t),t)) \\
\mathrm{s}_2(((s,s),s),((t,t),t)) & = & (((s,t)(s,t)),(s,t)) \\
\mathrm{s}_3\ (((s,t)(s,t)),(s,t)) & = & (((((s,s),s),t),t),t)
\end{array}$$

- A gate or plug X of type $\alpha \to \beta$ is simply triplicated in $[\![[\![X, X]\!], X]\!]$ which has type $(((\alpha * \alpha) * \alpha)) \to (((\beta * \beta) * \beta))$.

- The sequential composition amounts to triplicating the two sub circuits and composing them. Assuming that its two sub-circuits have types $C_1 : \alpha \to \gamma$ and $C_2 : \gamma \to \beta$, then $C_1 \multimap C_2$ has type $\alpha \to \beta$ and the triplicated input has type $((\alpha * \alpha) * \alpha)$ which is the input type expected by $\mathrm{TMR}(C_1)$. It returns $((\gamma * \gamma) * \gamma)$ which is the input type expected by $\mathrm{TMR}(C_2)$. The triplicated sequential composition has therefore the required type $((\alpha * \alpha) * \alpha) \to ((\beta * \beta) * \beta)$

- The parallel composition needs some reshuffling. Assuming that its two sub-circuits have types $C_1 : \alpha_1 \to \beta_1$ and $C_2 : \alpha_2 \to \beta_2$, then $[\![C_1, C_2]\!]$ has type $(\alpha_1 * \alpha_2) \to (\beta_1 * \beta_2)$. Its triplicated inputs $(((\alpha_1 * \alpha_2) * (\alpha_1 * \alpha_2)) * (\alpha_1 * \alpha_2))$ are first reordered by s_1 to pass the correct expected triplicated inputs to $\mathrm{TMR}(C_1)$ (*i.e.,* $((\alpha_1 * \alpha_1) * \alpha_1)$) and to $\mathrm{TMR}(C_2)$ (*i.e.,* $((\alpha_2 * \alpha_2) * \alpha_2)$). Likewise, the outputs $(((\beta_1 * \beta_1) * \beta_1) * ((\beta_2 * \beta_2) * \beta_2))$ are reordered by s_2 so that the triplicated parallel composition has the required type

$$(((\alpha_1 * \alpha_2) * (\alpha_1 * \alpha_2)) * (\alpha_1 * \alpha_2)) \to (((\beta_1 * \beta_2) * (\beta_1 * \beta_2)) * (\beta_1 * \beta_2))$$

- Each $\boxed{x}\!\!-\!\!C$ is replaced by three cells followed by a triplicated voter vot, the triplicated circuit $\mathrm{TMR}(C)$ and some reshuffling. Assuming that $\boxed{x}\!\!-\!\!C$ has type $\alpha \to \beta$ then C has

type $(\alpha * \omega) \to (\beta * \omega)$. After the triplication of the input and cell, vot takes an input of type $(((((\alpha * \alpha) * \alpha) * \omega) * \omega) * \omega)$. It is defined as

$$\mathsf{vot} \quad = \quad \textsc{lsh} \multimap \textsc{lsh} \multimap [\![\textsc{id}, \textsc{rsh} \multimap \textsc{fork} \multimap [\![\textsc{fork} \multimap [\![\mathsf{Voter31}, \mathsf{Voter31}]\!], \mathsf{Voter31}]\!]\!] \multimap s_2$$

It starts by reshuffling its input so that $((\omega * \omega) * \omega)$ can be sent to the three majority voters Voter31 (see Fig.4). The final plug s_2 reshuffles the result to provide $\textsc{Tmr}(C)$ with an input of type $(((\alpha * \omega) * (\alpha * \omega)) * (\alpha * \omega))$. $\textsc{Tmr}(C)$ returns a result of type $(((\beta * \omega) * (\beta * \omega)) * (\beta * \omega))$ which is then reshuffled by s_3 to feed back the three signals to the cells and to give $\textsc{Tmr}(\boxed{x} \text{–} C)$ the expected output type $((\beta * \beta) * \beta)$.

5.2 Layout constraints for Tmr and Semts

The key property of full Tmr is that each Set can only corrupt one copy that can be corrected by majority voting during the next cycle (supposed to be fault-free). In contrast, without additional layout constraints, an Semt may corrupt several copies of a full Tmr circuit at once by impacting several gates or cells belonging to different copies.

In the following, we say that two elements (gate, cell, wire) or domains are *distant* (resp. *close*) if they are more (resp. less) than $2r$ apart, r being the considered radius of a particle impact. The layout of the full Tmr circuit must satisfy some layout constraints:

Any two domains D_i and D_j $(i \neq j)$ should be distant from each other, otherwise both could be corrupted by the same Semt. We view primary inputs and outputs as sockets of their respective domains and constraints on them are already taken into account by constraints on domains. Assuming that the distance d between any two domains is greater than $2r$, these constraints are fulfilled by the circuit of Fig. 11. They are however not completely sufficient to prevent an Semt from corrupting several domains.

Since we consider Semt on wires, we should pay attention to the interconnections between different domains. The three inputs of a voter, say, v_1 which belong to D_1, are considered close from each other. They can therefore be striked by a single Semt modeled by a glitch at the outputs of their source components *i.e.,* c_1, c_2 and c_3. These glitches would propagate back to the three inputs of the three voters, making the three domains potentially corrupted.

It is not reasonable to impose that the inputs of a basic circuit such as a voter be separated by more than $2r$. Therefore, Tmr must be adapted to prevent such source of corruption. A simple solution is to protect each interconnecting wire by a yes gate preventing an Semt on a voter from propagating to other domains. In Fig. 11, the wires going to voters are protected by a yes gate inserted before leaving their source domain. Now, if an Semt corrupts the three inputs of v_1 it will be modeled as a glitch at the outputs of c_1 and the two corresponding yes gates of D_2 and D_3. The glitch from c_1 propagates to v_1, v_2 and v_3 while the two glitches from the yes gates in D_2 and D_3 can only propagate to v_1. The voter v_1 has its three inputs corrupted and so might be D_1. On the other hand, at most one input of v_2 and v_3 is corrupted; hence the glitch is masked and both D_2 and D_3 remain correct.

Interconnecting wires between two domains can be close to each other. An Semt on such wires, *e.g.,* between D_1 and D_2 $(D_1 \leftrightarrow D_2)$, is equivalent to introducing a glitch at the output of each corresponding yes gate. These two glitches will be masked by v_1 and v_2. However, interconnections $D_1 \leftrightarrow D_2$, $D_1 \leftrightarrow D_3$ and $D_2 \leftrightarrow D_3$ should not all three close to each other. Indeed, an Semt introducing a glitch at the output of all yes gates could corrupt the three domains during the same cycle.

To summarize, the required adjustments and layout constraints on Tmr to tolerate the fault model $\mathit{SEMT}(1, 2)$ of Sec. 4.2 are:

- each interconnection between two domains leaves its domain by crossing a yes gate;

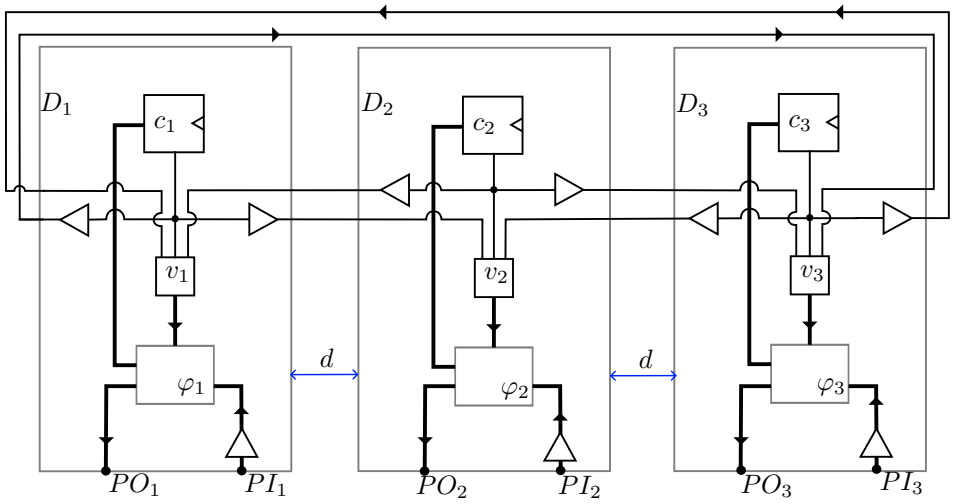


Figure 11 – A TMR circuit masking all SEMTs ($d > 2r$)

- the three copies/domains D_1, D_2 and D_3 are distant from each other;
- the interconnections between D_i and D_j are distant from D_k, from the interconnections between D_i and D_k and from the interconnections between D_j and D_k (with i, j and k distinct from each other).

Fig 11 shows a triplicated circuit with a layout respecting these constraints. The last constraint implies that any SEMT on interconnections outside a domain will be represented by a neighbor set made only of the YES gates of two domains. This kind of fault is subsumed by an SEMT on the voters of one of these two domains.

5.3 A TMR transformation to tolerate SEMTs

The adaptation of TMR for SEMTs relies on a new version of the triplicated voter inserted after each triplicated cell. The new triplicated voter vot_+ is similar to the one used in Sec. 5.1 but includes the aforementioned YES gates. It is defined as

$$\mathsf{vot}_+ = \text{LSH} \multimap \text{LSH} \multimap \text{FORK} \multimap [\![\text{FORK} \multimap [\![[\![\text{ID}, \overset{2\to1}{\text{YES}}]\!], \overset{3\to1}{\text{YES}}]\!] \multimap \mathsf{Voter31}, [\![[\![\overset{1\to2}{\text{YES}}, \text{ID}]\!], \overset{3\to2}{\text{YES}}]\!] \multimap \mathsf{Voter31}]\!], [\![[\![\overset{1\to3}{\text{YES}}, \overset{2\to3}{\text{YES}}]\!], \text{ID}]\!] \multimap \mathsf{Voter31}]\!] \multimap s_2$$

Fig 12 provides a graphical representation of vot_+. Note that a gate $\overset{i\to j}{\text{YES}}$ is located in domain D_i as depicted in Fig. 11.

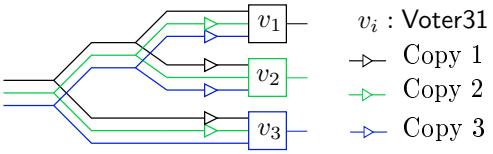


Figure 12 – Triplicated voter vot^+

Apart from the new voter, the modified TMR is specified by the same transformation as TMR.

$$\begin{array}{lcll} \text{TMR}^+(X) & = & [\![[\![X, X]\!], X]\!] & \text{with } X \text{ a gate or plug} \\ \text{TMR}^+(C_1 \multimap C_2) & = & \text{TMR}^+(C_1) \multimap \text{TMR}^+(C_2) & \\ \text{TMR}^+([\![C_1, C_2]\!]) & = & \text{S}_1 \multimap [\![\text{TMR}^+(C_1), \text{TMR}^+(C_2)]\!] \multimap \text{S}_2 & \\ \text{TMR}^+(\boxed{x}\!\!-\!\!C) & = & \boxed{x}\!\!-\!\!\boxed{x}\!\!-\!\!\boxed{x}\!\!-\!\!(\mathsf{vot}_+ \multimap \text{TMR}^+(C) \multimap \text{S}_3) & \end{array}$$

Of course, in addition, the layout of the triplicated circuit must respect the layout constraints of Sec. 5.2.

6 Correctness Properties and Proofs

Before stating correctness properties we must address two technical problems. First, since we want to prove that for *any* circuit C, $\text{TMR}^+(C)$ tolerates the fault model $SEMT(1, 2)$ (see Sec. 4.2). We cannot rely on precise layouts and must consider generic neighbor sets valid for any circuit satisfying the generic constraints listed above. Second, the LDDL language does not provide any easy way to distinguish between components of different copies of a TMR circuit. We must therefore redefine the fault semantics specifically for TMR circuits. We can then present the general property we want to prove. Describing the proof in details is out of the scope of this paper and would be tiresome. Instead, we describe its structure and the main lemmas.

The TMR^+ transformation, its fault-tolerance properties, and their proofs have all been specified and completed within the Coq proof assistant; they are available online [1].

6.1 Generic neighbor sets

We consider the largest neighbor sets based on the generic constraints of Sec. 5.2. This guarantees that any SEMT on any circuit will be represented by a neighbor set that is included in one of these sets. In other words, our fault model subsumes any possible SEMT of any triplicated circuit respecting the generic constraints.

Largest neighbor sets are defined only based on constraints expressed in terms of domains and interconnections between them. Two gates or cells belonging to the same domain D_i of a circuit $\text{TMR}^+(C)$ belong to the same neighbor set since there is no constraint preventing them from being close. So, all components of D_i belong to the same neighbor set $\mathcal{H}_i$. On the other hand, gates, cells, primary inputs and primary outputs of a domain must be distant from the components of the other two domains. They must belong to a different neighbor set. This leads us to define three neighbor sets $\mathcal{H}_1$, $\mathcal{H}_2$ and $\mathcal{H}_3$ representing/including of possible SEMTs corrupting one of the three domains D_1, D_2 and D_3 of a circuit $\text{TMR}^+(C)$. An SEMT on a circuit $\text{TMR}^+(C)$ respecting the layout constraints of Sec.5.2 is represented by a neighbor set $\mathcal{H}$ included in either $\mathcal{H}_1$, $\mathcal{H}_2$, or $\mathcal{H}_3$. Any SEMT impacting either D_1 (*i.e.,* potentially all gates of D_1) or the interconnections between D_1 and D_2 (*i.e.,* potentially all YES gates from D_1 to D_2 and all YES gates from D_2 to D_1) or the interconnections between D_1 and D_3 (*i.e.,* potentially all YES gates from D_1 to D_3 and all YES gates from D_3 to D_1) will have its neighbor set included in $\mathcal{H}_1$ with

$$\mathcal{H}_1 = \{\text{cells or gates of } D_1\} \cup \{\text{YES gates from } D_2 \text{ to } D_1\} \cup \{\text{YES gates from } D_3 \text{ to } D_1\}$$

The two other sets $\mathcal{H}_2$ and $\mathcal{H}_3$ that include all possible SEMTs impacting D_2 or D_3 are defined similarly.

6.2 A fault model for all $\text{TMR}^+(C)$

A generic fault model can be defined by first selecting non deterministically one of the three above sets $\mathcal{H}_i$ and non deterministically inserting a glitch at the output of members of that set. Those glitches propagate and corrupt the primary outputs of D_i and possibly all its memory cells. Since glitches are introduced non deterministically the predicate $\mathsf{stepmg}_{\mathcal{H}_i}$ describes the corruptions of all possible subsets of $\mathcal{H}_i$ and therefore all SEMTs within $\mathcal{H}_i$.

This approach also describes impossible SEMTs (*e.g.,* on distant components within a domain) for specific circuits. The important point is that it ensures that the fault-tolerance properties hold for all possible SEMTs that may occur for any specific layout respecting the constraints.

As stated above, the LDDL language does not provide any easy way to distinguish between distinct components of a circuit such as two AND gates or two cells. For this reason, we define the fault model specifically for circuits triplicated by TMR^+. By doing so, it is possible to distinguish between gates or cells belonging to or connecting specific domains.

The fault model $SEMT(1,2)$ is expressed by the predicate $\mathsf{semtk\_eval}$ applied to the set of neighbor sets previously defined $\mathcal{E} = \{\mathcal{H}_1, \mathcal{H}_2, \mathcal{H}_3\}$ and a triplicated circuit $\text{TMR}^+(C)$.

$$\text{SemtN}\ \frac{\mathsf{step}\ \text{TMR}^+(C)\ i\ o\ \text{TMR}^+(C') \quad \mathsf{semt12\_eval}_{\mathcal{E}}\ (n-1)\ \text{TMR}^+(C')\ is\ os}{\mathsf{semt12\_eval}_{\mathcal{E}}\ n\ \text{TMR}^+(C)\ (i:is)\ (o:os)}$$

$$\text{SemtG}\ \frac{\mathsf{stepmg}_{\mathcal{H}_i}\ \text{TMR}^+(C)\ i\ o\ C^T \quad \mathsf{semt12\_eval}_{\mathcal{E}}\ 1\ C^T\ is\ os \quad \mathcal{H}_i \in \mathcal{E}}{\mathsf{semt12\_eval}_{\mathcal{E}}\ 0\ \text{TMR}^+(C)\ (i:is)\ (o:os)}$$

The rule (SemtN) describes a normal cycle without SEMT regardless of the value of the counter. It returns a triplicated circuit $\text{TMR}^+(C')$. The semantics of a faulty execution step (SemtG) of a circuit $\text{TMR}^+(C)$ is allowed if the counter is 0. It starts with a non-deterministic choice of one neighbor set $\mathcal{H}_i$ within $\mathcal{E} = \{\mathcal{H}_1, \mathcal{H}_2, \mathcal{H}_3\}$. The predicate $\mathsf{stepmg}_{\mathcal{H}_i}$ may insert a glitch after all components belonging to $\mathcal{H}_i$. The resulting circuit C^T is a triplicated one but might not be strictly on the form $\text{TMR}^+(C')$ since some cells of D_i might have been corrupted and therefore may differ from their corresponding cells of the two other domains. The infinite execution continues with a counter sets at 1, which only prevents the next cycle to be a faulty one.

In the following, we specify stepmg when the domain D_1 has been chosen *i.e.,* $\mathsf{stepmg}_{\mathcal{H}_1}$. The two other cases $\mathsf{stepmg}_{\mathcal{H}_2}$ and $\mathsf{stepmg}_{\mathcal{H}_3}$ are similar and we do not describe them here. The predicate $\mathsf{stepmg}_{\mathcal{H}_1}$ is defined by instantiating the definitions given in Sec. 4.2 to circuits of the form $\text{TMR}^+(C)$.

Triplicated gates are evaluated according to their logic but the output of the first one (belonging to D_1 therefore to $\mathcal{H}_1$) is possibly replaced by a glitch. To model the non deterministic introduction of a glitch at the output of the components in $\mathcal{H}_1$ we use the predicate $\mathsf{introglitch}$ such that

$$\mathsf{introglitch}\ x\ x \quad \wedge \quad \mathsf{introglitch}\ x\ \lightning$$

The rule for evaluating triplicated gates is

$$\text{TGate}\ \frac{[\![G]\!]a_i = b_i \quad \text{for } i = 1,2,3 \qquad \mathsf{introglitch}\ b_1\ b_1'}{\mathsf{stepmg}_{\mathcal{H}_1}\ \text{TMR}^+(G)\ ((a_1,a_2),a_3)\ ((b_1',b_2),b_3)\ \text{TMR}^+(G)}$$

where b_1' can either be b_1 or $\lightning$. The evaluation of triplicated plugs is

$$\text{TPlug}\ \frac{[\![P]\!]a_i = b_i \qquad \text{for } i = 1,2,3}{\mathsf{stepmg}_{\mathcal{H}_1}\ \text{TMR}^+(P)\ ((a_1,a_2),a_3)\ ((b_1,b_2),b_3)\ \text{TMR}^+(P)}$$

The rules for triplicated sequential and parallel compositions are

$$\text{TSeq}\ \frac{\mathsf{stepmg}_{\mathcal{H}_1}\ \textsc{Tmr}^+(C_1)\ a\ b\ C_1^T \qquad \mathsf{stepmg}_{\mathcal{H}_1}\ \textsc{Tmr}^+(C_2)\ b\ c\ C_2^T}{\mathsf{stepmg}_{\mathcal{H}_1}\ \textsc{Tmr}^+(C_1 \multimap C_2)\ a\ c\ (C_1^T \multimap C_2^T)}$$

$$\text{TPar}\ \frac{\mathsf{stepmg}_{\mathcal{H}_1}\ \textsc{Tmr}^+(C_1)\ a\ c\ C_1^T \qquad \mathsf{stepmg}_{\mathcal{H}_1}\ \textsc{Tmr}^+(C_2)\ b\ d\ C_2^T}{\mathsf{stepmg}_{\mathcal{H}_1}\ \textsc{Tmr}^+([\![C_1, C_2]\!])\ (a,b)\ (c,d)\ (\textsc{s}_1 \multimap [\![C_1^T, C_2^T]\!] \multimap \textsc{s}_2}\ )$$

Note that the evaluation of $\textsc{Tmr}^+(C_1)$ or $\textsc{Tmr}^+(C_2)$ does not result in a circuit of the form $\textsc{Tmr}^+(C)$. If the initial circuit contains cells, then $\mathsf{stepmg}_{\mathcal{H}_1}$ will introduce glitches that may corrupt cells of D_1. The resulting circuit may have triplicated cells where the cell of D_1 is different from the two others and therefore cannot be defined as a circuit triplicated by $\textsc{Tmr}^+$.

The rule for cells may introduce a glitch at the output of the first cell and of the gates of the voter of D_1 (all belonging to D_1 therefore to $\mathcal{H}_1$). It may also introduce a glitch after the YES gates of D_2 and D_3 fed to the voter of D_1. The rule (TCell) specify these faults is an equivalent way as follows:

- as with the rule for gates, we make use of the predicate $\mathsf{introglitch}$ to model the non deterministic introduction of glitches;
- the triplicated circuit is in a consistent state; the content of each triplicated cell is the same value x which is turned into a signal s ($\mathsf{b2s}\ x\ s$);
- this signal may be replaced by a glitch for the first input of the voters of $D_2((s', s), s)$ and $D_3\ ((s'', s), s)$; note that this is equivalent to inserting a glitch after the two YES gates belonging to D_1;
- the voter of D_1 is not executed but its output may be replaced by a glitch (this is equivalent to inserting a glitch after all its gates, the corresponding YES gates of D_2 and D_3 and executing it). This glitch may appear in the first input (a_1, v_1') fed to the triplicated circuit $\textsc{Tmr}^+(C)$;
- after the execution of $\textsc{Tmr}^+(C)$ the feedback signals are transformed into booleans ($\mathsf{s2b}\ w_i\ y_i$) to be latched. The correctness proof will establish that $y_2 = y_3$ whereas y_1 is non deterministically either 0 or 1.

$$\text{TCell}\ \frac{\begin{array}{c}\mathsf{b2s}\ x\ s \quad \mathsf{introglitch}\ s\ s' \quad \mathsf{introglitch}\ s\ s'' \quad \mathsf{introglitch}\ v_1\ v_1' \\ \mathsf{step}\ [\![[\![\textsc{yes}, \textsc{id}]\!], \textsc{yes}]\!] \multimap \mathsf{Voter31}\ \ ((s', s), s)\ v_2\ \ [\![[\![\textsc{yes}, \textsc{id}]\!], \textsc{yes}]\!] \multimap \mathsf{Voter31} \\ \mathsf{step}\ [\![[\![\textsc{yes}, \textsc{yes}]\!], \textsc{id}]\!] \multimap \mathsf{Voter31}\ \ ((s'', s), s)\ v_3\ \ [\![[\![\textsc{yes}, \textsc{yes}]\!], \textsc{id}]\!] \multimap \mathsf{Voter31} \\ \mathsf{stepmg}_{\mathcal{H}_1}\ \textsc{Tmr}^+(C)\ (((a_1, v_1'), (a_2, v_2)), (a_3, v_3))\ (((b_1, w_1), (b_2, w_2)), (b_3, w_3))\ C^T \\ \mathsf{s2b}\ w_i\ y_i \quad \text{for } i = 1..3\end{array}}{\mathsf{stepmg}_{\mathcal{H}_1}\ \textsc{Tmr}^+(\boxed{x} \text{—} C)\ ((a_1, a_2), a_3)\ ((b_1, b_2), b_3)\ \boxed{y_1} \text{—} \boxed{y_2} \text{—} \boxed{y_3} \text{—} (\mathsf{vot} \multimap C^T \multimap \textsc{s}_3)}$$

6.3 Correctness properties

We outline the proof structure by presenting the main lemmas needed to prove the main property relating the execution of the source circuit without fault to the execution of the transformed circuit under the fault model *SEMT(1, 2)*. The main theorem states that $\textsc{Tmr}^+$ tolerates 1 SEMT every 2 cycles. More precisely, a circuit $\textsc{Tmr}^+(C)$ always produces at least 2 correct outputs during faulty cycles (modeled by $\mathsf{stepmg}_{\mathcal{H}_i}$) and 3 during normal cycles (modeled by step).

Most of the lemmas relate the states and executions of the source and transformed circuits. These relations are expressed as inductive predicates.

Normal execution

We say that a circuit C^T is a *consistent triplicated circuit* if there is a circuit C such that $\textsc{Tmr}^+(C) = C^T$. This implies in particular that all triplicated cells share the same value.

A first property is that on normal operation (step) a circuit and its triplicated version with related inputs produces related outputs and related circuits. Furthermore, under normal operation, a consistent triplicated circuit evolves into another consistent triplicated circuit. Formally,

Lemma 1. *For all circuits C_1, C_2 and signals a, b*

$$\mathsf{step}\ C_1\ a\ b\ C_2 \Rightarrow \mathsf{step}\ \textsc{Tmr}^+(C_1)\ ((a,a),a)\ ((b,b),b)\ \textsc{Tmr}^+(C_2)$$

Corruption confinement

A first key property is that an SEMT can corrupt only a single redundant copy and that such corruption stays confined in this copy/domain.

To express corruption confinement, we use a predicate relating source and transformed programs expressed on the syntax of LDDL. The corruption of a single copy of a transformed circuit C^T *w.r.t.* to its source circuit C is expressed by predicates $\overset{1}{\sim}$, $\overset{2}{\sim}$, and $\overset{3}{\sim}$. The relation $C \overset{i}{\sim} C^T$ can be read as "C^T is a triplicated version of C where only the values cells of D_i may differ from their corresponding cells in C". The predicate $\overset{i}{\sim}$ is defined inductively by the rules

$$\begin{array}{rcll}
X & \overset{i}{\sim} & [\![X, X]\!], X]\!] & \text{with } X \text{ a plug or gate} \\
C_1 \multimap C_2 & \overset{i}{\sim} & C_1^T \multimap C_2^T & \textit{iff} \quad C_1 \overset{i}{\sim} C_1^T \ \wedge\ C_2 \overset{i}{\sim} C_2^T \\
[\![C_1, C_2]\!] & \overset{i}{\sim} & s_1 \multimap [\![C_1^T, C_2^T]\!] \multimap s_2 & \textit{iff} \quad C_1 \overset{i}{\sim} C_1^T \ \wedge\ C_2 \overset{i}{\sim} C_2^T \\
(\boxed{x} \!\!-\!\! C) & \overset{i}{\sim} & (\boxed{x_1} \!\!-\!\! \boxed{x_2} \!\!-\!\! \boxed{x_2} \!\!-\!\! (\mathsf{vot}^+ \multimap C^T \multimap \textsc{s}_3)) & \textit{iff} \quad C \ \overset{i}{\sim}\ C^T \ \wedge\ (j \neq i \Rightarrow x_j = x)
\end{array}$$

The last rule states that $\boxed{x} \!\!-\!\! C$ is in relation $\overset{i}{\sim}$ with its transformed version if and only if $C \overset{i}{\sim} C^T$ and the value of cells of domains other than D_i are equal to x. The other rules just check recursively the relationship.

In the following, we write $\sim$ for the relation $\overset{1}{\sim} \vee \overset{2}{\sim} \vee \overset{3}{\sim}$.

Assuming a consistent triplicated circuit, it must be shown that the cells and primary outputs that can be corrupted during a cycle modeled by stepmg belong to a single domain. If the SEMT occurs in D_i then the cells and outputs of the two other domains are correct.

Lemma 2. *For all circuits C, C', C'^T, signals a, b, o_1, o_2, o_3 and $i \in \{1,2,3\}$*

$$\begin{array}{rl}
& \mathsf{step}\ C\ a\ b\ C' \ \wedge\ \mathsf{stepmg}_{\mathcal{H}_i}\ \textsc{Tmr}^+(C)\ ((a,a),a)\ ((o_1,o_2),o_3)\ C'^T \\
\Rightarrow & C' \ \overset{i}{\sim} C'^T \ \wedge\ (\forall j \in \{1,2,3\}, j \neq i \Rightarrow\ o_j = b)
\end{array}$$

Recovery after corruption

The second key property states that a corrupted circuit always recovers after a non-faulty execution step. If the triplicated circuit has only one domain corrupted then it returns to a consistent triplicated circuit after a single normal cycle (step).

Lemma 3. *For all circuits C, C', C^T and signals a, b*

$$C \ \sim\ C^T \ \wedge\ \mathsf{step}\ C\ a\ b\ C' \ \Rightarrow\ \mathsf{step}\ C^T\ ((a,a),a)\ ((b,b),b)\ \textsc{Tmr}^+(C')$$

Main fault-tolerance theorem

Just as full TMR cannot tolerate two SETs in a row, TMR$^+$ cannot tolerate two SEMTs in a row. Indeed, a first SEMT might corrupt a cell in D_1 whereas an SEMT in the following cycle might corrupt the output signal of the corresponding cell in D_2. The corresponding voters would have two incorrect inputs and produce incorrect values, therefore corrupting cells and/or outputs in the three domains.

The fault-tolerance property relates infinite execution over streams. The input stream of the triplicated circuit is the input stream of the source circuit where each element is triplicated. We write $\mathsf{tripl}(i_1 :: i_2 :: \ldots :: i_n :: \ldots)$ for the stream

$$((i_1, i_1), i_1) :: (i_2, i_2), i_2) :: \ldots :: (i_n, i_n), i_n) :: \ldots)$$

The output streams of the source and transformed circuits are related by the following predicate

$$(o_1 :: o_2 :: \ldots :: o_n :: \ldots) \overset{s}{\sim} ((x_1, x_1'), x_1'') :: (x_2, x_2'), x_2'') :: \ldots :: (x_n, x_n'), x_n'') :: \ldots)$$

which holds is if each triplet $((x_i, x_i'), x_i'')$ has at least two members equal to o_i.

The main correctness theorem states that TMR$^+(C)$ circuits tolerate the fault model $SEMT(1, 2)$. For related input streams, the normal execution (eval) of the source circuit and the execution under the considered fault model ($\mathsf{semt12\_eval}$) of the transformed circuit give related output streams. It is expressed as follows.

Theorem 4. *For all circuit C, integer n, streams of signals i, o and o^T*

$$\mathsf{eval}\ C\ i\ o\ \wedge\ \mathsf{semt12\_eval}\ n\ \text{TMR}^+(C)\ (\mathsf{tripl}\ i)\ o^T \Rightarrow o\ \overset{s}{\sim}\ o^T$$

7 Related work

Due to the downscaling of transistor size, most SETs are expected to evolve into SEMTs. The first studies were concerned by modeling multiple transient faults and analyzing their effect on circuits depending on the particle energy and circuit technology (*e.g.,* [23, 15]). A few techniques have been considered to mitigate SEMTs: layout-aware placement and hardening techniques. Kiddie and Robinson [18] investigate different placement strategies and find that some improve SEMT resilience with minimal area and timing penalties compared to the standard placement. Hardening techniques include the use of well contacts [9] or guard rings [11] on selected sensitive areas to mitigate SEMTs. Georgakidis *et al.* [13] compare two techniques to mitigate SEMTs: spacing all logical components so that a particle can affect only one and the use of TMR. Spacing among all components is found to produce worse area and performance results than TMR. However, only combinational circuits are considered, triplication is used only on critical subparts of the circuit and a single voter is used (making the design vulnerable to an SEMT on that component). Most of these works are pragmatic and aim at decreasing probabilities of faults with small costs. Effectiveness of these solutions are always estimated as soft error rates or probabilities using simulation and/or fault injection. These approaches radically differ from our approach since our objective is to provide formal guarantees *w.r.t.* a fault model.

Most uses of formal methods on digital circuits have been devoted to the functional verification of circuits [14]. It is usually performed for specific circuits using model-checking or SAT solving techniques. However, such an approach is inappropriate to prove the correctness of a synthesis or transformation tool for *all* possible circuits; theorem proving must be used instead. Still, proof-assistants have been first used for the functional verification of specific circuits. Let us

cite, among many others, the application of ACL2 to prove the out-of-order microprocessor architecture FM9801 [28], HOL for the Uinta pipelined microprocessor [31], and Coq for an ATM Switch Fabric [10]. The language proposed by Braibant [5], somewhat close to our LDDL language, has been used to prove the correctness of parametric combinational circuits (*e.g.,* n bits adders).

Proof-assistants have also been used to certify tools used in circuit synthesis or hardware compilers. An old survey of formal circuit synthesis is given in [19]. More recently, S. Ray et al. proved circuit transformations used in high-level synthesis with ACL2 [27]. Several formally verified hardware compilers have been proposed. Braibant and Chlipala certified a compiler from a simplified version of BlueSpec to RTL in Coq [6]. Bourgeat *et al.* present a hardware compiler implemented in Coq for Kôika, an experimental hardware design language inspired by BlueSpec [4]. Lööw introduces Lutsig, a Verilog-to-netlist compiler verified in HOL [20].

This article is based on our previous work [8] which presented the LDDL language and its use to certify several circuit transformations for fault-tolerance of SETs. Contrary to most works that specify circuits within the logic of the prover, we use a hardware description language. This approach, known as deep-embedding, allows us to reason on circuits (gates and wires), to model Single-Event Transient (SET) as glitches occurring at specific places and to express fault-tolerance techniques as syntactic program transformations. To the best of our knowledge, our work remains the only one using formal methods to verify fault-tolerance properties of digital circuits.

8 Conclusions

Most evaluation of fault-tolerance techniques are based on experiments (*e.g.,* fault injection, irradiation, *etc.*) measuring the probabilities of specific circuits to mask faults. The relevance of these probabilities depends on the probabilities of the experiments themselves and their level of test coverage. We follow a different approach which consists in formally proving that a given fault-tolerance technique masks all possible faults of a given fault model. This formal proof is conducted using a proof-assistant and guarantees fault-tolerance properties. Of course, it does not ensure that the circuit tolerate any kind of faults but the probabilities of fault-tolerance depends only on the coverage of the fault model.

This work is an extension of the approach described in [8] to SEMTs. Here, we proposed adjustments and layout constraints to full TMR so that it can mask any SEMT every other cycle.

We first showed how to formalize SEMTs knowing the circuit layout. From the layout, we can deduce neighbor sets, which are the possible sets of components than can be impacted by an SEMT. A faulty cycle of a given circuit is formalized by selecting non deterministically a neighbor set and introducing a glitch at the output of all the members of that set. We then showed that the standard full TMR must be adapted and comply to some placement constraints in order to mask SEMTs. Our fault-tolerance properties proofs for SETs [8] were quite straightforward since a faulty cycle for any circuit was modeled by introducing a glitch non- deterministically after a single gate or cell. It was therefore described in the same way for all possible circuits. For SEMTs, the event depends on the layout of the circuit. We had to first describe maximum neighbor sets that describe any possible SEMTs for all circuits. Second, we had to describe a faulty cycle on TMR$^+$ circuits in order to introduce glitches only on the members of one of these maximum sets.

We pointed out several possible fault models for SEMTs. The weaker model assumed that an SEMT can only disturb adjacent components. The model we chose is more general since components with close outgoing connections are considered to be close as well and may be impacted by the same SEMT. With the first model full TMR with distant triplicated domains is

sufficient to mask SEMTs. The second fault model requires in addition to protect inter-domain connections with YES gates. In both cases, full TMR tolerates the total corruption of a domain and does not put any restriction on technology scaling. The key property is domain isolation, which is achieved by spacing out the three domains and protecting interconnections using YES gates.

The proof that TMR$^+$ tolerates SEMTs is around 700 lines of Coq (excluding comments and blank lines) and took only a few days to complete. It relied on the already developed library for LDDL (5000 lines describing its syntax, semantics, properties and taylor-made tactics) and a previous proof of TMR for SETs. The proofs themselves are, for the most part, straightforward inductions. The approach makes an essential use of dependent types in Coq that provided an elegant solution to ensure that all circuits were well-formed. The Coq files of the formalization, properties and proofs associated with this work (as well as formalization and proofs of three other fault-tolerance transformations for SETs) are available online [1].

Our approach is general and applicable to many other circuit transformations and well-known techniques used in circuit synthesis (*e.g.*, FSM-encoding). For instance, the fault-tolerance techniques based on time redundancy described in [7] could also be considered for an extension to SEMTs. More generally, proof-assistants are now sufficiently mature to consider the formal certification of the whole circuit synthesis tool chain including transformations and optimizations.

References

[1] Coq proofs of circuit transformations for fault-tolerance. available at https://team.inria.fr/spades/fthwproofs/, 2014-2023.

[2] A. Balasubramanian, O. A. Amusan, B. L. Bhuva, R. A. Reed, A. L. Sternberg, L. W. Massengill, D. McMorrow, S. A. Nation, and J. S. Melinger. Measurement and analysis of interconnect crosstalk due to single events in a 90 nm cmos technology. *IEEE Transactions on Nuclear Science*, 55(4):2079–2084, 2008.

[3] A. Bogorad et al. On-orbit error rates of RHBD SRAMs: Comparison of calculation techniques and space environmental models with observed performance. *IEEE Trans. on Nuclear Science*, 58(6):2804–2806, 2011.

[4] T. Bourgeat, C. Pit-Claudel, A. Chlipala, and Arvind. The essence of bluespec: a core language for rule-based hardware design. In A. F. Donaldson and E. Torlak, editors, *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation, PLDI 2020, London, UK, June 15-20, 2020*, pages 243–257. ACM, 2020.

[5] T. Braibant. Coquet: A coq library for verifying hardware. In *Proc. of Certified Programs and Proofs - CPP*, pages 330–345, 2011.

[6] T. Braibant and A. Chlipala. Formal verification of hardware synthesis. In *Computer Aided Verification*, volume 8044, pages 213–228. 2013.

[7] D. Burlyaev. *Design, Optimization, and Formal Verification of Circuit Fault-Tolerance Techniques*. Theses, Université Grenoble Alpes, Nov. 2015.

[8] D. Burlyaev and P. Fradet. Formal verification of automatic circuit transformations for fault-tolerance. In *2015 Formal Methods in Computer-Aided Design (FMCAD)*, pages 41–48, 2015.

[9] I. Chatterjee, S. Jagannathan, D. Loveless, B. L. Bhuva, S.-J. Wen, R. Wong, and M. Sachdev. Impact of well contacts on the single event response of radiation-hardened 40-nm flip-flops. In *2012 IEEE International Reliability Physics Symposium (IRPS)*, pages SE.4.1–SE.4.6, 2012.

[10] S. Coupet-Grimal and L. Jakubiec. Certifying circuits in type theory. *Formal Asp. Comput.*, 16(4):352–373, 2004.

[11] Y. Du, S. Chen, and J. Chen. A layout-level approach to evaluate and mitigate the sensitive areas of multiple sets in combinational circuits. *IEEE Transactions on Device and Materials Reliability*, 14(1):213–219, 2014.

[12] M. Ebrahimi, H. Asadi, R. Bishnoi, and M. B. Tahoori. Layout-based modeling and mitigation of multiple event transients. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 35(3):367–379, 2016.

[13] C. Georgakidis, G. I. Paliaroutis, N. Sketopoulos, P. Tsoumanis, C. Sotiriou, N. Evmorfopoulos, and G. Stamoulis. A layout-based soft error rate estimation and mitigation in the presence of multiple transient faults in combinational logic. In *2020 21st International Symposium on Quality Electronic Design (ISQED)*, pages 231–236, 2020.

[14] A. Gupta. Formal hardware verification methods: A survey. *Form. Methods in System Design*, 1(2-3):151–238, Oct. 1992.

[15] R. Harada, Y. Mitsuyama, M. Hashimoto, and T. Onoye. Neutron induced single event multiple transients with voltage scaling and body biasing. In *2011 International Reliability Physics Symposium*, pages 3C–4. IEEE, 2011.

[16] E. Ibe, H. Taniguchi, Y. Yahagi, K.-i. Shimbo, and T. Toba. Impact of scaling on neutron-induced soft error in SRAMs from a 250 nm to a 22 nm design rule. *IEEE Transactions on Electron Devices*, 57(7):1527–1538, 2010.

[17] G. Jones and M. Sheeran. Designing arithmetic circuits by refinement in Ruby. *Sci. Comput. Program.*, 22(1-2):107–135, 1994.

[18] B. T. Kiddie and W. H. Robinson. Alternative standard cell placement strategies for single-event multiple-transient mitigation. In *2014 IEEE Computer Society Annual Symposium on VLSI*, pages 589–594, 2014.

[19] R. Kumar, C. Blumenröhr, D. Eisenbiegler, and D. Schmid. Formal synthesis in circuit design. A classification and survey. In *FMCAD*, pages 294–309, 1996.

[20] A. Lööw. Lutsig: a verified verilog compiler for verified circuit development. In C. Hritcu and A. Popescu, editors, *CPP '21: 10th ACM SIGPLAN International Conference on Certified Programs and Proofs, Virtual Event, Denmark, January 17-19, 2021*, pages 46–60. ACM, 2021.

[21] Coq development team. The coq proof assistant, software and documentation available at http://coq.inria.fr/, 1989-2023.

[22] N. Miskov-Zivanov and D. Marculescu. Multiple transient faults in combinational and sequential circuits: A systematic approach. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 29(10):1614–1627, 2010.

[23] N. Miskov-Zivanov and D. Marculescu. Multiple transient faults in combinational and sequential circuits: A systematic approach. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 29(10):1614–1627, 2010.

[24] S. Mitra, N. Seifert, M. Zhang, Q. Shi, and K. S. Kim. Robust system design with built-in soft-error resilience. *IEEE Computer*, 38(2):43–52, Feb. 2005.

[25] G. I. Paliaroutis, P. Tsoumanis, N. Evmorfopoulos, G. Dimitriou, and G. I. Stamoulis. SET Pulse Characterization and SER Estimation in Combinational Logic with Placement and Multiple Transient Faults Considerations. *Technologies*, 8(1):5, 2020.

[26] N. P. Rao and M. P. Desai. Quantification of the likelihood of single event multiple transients in logic circuits in bulk CMOS technology. *Microelectronics Journal*, 72:86–99, 2018.

[27] S. Ray, K. Hao, Y. Chen, F. Xie, and J. Yang. Formal verification for high-assurance behavioral synthesis. In *Int. Symposium on Automated Technology for Verification and Analysis*, pages 337–351, 2009.

[28] J. Sawada and W. A. Hunt Jr. Verification of FM9801: An out-of-order microprocessor model with speculative execution, exceptions, and program-modifying capability. *Formal Methods in System Design*, pages 187–222, 2002.

[29] M. Sheeran. muFP, A language for VLSI design. In *LISP and Functional Programming*, pages 104–112, 1984.

[30] J. von Neumann. Probabilistic logic and the synthesis of reliable organisms from unreliable components. *Automata Studies*, pages 43–98, 1956.

[31] P. J. Windley and M. L. Coe. A correctness model for pipelined multiprocessors. In *Theor. Provers in Circuit Design*, pages 33–51, 1994.

RESEARCH CENTRE
GRENOBLE – RHÔNE-ALPES

Inovallée
655 avenue de l'Europe Montbonnot
38334 Saint Ismier Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399